Elemi algoritmusok
Számjegyek és osztathatóság

dr. Ionescu Klára

clara@cs.ubbcluj.ro

claraionescu06@gmail.com

Mit nevezünk elemi algoritmusnak?

- Egyszerű „recept” egy adott feladat megoldására, ahol:
  - Nincs szükség összetett adatszerkezetre
  - Aritmetikai, logikai és relációs műveleteket alkalmazunk
  - Az eredmény könnyen generálható, értelmezhető
- Ugyanakkor: az illető feladat megjelenhet más feladat részfeladataként
- Általános megoldást adunk, ahhoz, hogy újra-felhasználható legyen

Alapszabályok

- Minden feladatot részfeladatokra bontunk, a részfeladatok megoldásait alprogramok (al-algoritmusok) formájában adjuk meg
- ***Egy alprogram = egy funkció***
- ***Egy alprogram nem olvas be, nem ír ki, csak ha pontosan ez az egyetlen funkciója***
- A programegységek a ***paramétereik*** segítségével kommunikálnak egymással
- Minden alprogramnak pontosan annyi paramétere lesz, ***ahányra szüksége van*** (minden, amit kintről kap, és minden, amit a hívás helyére továbbítania kell)

1. Kedvcsináló: Elnökválasztás

Egy elnökválasztás alkalmával igen sok jelölt indul. Nem tudjuk hány, de biztos több, mint 1 millió. Minden elnökjelölthöz egy természetes számot rendeltek. Állapítsuk meg a győztest, ha van ilyen! (Ahhoz, hogy nyerjen valaki a szavazóknak több, mint a felének rá kell szavaznia.)

***Másként*:**

- Adva van nagyon sok nagy szám
- Ezeket nem lehet egy tömbben tárolni
- Meg kell találnunk, (ha létezik!), azt a számot, amely legalább a számok fele plusz 1-szer megtalálható az adott számok között

Példák

N = 10

1, 2, 1, 1, 1, 2, 2, 2, 1, 1	**Nyerő: 1**
1, 2, 1, 2, 1, 2, 1, 2, 1, 2	**Nem nyer senki**
1, 2, 1, 2, 1, 2, 1, 2, 1, 1	**Nyerő: 1**
1, 1, 1, 1, 2, 3, 4, 5, 1, 1	**Nyerő: 1**
1, 2, 3, 4, 5, 6, 1, 1, 1, 1	**Nem nyer senki**

Kérdés:

Mit lehet tenni egy számmal, ahhoz, hogy maximális értékű információt facsarjunk belőle?

```
Algoritmus Szavazás:
  Be: szám
  jelölt ← szám
  hány ← 1
  Amíg nincs vége az állománynak végezd el:
    Be: szám
    Ha szám = jelölt akkor
      hány ← hány + 1
    különben
        Ha hány = 0 akkor
          jelölt ← szám
          hány ← 1
        különben
          hány ← hány - 1
        vége(ha)
    vége(ha)
  vége(amíg)
```

```
hány ← 0
n ← 0
Amíg nincs vége az állománynak végezd el:
   Be: szám
   n ← n + 1
    Ha szám = jelölt akkor
      hány ← hány + 1
    vége(ha)
vége(amíg)
Ha hány > n div 2 akkor
  Ki: 'Nyertes a ',jelölt,'!'
különben
  Ki: 'Sajnos nem nyert senki!'
vége(ha)
Vége(algoritmus)
```

szavazat	jelölt	hány
1	1	1
1		2
2		1
3		0
2	2	1
1		0
1	1	1
2		0
3	3	1
1		0
1	1	1

- Mit ábrázol a ***jelölt*** változó?
- Mi a jelentése a ***hány*** változónak?
- Miért különböző színűek a ***szavazat*** oszlopban az egyes számok?

2. Két természetes szám legnagyobb közös osztója

Határozzuk meg két természetes szám legnagyobb közös osztóját!

Megoldás

Legyen a két szám: ***a*** = 24 és ***b*** = 36

24 = 0 * 36 + 24 (Mivel az ***a / b*** egész osztás maradéka nem 0, felírjuk 36-ot)

36 = 1 * 24 + 12 (A maradék most sem 0, folytatjuk az osztást)

24 = 2 * 12 + 0 (A maradék 0, megvan a legnagyobb közös osztó: 12)

Általánosítva: a = q * b + r

Mivel tulajdonképpen egy *közös* osztót keresünk, ez felírható így is:

osztó * (a/osztó) = osztó * (q * b + r)/osztó = osztó * ((q * b)/osztó) + osztó * (r/osztó)

Az utolsó relációban látszik, hogy az ***osztó*** a ***b***-nek és ***r***-nek is osztója, ezért dolgozhatunk a következő lépésben ***a*** és ***b*** helyett ***b***-vel és ***r***-rel.

