Elemi algoritmusok

Ionescu Klára

clara@cs.ubbcluj.ro

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 megoldást alprogram (al-algoritmus) formájában adunk meg
- Egy alprogram = egy funkció
- Egy alprogram nem olvas be, nem ír ki, csak ha pontosan ez a funkciója
- Minden alprogramnak pontosan annyi paramétere lesz, ahányra szüksége van (minden, amit kintről kap, és minden, amit kifele továbbítania kell)

1. Feladat: 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!

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

Elemzés: 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)
```

2.a. Két természetes szám legnagyobb közös osztója

Határozzuk meg két természetes szám legnagyobb közös osztóját!

Megoldás

Legyen $a, b \in \mathbf{N}^*$ és a következő jelölések:

(a, b) : a és b legnagyobb közös osztója;

```
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: „térítsd a” }
Vége(algoritmus)
```

2.b. 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:* „térítsd 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. Másodfokú egyenlet

Írjunk algoritmust az $ax^2 + bx + c$ *valós együtthatójú másodfokú egyenlet megoldására! Tárgyaljunk minden lehetséges esetet!*

Elemzés

- Mivel az együtthatókra vonatkozóan a feladat szövegében nincs semmilyen információ, feltételezhetjük, hogy ezek tetszőleges értékekkel rendelkezhetnek.
- A figyelmünk a 0-val egyenlő, illetve a 0-tól különböző értékekre irányul.
- Így a 8 (alap)-eset:

1. $a = 0, b = 0, c = 0$, 2. $a = 0, b = 0, c \neq 0$, …, 8. $a \neq 0, b \neq 0, c \neq 0$.

  - De ezeken belül újabb eseteink lesznek, annak függvényében, hogy pozitív vagy negatív számból kell négyzetgyököt vonnunk.

5. 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)
```

6. 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, ezzel párhuzamosan felépítjük az új számot.
- Az új szám generálását ugyancsak a ***Horner-séma*** néven ismert módszer segítségével végezzük: ***újszám ← újszám · 10 + szj***.

6. 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!!! }

7. Számgenerálás

Olvassunk be ismeretlen számú karaktert (sorvége-jelig). Építsünk fel ezekből – ha lehetséges – egy egész számot.

Elemzés

Észreveszünk két lényeges különbséget az előző feladat és e között:

- most karaktereket olvasunk be, amelyekről nem állíthatjuk, hogy biztos, hogy számjegyek.
- most a számjegyek száma ismeretlen.

- Mivel nem biztos, hogy minden karakter számjegy, ezt ellenőrizni fogjuk.

7. Számgenerálás

- Ha „idegen” karaktert találunk, kilépünk a feldolgozásból, és kiírjuk a megfelelő üzenetet.
- Fogunk vigyázni arra is, hogy ne próbáljunk olyan túl nagy számot generálni, amelyet nem tudunk ábrázolni (vigyázunk a *túlcsordulás*ra).
- Minden típuson belül létezik egy legnagyobb szám, amit a számítógép még ábrázolni tud (*legnagyobb*).
- Nem kérdezhetjük meg, hogy **Ha** szám > legnagyobb **akkor** … mivel nem létezik a *legnagyobb*nál nagyobb szám.
- Feltesszük a kérdést egy lépéssel „hamarabb” mint, hogy az aktuális számjegyet is feldolgoznánk:

Ha szám > (legnagyobb - számjegy) **div** 10 **akkor**…

```
Algoritmus Számgenerálás:
  szám ← 0
  idegen ← hamis                    { még nem olvastunk be idegen karaktert }
  túlnagy ← hamis                   { még nem generáltunk túl nagy számot }
  Amíg nem sorvége köv. és nem idegen és nem túlnagy végezd el:
    Be: kar
    Ha kar < '0' vagy kar > '9' akkor
      idegen ← igaz                 { kar a [0..9] intervallumon kívül esik }
    különben
      szj ← a kar karakter számértéke
      Ha szám > (legnagyobb - szj) div 10 akkor
                                    { ha beépítenénk a szj számjegyet }
        túlnagy ← igaz              { a számba, a számot már nem lehetne ábrázolni }
      különben
        szám ← szám*10 + szj        { beépítjük a szj számjegyet a számba }
      vége(ha)
    vége(ha)
  vége(amíg)
```

```
Ha nem idegen és nem túlnagy akkor
    Ki: szám                          { sikerült egy helyes számot generálni }
 különben
    Ha idegen akkor                   { a beolvasott karakter nem volt számjegy }
     Ki: 'Idegen karakter!'
    különben                          { túl nagy számot kellett volna generálnunk }
     Ki: 'Túl nagy szám!'
    vége(ha)
 vége(ha)
Vége(algoritmus)
```

