Az algoritmus felépítése: Először elkészítjük az adatbekérő részt (2–3. sor). Hátultesztelő ciklusba ágyazzuk (1–5. sor). Végül megadjuk a ciklusban maradás feltételét (4. sor). Addig kell ismételni az adatbekérést, míg a bekért szám egynél kisebb vagy ötnél nagyobb.

```
1) Ciklus
2)   Ki: „Kérek egy osztályzatot!"
3)   Be: A
4) amíg (A < 1) v (A > 5)
5) Ciklus vége
6) ...
```

ciklusmag

bentmaradás feltétele

Összefoglalva: a hátultesztelő ciklus egy olyan feltételes ciklus, ahol a feltételt a ciklus végén adjuk meg. Ennek megfelelően akkor használjuk, ha **nem tudjuk előre megadni az ismétlések számát**, de **egyszer mindenképpen végre kell hajtani a ciklusmagot**.

A feltétel megadására az elágazásoknál megismert szabályok érvényesek, azaz csak igaz/hamis értéket adó kifejezés lehet. Összetett feltételeket úgy adhatunk meg, hogy a részfeltételeket logikai műveletekkel kapcsoljuk össze.

A hátultesztelő ciklust gyakran alkalmazzuk ellenőrzött adatbevitelre, illetve a program újrakezdésére. Utóbbira példa a dobókockával történő dobás szimulálása többször egymás után bármely billentyű (kivétel a szóköz) megnyomásával. A programnak akkor van vége, mikor a felhasználó szóköz billentyűt nyom.

A Python nem használ hátultesztelő ciklust. A kódot úgy kell alakítani, hogy elöltesztelő ciklust használjon. Ehhez: 1. A ciklusmagban található utasításokat a ciklus elé is be kell írni. 2. A feltételt a ciklus elejére kell helyezni.

43. mintafeladat – hátultesztelő ciklusból elöltesztelő

Addig kérjünk be a felhasználótól egy egész számot, amíg az meg nem felel egy osztályzatnak, tehát 1…5 egész számnak.

1. Kérjük be a felhasználótól az egész számot az a változóba (1. sor).
2. Vizsgáljuk meg egy feltétellel, hogy a szám rossz-e. Rossz a szám, ha kisebb, mint 1 vagy nagyobb, mint 5: a < 1 or a > 5 (2. sor).
4. A feltételhez kötött ismétlés (ciklus) utasítása a while (2. sor).
5. A while-t tartalmazó sor végére :-ot rakunk, hiszen ez egy vezérlési szerkezet (2. sor).
5. Rossz szám esetén ismételni kell az adatbekérést. A ciklus magjában az 1. sort megismételjük (3. sor).
6. Végül a helyesen megadott számot kiírjuk (4. sor). Nyilván egy komolyabb programnak ez lenne a kezdete.

```
1 a = int(input("Kérek egy osztályzatot: "))
2 while a < 1 or a > 5:
3     a = int(input("Kérek egy osztályzatot: "))
4 print("Az ellenőrzött osztályzat: ",a)
```


Végtelen ciklusok

Megadhatunk olyan feltételt, hogy a ciklusból nem lép ki a program. (A mellékelt példában a 3 mindig kisebb, mint a 4.) Addig ismétli a ciklusmagot, míg a programot meg nem állítjuk külső beavatkozással. A szaknyelv erre mondja, hogy végtelen ciklusba került a program.

```
1 while 3 < 4:
2     print ("végtelen ciklus")
```

A példában egyszerűen átlátható, hogy a feltétel végtelen ciklust okoz. Egy összetett programban azonban ez nem szokott ennyire egyértelmű lenni.

Ez különösen zavaró, ha nem szándékosan tesszük, hanem valami hiba következménye. Ilyenkor az ablakban nem történik semmi változás, hiába próbáljuk használni a billentyűzetet vagy egeret az érintett ablakon, nem reagál. A program látszólag lefagyott. Valójában valamit tesz, csak annak esetleg nincs semmilyen jele számunkra.

Fejlesztés közben a lefagyott programot megállíthatjuk a parancsértelmező ablakban a **Shell/Restart Shell** menüvel, vagy a **Ctrl+F6** billentyűkombinációval. Végszükség esetén a Feladatkezelőben állíthatjuk le a teljes parancsértelmezőt.

44. mintafeladat – végtelen ciklus megállítása

Írjunk egy végtelen ciklust tartalmazó rövid programot. Futtassuk, majd állítsuk meg.

1. Gépeljük be a Végtelen ciklusok fejezetben olvasható kétsoros programot.
2. Futtassuk (**F5**).
3. Menjünk át a parancsértelmező ablakba (**Python 3.8.0 Shell**).
4. Válasszuk a **Shell/Restart Shell** menüpontot, vagy nyomjuk meg a **Ctrl+F6** billentyűket.
5. Megáll a program futása, és megjelenik a prompt jel >>>. Ez mutatja, hogy az értelmező várja a következő parancsot.

Feladatok

1. Bővítsük a legkisebb többszöröst kiszámító programot úgy, hogy előzőleg ellenőrizzük a felhasználó által megadott számokat, és csak akkor engedjük tovább a programot, ha 0-nál nagyobb számot adott meg.
2. Határozzuk meg két pozitív egész szám legnagyobb közös osztóját az alábbi algoritmussal. Ha a két szám egyenlő, akkor megvan a legnagyobb közös osztó. Ha nem egyenlők, a nagyobból kivonjuk a kisebbet, a kisebbet változatlanul leírjuk. (Lásd mellékelt ábra.)
3. Egészítsük ki a fenti programot úgy, hogy a számítást ismételten, több számpárral is legyen módunk elvégezni. A felhasználó a kilépési szándékát úgy jelezze, hogy valamelyik szám 0 vagy negatív. Egyéb ellenőrzés nem kell.
4. Írjunk programot, ami eldönti egy bekért pozitív egész számról, hogy prím-e. (Eratoszthenész szitája)
5. Írjunk programot, ami elvégzi egy pozitív egész szám prímtényezős felbontását.