A feladat átalakult: már nem ***a*** és ***b*** lnko-ját keressük, hanem ***b*** és ***r*** lnko-ját, de ez egyben ***a*** és ***b***-nek is lnko-ja

```
Algoritmus Lnko(a, b, lnko):
  // Funkció: meghatározza a és b legnagyobb közös osztóját
                                        // Bemeneti paraméterek: a, b
                                          // Kimeneti paraméter: lnko
   Amíg a ≠ b végezd el:                    // ismételt kivonásokkal
        Ha a > b akkor
              a ← a – b
        különben
              b ← b – a
        vége(ha)
   vége(amíg)
   lnko ← a          // kimeneti paraméter nélkül: „visszatérít a”
Vége(algoritmus)
```

Kivonások helyett osztásokkal

```
Algoritmus Lnko(a, b, lnko):
  // Funkció: meghatározza a és b legnagyobb közös osztóját
                                  // Bemeneti paraméterek: a, b
                                    // Kimeneti paraméter: lnko
    Ismételd                                  // ismételt osztásokkal
        maradék ← a mod b
        a ← b
        b ← maradék
    ameddig maradék = 0    // kilépünk, amikor maradék = 0
    lnko ← a          // kimeneti paraméter nélkül: „visszatérít a”
Vége(algoritmus)
```

3. Mi a hatása?

```
Algoritmus Mi_ez(a, b, érték):
    Amíg a mod b ≠ 0 végezd el:
        a ← b
        b ← a mod b
    vége(amíg)
    érték ← b
Vége(algoritmus)
```

Úgy néz ki, mintha Euklidész algoritmusa lenne, de vegyük észre, hogy, mivel nincs segédváltozó, b kiszámolásakor a-nak megváltoztatott értékét használjuk. Ezért, bármely a és b értékek esetében b 0 lesz az első lépésben, így az Amíg feltételében már 0-val osztunk. Tehát a hatás: hibaüzenet (Nullával osztás).

4. Fordított szám

„Fordítsunk” meg adott (legtöbb 9 számjegyű) természetes számot! (Generáljuk az adott szám számjegyeit az eredetivel fordított sorrendben tartalmazó számot!)

Elemzés

Az ***újszám*** kezdeti értéke nulla, majd minden lépésben meghatározzuk ***újszám*** új értékét a *Horner* sémával.

```
Algoritmus fordítottSzám(szám, újSzám):
                                  // bemeneti adat: szám, kimeneti adat: újSzám
   újSzám ← 0
   Amíg szám ≠ 0 végezd el:
      újSzám ← újSzám * 10 + szám mod 10          // Horner séma
      szám ← szám div 10
   vége(amíg)
Vége(algoritmus)
```

5. Palindromszám

Adott természetes számról döntsük el, hogy palindromszám-e vagy sem! Egy számot *palindromszám*nak (vagy tükörszámnak) nevezünk, ha egyenlő a „fordított”-jával, vagyis azzal a számmal, amelyet a szám számjegyei fordított sorrendben alkotnak.

Megoldás

- A számot számjegyekre bontjuk.
- Hasonlóan az előző algoritmushoz, a felbontással párhuzamosan felépítjük az új számot.
- Az új szám generálását ugyancsak a ***Horner-séma*** néven ismert módszer segítségével végezzük: ***újszám ← újszám * 10 + szj***.

5. Palindromszám

- Az algoritmus a számjegyeket úgy határozza meg, hogy ismételten osztja az eredeti számot 10-zel

$\Rightarrow$ az algoritmus végén az eredeti szám értéke 0.

De: nekünk szükségünk van az eredeti értékre ahhoz, hogy összehasonlíthassuk az új számmal!!!

- Ezért a beolvasott számról a feldolgozás előtt ***másolatot*** készítünk.

```
Algoritmus Palindrom(szám, p):
                          // bemeneti adat: szám, kimeneti adat: p
                  // p = igaz, ha szám palindrom, különben hamis
  másolat ← szám
  újszám ← 0
  Amíg szám > 0 végezd el:
    számjegy ← szám mod 10              // számjegyekre bontás
    újszám ← újszám*10 + szj                  // Horner séma
    szám ← szám div 10  // a feldolgozott számjegyet levágjuk
  vége(amíg)
  p ← újszám = másolat
Vége(algoritmus)
```

```
/* ugyanaz mint:
        Ha újszám = másolat akkor
            p ← igaz
        különben
            p ← hamis
         vége(ha)        de jobb!!!
*/
```

6. Törzstényezők

*Bontsuk törzstényezőkre az adott **n** számot!*

Elemzés

- A számot ismételten osztjuk, előbb 2-vel, majd 3-mal stb., ameddig az ***n*** 1-gyé nem válik.
- Minden osztóval addig osztunk amíg a maradék 0.
- Amint egy bizonyos osztó már nem osztja maradéktalanul a számunkat, kiírjuk (az osztások számát hatványként használva).
- A kiírást figyelmesen végezzük!
- Lehet így is: 2 2 3
- Elegánsabb így: 2^2 * 3

Példa

12	2
6	2
3	3
1	

```
Algoritmus Törzstényezők(n):
   osztó → 2                                        // Bemeneti adat: n
   Ismételd
      hatvány → 0
      Amíg n mod osztó = 0 végezd el:
         hatvány → hatvány + 1
         n → n div osztó
      vége(amíg)
      Ha hatvány ≠ 0 akkor
            Ki: osztó, '^', hatvány, , ,      // kiírás helyett el lehetne
            Ha n ≠ 1 akkor Ki: '* ,           // menteni egy tömbbe
         vége(ha)
      vége(ha)
      Ha osztó = 2 akkor osztó → osztó + 1
                   különben osztó → osztó + 2
   ameddig n = 1                                    // kilépünk, ha n = 1
Vége(algoritmus)
```

7. Tökéletes szám

Keressük meg és írjuk ki az első ***n*** (***n*** ≤ **4**) *tökéletes számot! Egy szám tökéletes, ha egyenlő a nála kisebb osztóinak összegével.*

Példa: **6 = 1 + 2 + 3**.