8. Törzstényezők

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

Elemzés

- A számot ismételten osztjuk, előbb 2-vel, majd 3-mal stb. ameddig 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!
- 2 2 3 -> 1p
- 2^2 * 3 -> 2p

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, ' '
            Ha n ≠ 1 akkor Ki: '* '
         vége(ha)
      vége(ha)
      osztó → osztó + 1
   ameddig n = 1
Vége(algoritmus)
```

9. 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.

```
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)      { elkerülhető ngy kiszámítása?? }
    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)
```

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

10. Fibonacci-számok

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

Elemzés

- ***Bemeneti adatok: n*** természetes szám
- ***Kimeneti adatok: n* darab *Fibonacci-szám***
- nincs szükség adatszerkezetre (a feladat nem kéri ezek megőrzését sorozat formájában)
- a feladat követelményeinek szétválasztása:
  - beolvasunk egy természetes számot,
  - kiszámítunk és kiírunk sorban egy-egy Fibonacci-számot

Megoldási módszer

- A *Fibonacci-számok*at az alábbi rekurzív összefüggéssel definiáljuk:

  $F_0 = 0, F_1 = 1, F_i = F_{i-1} + F_{i-2}$, ha $i \geq 2$.
- Minden *Fibonacci-szám* tehát a megelőző kettőnek az összege.
- A *Fibonacci-számok* kapcsolatban vannak az ***aranymetszés*** arányával.
- A Fibonacci-számok exponenciálisan nőnek.
- 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
    Ki: c
    k ← k + 1
  Vége(amíg)
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
      Ki: a
    különben
      Ha n = 2 akkor
        Ki: a, b
      különben
        c ← a + b
        Ki: a, b, c
        k ← 3
        Amíg k < n végezd el:
           a ← b
           b ← c
           c ← a + b
           Ki: c
           k ← k + 1
        vége(amíg)
     vége(ha)
   vége(ha)
Vége(algoritmus)
```

11. Fibonacci-szám-e?

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

Elemzés

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

- 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):                    { az n számot vizsgáljuk }
   a ← 0; b ← 1; c ← a + b
   Amíg c < n végezd el:
      a ← b
      b ← c
      c ← a + b                          { b a legnagyobb Fibonacci szám, amely < n }
   vége(amíg)
   Ha c = n akkor Ki: 'Fibonacci szám'
               különben Ki: n,'=',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)
      Ki: '+',b; n ← n - b
   vége(amíg)
  vége(ha)
Vége(algoritmus)
```

12. Mi a hatása?

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

Generálja és kiírja az első n Fibonacci számot.

Programok\5_Mit_generál.cpp

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 Gyorshatvány), 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 algoritmust találni, hogy változna a másik bonyolultság, vagy szerencsés esetben az is csökken***.

13. 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)
```

14. 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(a, b, p):

{ *Kiszámítjuk* a *és* b *szorzatát, „szorzás” nélkül* }

{ *Bemeneti adatok*: a, b }

{ *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)

15. 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, prím):                    { 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)
   prím ← osztók_száma = 2
                                  { értéket adtunk a prím változónak }
                              { C++-ban: return osztók_száma == 2 }
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(határ):
   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 }
```

További optimalizálás

- Továbbá, ismeretes, hogy a négyzetgyököt számoló függvény ismeretlen lépésszámban határozza meg az eredményt, amely valós szám. Ezt elkerülendő, lemondunk a négyzetgyök kiszámításáról és az Amíg feltételét a következőképpen írjuk:

Amíg prím **és** (osztó * osztó ≤ n) **végezd el**:

- Így, nem dolgozunk valós számokkal és nem számítjuk ki fölöslegesen a négyzetgyököt.

16. Prímszámok generálása

Generáljuk egy adott számnál kisebb összes prímszámot!

Megoldás

Generáljuk a vizsgálandó számokat és ezekre alkalmazzuk az előbbi algoritmust:

```
Algoritmus Prímek_1(határ):
   Ki: 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 Ki: n
      n ← n + 2
   vége(amíg)
Vége(algoritmus)
```

17. 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, i * i <= 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.

18. 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))