Elemzés:

- Az első szám amit vizsgálunk: 2
- Osztókat csak a szám feléig érdemes keresni
- Amint találunk egy osztót, hozzáadjuk egy segédváltozó értékéhez
- Ha ez a szám egyenlő a vizsgált számmal, találtunk egy tökéletes számot.
- Ha implementáljuk a következő algoritmust, és megpróbáljuk futtatni **n** = 5-re, észlelni fogjuk, hogy a program nagyon sokat „gondolkozik”.

```
Algoritmus TökéletesSzámokKiírása(hány):
        // Bemeneti adat: hány, a kiírandó tökéletes számok száma
  darab ← 0; szám ← 2          // még nincs egy tökéletes szám sem
  Amíg darab < hány végezd el:
    osztókÖsszege ← 1                              // az első osztó az 1
    ngy ← négyzetgyök(szám)
    Minden osztó = 2, ngy végezd el:
      Ha szám mod osztó = 0 akkor
        osztókÖsszege ← osztókÖsszege + osztó // osztók összege
        Ha osztó ≠ szám div osztó akkor
                                        // ha a szám nem négyzetszám
            osztókÖsszege ← osztókÖsszege + szám div osztó
        vége(ha)
      vége(ha)
    vége(minden)
```

```
Ha osztókÖsszege = szám akkor
        Ki: szám
        darab ← darab + 1
    vége(ha)
    szám ← szám + 1          // A következő vizsgálandó szám
  vége(amíg)
Vége(algoritmus)
```

Érdekes: az alábbi képletekkel tökéletes számokat számíthatunk ki.

$6 = 2^1 * (2^2 - 1)$

$28 = 2^2 * (2^3 - 1)$

$496 = 2^4 * (2^5 - 1)$

$8128 = 2^6 * (2^7 - 1)$

…

$?? = 2^i * (2^{i+1} - 1)$, ahol $i = 2, 4, 6…$ és $t_1 = 6$

Lépések finomítása

- Bonyolultabb feladatok esetében az algoritmus leírása nem könnyű feladat.
- Ezért célszerű először a megoldást körvonalazni, és csak azután részletezni:

*A **feladat elemzése** során sor kerül:*

- a ***bemeneti és kimeneti adatok*** megállapítására;
- a megfelelő ***adatszerkezetek*** kiválasztására és megtervezésére;
- a feladat ***követelményeinek*** szétválasztására;
- a megoldási ***módszer*** megállapítására;
- a megoldás lépéseinek leírására.

Lépések finomítása

Lépések finomítása: folyamat, amely tart az algoritmus ***kezdeti vázlatától***, a ***végleges, kidolgozott algoritmus***ig.

- Kiindulunk a feladat ***specifikációi***ból és ***fentről lefele*** megtervezzük az algoritmust.
- Újabb meg újabb ***változatok***at dolgozunk ki, amelyek eleinte tartalmaznak magyarul leírt magyarázó sorokat, majd ezeket standard utasításokra írjuk át.
- Így az algoritmus változatai mindegyre bővülnek egyik változattól a másikig.

Végül: kódolás, majd tesztelés

8. Fibonacci-számok

*Generáljuk és írjuk ki az **n**-edik Fibonacci-számot!*

Elemzés

- ***Bemeneti adatok: n*** természetes szám
- ***Kimeneti adatok:*** az ***n***-edik Fibonacci-szám
- nincs szükség adatszerkezetre (a sorozatot nem szükséges tárolni)
- a feladat követelményeinek szétválasztása:
  - beolvasunk egy természetes számot,
  - kiszámítjuk az ***n***-edik Fibonacci-számot
  - kiírjuk az eredményt
- Minden *Fibonacci-szám* a megelőző kettőnek az összege

  $F_0 = 0, F_1 = 1, F_i = F_{i-1} + F_{i-2}$, ha $i \geq 2$.
- A *Fibonacci*-számok sorozata: 0,1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

A megoldás lépései

Algoritmus Fibonacci(n):
  Kezdőértékadások: a, b, c
  Ki: a, b, c
  k ← 3
  Amíg k < n **végezd el:**
    egy Fibonacci szám kiszámítása c-ben
    k ← k + 1
  Vége(amíg)
  visszatérít c
Vége(algoritmus)

Lépések finomítása

- Az első ***n*** *Fibonacci* szám generálásához elegendő három változó: ***a***, ***b*** és ***c***.
- A számsorozat egy új tagját (***c***) úgy állítjuk elő, hogy a már kiszámolt elemeket ***balra „toljuk”*** egy pozícióval.
- Így rendre megkapjuk ***a*** és ***b*** új értékét, majd ezek ismeretében kiszámíthatjuk ***c*** új értékét.
- Ezeket a lépéseket addig ismételjük, amíg a kért számú elemet elő nem állítottuk!
- A fenti vázlat módosítása: ha ***n = 1*** vagy ***n = 2***, csak egy, illetve két számot kell kiírnunk.
- A kidolgozásban leírjuk ***c*** új értékének kiszámítását

```
Algoritmus Fibonacci(n):
   a ← 0;  b ← 1                          { Bemeneti adat: n }
   Ha n = 1 akkor visszatérít a
    különben
        Ha n = 2 akkor  visszatérít b
        különben
           c ← a + b
           Ha n = 3 akkor visszatérít c
           különben
              k ← 3
              Amíg k < n végezd el:
                  a ← b; b ← c; c ← a + b
                  k ← k + 1
              vége(amíg)
              visszatérít c
           vége(ha)
      vége(ha)
Vége(algoritmus)
```

9. Fibonacci-szám-e?

*Állapítsuk meg egy adott (**n > 1**) számról, hogy Fibonacci-szám-e? Ha nem, bontsuk fel különböző (minimális számú) Fibonacci-szám összegére.*

Elemzés

Az előbbi algoritmussal generáljuk a *Fibonacci*-számokat, amíg egy olyan számot nem kapunk, amely nagyobb mint az adott szám vagy egyenlő vele.

- Ha ***c*** utolsó értéke *Fibonacci* szám, az algoritmus véget ér.
- Különben kiírjuk azt a legnagyobb *Fibonacci* számot, amely kisebb vagy egyenlő az adott számmal (ez a ***b***-ben található).
- Majd ***b***-t kivonjuk ***n*** aktuális értékéből.
- Ha ***b*** egyenlő ***n*** aktuális értékével, az algoritmus véget ér.
- Különben megismételjük az ellenőrzést.

Fibonacci-szám-e?

- Amíg ***n*** (amelyből minden lépésben kivonjuk a ***b*** *Fibonacci*-számot, amelyet kiírunk) pozitív, meghatározunk egy-egy újabb ***b*** *Fibonacci*-számot, amely kisebb vagy egyenlő az adott szám aktuális értékével.
- ***A feladat egyedi felbontást kér***. Ezt biztosítja az elemek növekvő sorrendben való megkeresése.
- Vegyük észre, hogy ez a megkötés nem teszi lehetővé, hogy az összegként felírt számban szerepeljen két egymás utáni eleme a *Fibonacci* sorozatnak.

```
Algoritmus Fibonacci_szám_e(n, k, tagok):      // az n számot vizsgáljuk
    a ← 0; b ← 1; c ← a + b
    k ← 0
    Amíg c < n végezd el:
       a ← b; b ← c
       c ← a + b                    // b a legnagyobb Fibonacci szám, amely < n
    vége(amíg)                                                  // c >= n
    Ha c ≠ n akkor
       k ← k + 1; tagok[k] ← b; n ← n – b
       Amíg n > 0 végezd el:  // amíg van még maradék az adott számból
          Amíg b > n végezd el:                          // elindulunk visszafele
             c ← b; b ← a
             a ← c – b              // b a legnagyobb Fibonacci szám, amely ≤ n
          vége(amíg)
          k ← k + 1; tagok[k] ← b; n ← n - b
       vége(amíg)
    vége(ha)
Vége(algoritmus)
```

10. Kettőhatvány

*Adott az **n** nem nulla természetes szám* ($0 < n \leq 10^9$). *Döntsük el, hogy kettőhatvány-e! Ha az adott szám nem kettőhatvány, bontsuk kettőhatványok összegére!*

Elemzés

- Generáljuk a kettőhatványokat, amíg kisebbek, mint ***n***
- Miután leállt a generálás, megvizsgáljuk, hogy a legutolsó kettőhatvány egyenlő-e ***n***-nel és így eldöntjük, a választ az kérdésre
- Ha az utolsó kettőhatvány nagyobb, mint ***n***, osztjuk kettővel, így megvan a felbontás első tagja
- Ezt kivonjuk ***n***-ből

Kettőhatvány

- A következő tagokat megkaphatnánk, ha újra végrehajtanánk az előbb leírt műveletsort, amíg ***n*** nullává nem válik
- Észrevétel: így újból elölről kezdenénk a kettőhatványok generálását; ha visszafele haladunk, osztva kettővel az aktuális kettőhatványt, hatékonyabb megoldáshoz jutunk
- Ugyanakkor, ha nem lenne a második követelmény, a döntést sokkal egyszerűbben így oldhatnánk meg:

```
Algoritmus kettőHatvány(n):
        visszatérít n & (n - 1) = 0 // & = bitenkénti „és” művelet
Vége(algoritmus)
```

- Ugyancsak bitenként hajtódnak végre a balra, illetve jobbra tolások:

<<: balra tolás (egyenértékű a 2-vel történő szorzással)

>>: jobbra tolás (egyenértékű a 2-vel történő osztással)

```
Algoritmus kettőHatvány(n, m):                    // m kimeneti paraméter
    m ← 1
    Amíg m < n végezd el:
        m ← m * 2
    vége(amíg)
    visszatérít m = n
Vége(algoritmus)

Algoritmus felbontás(n, m, k, tagok):// ha k értéke 0 marad ⇒ n kettőhatvány
    k ← 0
    Ha nem kettőHatvány(n, m) akkor
        Amíg n > 0 végezd el:
            Amíg m > n végezd el:
                m ← m / 2
            vége(amíg)
            k ← k + 1
            tagok[k] ← m
            n ← n – m
        vége(amíg)
    vége(ha)
Vége(algoritmus
```

11. Mi a hatása?

```
Algoritmus Mi_ez(n):
    a = 1
    b = 0
    Minden k = 1, n végezd el:
        b = a + b
        a = b – a
    vége(minden)
    visszatérít b
Vége(algoritmus)
```

Algoritmusok bonyolultsága

- Amikor egy algoritmus hatékonyságát vizsgáljuk, tulajdonképpen a bonyolultságát szeretnénk megállapítani.
- A bonyolultsághoz kapcsolódó információkról egy másik találkozásunk alkalmával fogok beszélni.
- Addig is: a bonyolultságot a Θ (teta) a O (nagy O) és az Ω (omega) függvényekkel jelöljük.
- Ezeknek az argumentumuk egy $g(n)$ függvény.
- A $g(n)$ lehet a lineáris függvény, a másodfokú függvény, a logaritmikus, az exponenciális stb.

Algoritmusok optimalizálása

- Optimalizáláskor egy ***kész*** algoritmus ***hatékonyságát próbáljuk növelni***.
- Tehát túl vagyunk a ***finomításon***.
- Az algoritmus teljesen kész, de elégedetlenek vagyunk a ***teljesítménnyel***.
- Ha az algoritmus egyetlen **Minden** ciklust tartalmaz, megvizsgáljuk, hogy valóban minden adatot föl kell dolgoznunk?
- Vajon nem állítható le a feldolgozás hamarabb?
- Néha észrevesszük, hogy esetleg létezik egy ***O(log n)*** bonyolultságú algoritmus…

Algoritmusok optimalizálása

- Ha egy algoritmus bonyolultsága *lineáris*, keresünk egy logaritmikusat (lásd a következőkben például a Gyorshatvány algoritmust), ha nem sikerül, megpróbáljuk csökkenteni a lépések számát!
- Csökkentjük a bonyolultságfüggvényt:
  1. Megpróbáljuk gyorsítani az algoritmust ***vagy***
  2. Kevesebb memóriaigénnyel akarjuk megoldani a problémát
  - ***Igazi optimalizálás akkor történik, ha anélkül sikerül jobb futási idejű algoritmust találni, hogy növelnénk a tárigényt; szerencsés esetben ez is csökken***.

12. Gyorshatvány

Adottak az $x > 0$ *valós és az* $n > 6$ *természetes számok. Számítsuk ki az* x^n *számot minél kevesebb szorzással!*

Elemzés

A feladat triviális megoldása $n - 1$ szorzással történne (**Minden** típusú struktúrával):

```
...
eredmény ← alap
Minden i=2, n végezd el:
  eredmény ← eredmény * alap
vége(minden)
...
```

Gyorshatvány

- A feladat megoldható kevesebb szorzással is, ha a ***kitevő***t kifejezzük a **2**-es számrendszerben, és x^n-*t* felírjuk x^{2^k} alakú számok szorzataként ($k > 0$).
- Az x^{2^k} számokat kiszámolhatjuk egymást követő négyzetre emelésekkel.

Példa

Mivel $9 = (1001)_2$,

$x^9 = x^8 \cdot x = ((x^2)^2)^2 \cdot x$

Ennek alapján a következő algoritmust már ***Abu-Ja far Mohammed ibn Mura al-Kvarîzmi*** (780 k.–850 k.) arab matematikus ismerte:

```
Algoritmus Gyorshatvány(alap, kitevő, eredmény):
                        { Bemenet: alap, kitevő. Kimenet: eredmény }
    eredmény ← 1
    Amíg kitevő > 0 végezd el:
        Ha kitevő páratlan akkor
            eredmény ← eredmény * alap
        vége(ha)
        alap ← alap * alap
        kitevő ← kitevő div 2
    vége(amíg)
Vége(algoritmus)
```

13. Példa: Orosz szorzás

Legyen a, b $\in$ ***N*****. Számítsuk ki a és b szorzatát! Tudjuk, hogy az orosz muzsik csak a következő műveleteket tudja elvégezni:*

- el tudja dönteni, hogy egy szám páros vagy páratlan;
- össze tud adni két számot;
- össze tud hasonlítani egy számot 0-val;
- felezni tud egy számot (elosztani 2-vel).

Példa

x	45	45	22	11	5	2	1
y	17	17	34	68	136	272	544
p	0	17		85	221		765

```
Algoritmus Orosz_szorzás(x, y, p):
                    // Kiszámítjuk x és y szorzatát, „szorzás” nélkül
                                        // Bemeneti adatok: x, y
                                    // Kimeneti adat: p (a szorzat)
    p ← 0
    Amíg x > 0 végezd el:
        Ha x páratlan akkor
            p ← p + y
        vége(ha)
        x ← x div 2
        y ← y + y
    vége(amíg)
Vége(algoritmus)
```

14. Prímszám-e?

*Döntsük el az **n** számról, hogy prímszám-e! (Létezik-e egész osztója önmagán és az 1-en kívül?)*

Elemzés:

- A prímszám definíciójából indulunk ki: ***egy szám akkor prím, ha pontosan két osztója van: 1 és maga a szám***
- ***Első ötlet:*** az algoritmus számolja meg az adott szám osztóit, elosztva ezt sorban valamennyi számmal **1**-től ***n***-ig
- A döntésnek megfelelő üzenetet az osztók száma alapján írjuk ki

Prím-e?

```
Algoritmus Prím_1(n):                          // naiv algoritmus!!!
   osztók_száma ← 0
   Minden osztó=1, n végezd el:
      Ha n mod osztó = 0 akkor
         osztók_száma ← osztók_száma + 1
      vége(ha)
   vége(minden)
   visszatérít osztók_száma = 2
         // ha az osztók száma 2, igaz értéket térítünk vissza,
         // különben hamis értéket
Vége(algoritmus)
```

Optimalizálások

Észrevétel: az osztások száma fölöslegesen nagy.

- Ezt a számot csökkenteni lehet, mivel ha **2** és ***n/2*** között nincs egyetlen osztó sem, akkor biztos nincs ***n/2*** és ***n*** között sem.
- Sőt, elég a szám ***négyzetgyökéig*** keresni a lehetséges osztót.

```
Algoritmus Prím_2(n, prím):
   prím ← igaz
   ngy ← négyzetgyök(n)
   Minden osztó=2, ngy végezd el:
      Ha n mod osztó = 0 akkor
         prím ← hamis
      vége(ha)
   vége(minden)
Vége(algoritmus)
```

További optimalizálások

Észrevétel: a logikai változónak lehet több munkát adni.

Leállítjuk a munkát abban a pillanatban, amikor találtunk egy osztót és a **prím** logikai változó hamissá vált.

- Lemondunk a **Minden** típusú ciklusról és egy **Amíg** vagy **Ismételd** típusú ciklussal helyettesítjük.
- Mivel ***n*** nem változik a ciklus-magban, a négyzetgyököt csak egyszer számítjuk ki, a ciklusba lépés előtt.

```
Algoritmus Prím_3(n, prím):
   prím ← igaz
   osztó ← 2
   ngy ← négyzetgyök(n)
   Amíg prím és (osztó ≤ ngy) v.e:
      Ha n mod osztó = 0 akkor
         prím ← hamis
      különben
         osztó ← osztó + 1
      vége(ha)
   vége(amíg)
Vége(algoritmus)
```

További optimalizálások

Észrevétel: ***a páros számok mind oszthatók 2-vel, és a 2 kivételével nem prímek!***

- De ha „megszabadultunk” a páros számok fölösleges vizsgálatától, akkor fölösleges páros számokkal osztani, hiszen páratlan számnak csak páratlan osztói lesznek!
- Ahhoz, hogy az algoritmusunk tökéletesen működjön, akkor is, ha ***n = 1***, a következőképpen járunk el:

```
Algoritmus Prím_4(n, prím):
   Ha n = 1 akkor
      prím ← hamis
   különben
      Ha n páros akkor prím ← n = 2
      különben
        prím ← igaz;     osztó ← 3
        ngy ← négyzetgyök(n)
        Amíg prím és (osztó ≤ ngy) végezd el:
          Ha n mod osztó = 0 akkor
             prím ← hamis
          különben
            osztó ← osztó + 2
          vége(ha)
        vége(amíg)
      vége(ha)
   vége(ha)
Vége(algoritmus)
```

```
// tudjuk, hogy a prímszámok 6 többszöröseinél eggyel
                                        kisebbek vagy nagyobbak
Algoritmus Prím_5(n, prím):
   Ha n = 1 akkor
      prím ← hamis
   különben
      Ha n páros akkor
        prím ← n = 2
      különben
        Ha n ≤ 5 akkor
           prím ← igaz
        különben
           Ha ((n - 1) mod 6 ≠ 0) és ((n + 1) mod 6 ≠ 0)
                akkor prím ← hamis
           különben
                osztó ← 3
         // tovább ugyanaz, mint az előző algoritmusban
```

15. Prímszámok generálása

Generáljuk egy adott számnál kisebb összes prímszámot!

Megoldás

Generáljuk a vizsgálandó számokat és ezekre alkalmazzuk az előbbi algoritmust:

```
Algoritmus Prímek_1(határ, k, prímek):
   prímek[1] ← 2
   n ← 3
   Amíg n < határ végezd el:
      prím ← igaz
      osztó ← 3
      ngy ← négyzetgyök(n)
      Amíg prím és osztó ≤ ngy végezd el:
         Ha n mod osztó = 0 akkor
            prím ← hamis
         különben osztó ← osztó + 2
          vége(ha)
       vége(amíg)
       Ha prím akkor k ← k + 1; prímek[k] ← n
      n ← n + 2
   vége(amíg)
Vége(algoritmus)
```

16. Eratoszthenész szitája

- *Eratoszthenész* felírt minden számot **2** és **3000** között egy papiruszra, megjegyezte a **2**-t, mint ***prímszámot***, majd ***kihúzott minden páros számot***.
- Most megjegyezte az első számot, amelyet még nem húzott ki (ez a **3**-as) és ***kihúzott minden 3-mal osztható számot***.
- Az algoritmusban a számokat tárolnunk kell, hogy megjegyezhessük, melyek azok, amelyeket kihúztunk, illetve melyek azok, amelyeket nem.

```
Algoritmus Prímek_3(határ, prím):
                                  // határ-nál kisebb számokat vizsgálunk
  Minden i = 2, határ végezd el:
    prím_i ← igaz                 // még nincs kihúzva egy szám sem
  vége(minden)
  Minden i = 2, sqrt(n) <= határ végezd el:
    Ha prím_i akkor                        // ha i még nincs kihúzva
      k ← i * i                           // az első kihúzandó szám
      Amíg k ≤ határ végezd el:
        prím_k ← hamis                          // kihúzzuk a k számot
        k ← k + i      // a következő kihúzandó többszöröse i-nek
      vége(amíg)
    vége(ha)
  vége(minden)
Vége(algoritmus)
```

Végül, a prím sorozat alapján kiírhatók (generálhatók) a prímszámok.

17. Mi a hatása a következő algoritmusnak?

Bemeneti adat: egy pozitív egész szám

Kimeneti adat: egy szám

1. Jelöljük a bemeneti számot ***N***-nel
2. Legyen ***X = 0*** és ***Z = 1***
3. Növeljük ***X*** értékét eggyel
4. Adjuk hozzá ***Z*** értékéhez ***X*** kétszeresét
5. Növeljük ***Z*** értékét eggyel
6. Ha ***Z*** nem nagyobb mint ***N***, akkor ismételjük meg az algoritmust a 3. lépéstől
7. Írjuk ki ***X*** értékét

floor(sqrt(N))

Általánosságok

- A számjegyekkel foglalkozó feladatok sok más feladat részfeladataiként jelentkeznek.
- Gyakran szükségünk van az „előfordulások”, illetve a „gyakoriság” tömbre.
- Ezeknek használata sok esetben vezet jobb hatékonyságú megoldásokhoz.
- Az oszthatóság vizsgálata szintén gyakran jelenik meg.

18. Hasonló számok

*Legyen **x** és **y** két természetes szám, amelyeknek legtöbb 9 számjegyük van. Döntsük el, hogy a két szám **hasonló-e**. Két számot hasonlónak nevezünk, ha számjegyeik halmaza azonos:* 2131 *és* 32211 *hasonló mivel számjegyeik halmaza ugyanaz:* {1,2,3}.

- a) Két tömb használatával
- b) Egy tömb segítségével
- c) Tömbhasználat nélkül

Hasonló számok

Elemzés

a) a feladat megoldása két tömb használatával rendkívül egyszerű:

- meghatározzuk ***x*** és ***y*** *előfordulási* tömbjét;
- összehasonlítjuk a két tömböt, és eldöntjük, hogy a két tömb azonos-e.

b) A feladat megoldása egy tömb használatával szintén egyszerű:

- meghatározzuk ***x*** *előfordulási* tömbjét;
- meghatározzuk, rendre az ***y*** szám számjegyeit, és vizsgáljuk, hogy ez a számjegy előfordult-e az ***x*** számban; ha ***y*** minden számjegyét megtaláljuk, következik, hogy ***x*** és ***y*** azonos. De ezt fordítva is megvizsgáljuk.

c) Ha nem szabad tömböt használni, meg kell vizsgálnunk, hogy ***x*** minden számjegye előfordul-e ***y*** -ban, és fordítva

```
Algoritmus előfordul(x, előf):                                  // a)
                    // létrehozzuk az x szám számjegyeinek előfordulástömbjét
    Mnden i = 0, 9 végezd el:
        előf[i] ← hamis
    vége(minden)
    Amíg x > 0 végezd el:
        előf[x mod 10] ← igaz
        x ← x /10
    vége(amíg)
Vége(algoritmus)

Algoritmus azonos_1(a, b): // eldöntjük, hogy a és b azonosak-e
    előfordul(a, előfA); előfordul(b, előfB)
    i ← 0
    Amíg i ≤ 9 és előfA[i] = előfB[i] végezd el:
        i ← i + 1
    vége(amíg)
    visszatérít i > 9
Vége(algoritmus)
```

```
Algoritmus ab(a, b):                                    // b)
// az a előfordulástömbjében vizsgáljuk a b számjegyeinek megfelelő elemeket
    előfordul(a, előfA)
    Amíg b > 0 és előfA[b mod 10] végezd el:
        b ← b /10
    vége(amíg)
    visszatérít b = 0
Vége(algoritmus)

Algoritmus azonos_2(x, y):
// x számjegyeit keressük y számjegyei között és fordítva
    visszatérít ab(x, y) és ab(y, x)
Vége(algoritmus)
```

```
Algoritmus AszámjegyeiBben(a, b):                    // c)
// vizsgáljuk, hogy a minden számjegye előfordul-e b-ben
    auxB ← b                                         // másolat b-ről
    Amíg a > 0 végezd el:
        aUtolsóSzj ← a mod 10
        Amíg auxB > 0 és aUtolsóSzj ≠ auxB mod 10 végezd el:
            auxB ← auxB / 10
        vége(amíg)
        Ha auxB = 0 akkor
                    // ha auxB 0 lett, az a aktuális számjegye nincs meg b-ben
            visszatérít hamis
        vége(ha)
        auxB ← b                                     // visszaállítjuk b értékét
        a ← a / 10                  // levágjuk az a szám utolsó számjegyét
    vége(amíg)
    visszatérít igaz
Vége(algoritmus)

Algoritmus azonos_3(x, y):
// vizsgáljuk, hogy x minden számjegye előfordul-e y-ban, és fordítva
    visszatérít AszámjegyeiBben(x, y) és AszámjegyeiBben(y, x)
Vége(algoritmus)
```

19. Legnagyobb palindrom

Adott az ***n*** (0 < ***n*** < 2^{31}) *természetes szám. Határozzuk meg azt a* ***maxPal*** *számot, amely a legnagyobb palindrom, amelyet az* ***n*** *szám minden számjegyének átrendezése által kaphatunk. Ha nem lehet kialakítani a palindromot, amelyben az* ***n*** *szám minden számjegye szerepeljen, akkor* ***maxPal*** *értéke* -1.

1. *Példa:* ha ***n*** = 21523531, akkor ***maxPal*** = 53211235.

2. *Példa:* ha ***n*** = 12272351, akkor ***maxPal*** = -1.

Legnagyobb palindrom

Elemzés

- Létrehozzuk az ***n*** számjegyeinek gyakoriságtömbjét, amelynek feldolgozása lehetővé teszi, hogy a kért számot hatékonyan generálhassuk
- Ha a számban csak páros gyakoriságok léteznek, elhelyezünk az új számba **gyak[i] / 2** darab ***i*** számjegyet, értékeik csökkenő sorrendjében, majd elhelyezünk az új számba **gyak[i] / 2** darab ***i*** számjegyet, értékeik növekvő sorrendjében
- Ha az ***i*** számjegy gyakorisága páratlan, és a szám közepének megfelelő hely még nem foglalt, ide teszünk egy ***i*** számjegyet
- Ha megjelenik még egy számjegy, amelynek gyakorisága páratlan, nem lehetséges palindromszámot generálni

```
Algoritmus palindrom(n):
    Minden i = 0, 9 végezd el:
        gyak[i] ← 0
    vége(minden)
    Amíg n > 0 végezd el:
        gyak[n mod 10] ← gyak[n mod 10] + 1
        n ← n / 10
    vége(amíg)
    szám ← 0                    // a legnagyobb palindromszám
    közép ← -1                  // a közepére még nem került semmi
    i ← 9      // a számjegyeket csökkenő sorrendben dolgozzuk fel
    …
```

```
Amíg i ≥ 0 végezd el:
    Ha gyak[i] mod 2 = 1 akkor
                        // az i számjegy páratlan számszor fordul elő
        Ha közép = -1 akkor    //ha a közepére még nincs téve semmi
            közép ← i                  // az i szjegy lesz a szám közepén
            gyak[i] ← gyak[i] – 1  // elhasználtunk egy i számjegyet
            i ← i + 1
         különben
            visszatérít -1          // ha a közép indexű hely foglalt, nem
                                    // tudunk palindromszámot generálni
        vége(ha)
    különben
        hány ← gyak[i] / 2
                // az i számjegyből hány darabot építünk a számba
        ...
```

```
            Minden j = 0, hány – 1 végezd el:
                szám ← szám * 10 + i
            vége(minden)
            gyak[i] ← gyak[i] – hány      // hány darab i-t felhasználtunk
        vége(ha)
        i ← i - 1                         // a következő (kisebb) számjegyérték
    vége(amíg)
    Ha közép ≠ -1 akkor                   // ha a közép kapott értéket
        szám ← szám * 10 + közép          // beépítjük a számba
    vége(ha)
    Minden i = 0, 9 végezd el:
            // most növekvő sorrendben dolgozzuk fel a számjegyeket
        Minden j = 0, gyak[i] - 1 végezd el:
            // minden számjegyet beépítünk annyiszor, ahány maradt
            szám ← szám * 10 + i
    visszatérít szám
Vége(algoritmus)
```