2. feladat	
70	15
55	15
40	15
25	15
10	15
10	5
5	5

11. ALPROGRAMOK

Az **eljárások** és **függvények** alprogramok (más néven részprogramok, szubrutinok).

Az eljárások és függvények segítségével programjainkat részekre bonthatjuk.

Az alprogramot megírjuk, majd névvel látjuk el. A főprogram különböző helyén már csak a nevével hivatkozunk rá, és végrehajtódik.

Az eljárások és függvények használatának egyik előnye a **kódismétlés elkerülése**. A programunk rövidebb, áttekinthetőbb lesz, és elég az alprogramban változtatni, ha a kód hibás, vagy másképpen szeretnénk működtetni a továbbiakban, tehát **egységesen módosítható**. További előny, hogy egy nagyobb program így részekre bontva, több emberrel, **csoportmunkában megoldható**.

Az eljárások és függvények használata megváltoztatja az utasítások alapvetően sorrendi végrehajtását. Az alprogramokon és a főprogramon belül megmarad a sorrendi végrehajtás.[36] A meghíváskor a program futása az alprogram első sorával folytatódik. Az alprogram befejeztével a főprogramban a meghívás utáni sorral folytatódik a végrehajtás.

Tanulmányozzuk a mellékelt ábrát. Legyen egy főprogramunk, ami sok feladatot old meg. A képernyőre kiírásnál az áttekinthetőség kedvéért szeretnénk a feladatokat vonalakkal (és egy üres sorral) elválasztani. Megírjuk tehát a jobb oldalon látható *elvalaszto* nevű eljárást, amit minden feladat végén meghívunk a főprogramból. Gondoljuk át, mennyivel hosszabb lenne a főprogram, ha az alprogramot a meghívás minden egyes helyére behelyettesítenénk. Ha úgy döntünk, hogy dupla vonalsort akarunk alkalmazni, akkor eljárást használva egy helyen kell hozzányúlni a programhoz. Ha nem alkalmaznánk, akkor minimum három helyen. Emellett gyorsabban célt érünk, ha felkérünk valakit az alprogram megírására, hiszen akkor ketten dolgozunk a feladaton. A főprogram és az eljárás megírható anélkül, hogy részletekbe menően tisztában kellene lenni egymás feladatával.

36 A program összes utasításának helyzetét egyszerre nézve változik meg a végrehajtás sorrendje.

12. ELJÁRÁSOK

Az eljárás egy alprogram. Hatását azon keresztül fejti ki, hogy valamilyen tevékenységet hajt végre. Például megjelenít valamit a képernyőn, vagy megváltoztat egy adatot egy fájlban vagy adatbázisban.

```
Eljárás alprogram2
...
Eljárás vége

Eljárás alprogram1
        alprogram2
        ...
Eljárás vége

Főprogram (main)
        alprogram1
```

Eljárás paraméter nélkül

Az eljárást a meghívás helye előtt kell létrehozni. Tehát, ha a főprogramban használni akarunk egy eljárást, azt a megelőző sorokban kell megírnunk. Ha az alprogram1 nevű eljárásban használni szeretnénk az alprogram2-t, akkor az alprogram2-t az alprogram1 előtt kell létrehoznunk. (Lásd mellékelt ábra.)

Az eljárás létrehozását a **def** kulcsszóval kezdjük. Ezt követi az eljárás neve *(border)*, majd egy üres zárójelpár, ha nincsenek átadandó paraméterek. Végül kettősponttal zárul a sor (1. sor). Az eljárás tartalmi részét egy szinttel beljebb kezdve adjuk meg (2–5. sor). Mint minden Python vezérlési szerkezet esetében, az eljárás végének sincs külön jele. A tagolás mutatja a végét.

```
1  def border():
2      print()
3      for i in range(80):
4          print("-", sep='', end='')
5      print('\n')
6
7  print("1. feladat")
8  border()
9  print("3. feladat")
10 border()
11 print("4. feladat")
12 border()
13 print("6. feladat")
```

Az eljárást a főprogramból a nevével és az üres zárójelpárral hívjuk meg (8., 10., 12. sorok).

Eljárás paraméterátadással

Az eljárások működését paraméterek segítségével befolyásolhatjuk. Az előző példa folytatásaként megadhatjuk, milyen legyen az elválasztójel, és milyen szélességű legyen. A paramétereket a név mögötti zárójelek között adjuk meg (7., 10., 12. sorok).

Az eljárás létrehozásakor a zárójelek között fel kell sorolni a paraméterként használt változók nevét (1. sor – n, s). Ezekbe fognak (alapesetben) behelyettesítődni a főprogram meghívásánál szintén a zárójelek között megadott értékek, mégpedig sorrendben. Az ábrán a 80 (7. és 12. sor), illetve 10 (10. sor) az n változóba, a * (7. és 12. sor), illetve – karakterek (10. sor) az s-be.

```
1  def border(n,s):
2      print()
3      for i in range(n):
4          print(s, sep='', end='')
5      print('\n')
6
7  border(80,'*')
8  print("1. feladat")
9  print("a) feladat")
10 border(10,"-")
11 print("b) feladat")
12 border(80,'*')
13 print("2. feladat")
```

Változók hatóköre

A Pythonban a modulban létrehozott változó csak a modulban használható fel, ha erről másképpen nem rendelkezünk. Hiába hivatkozunk rá a főprogramban, vagy egy másik modulban, hibaüzenetet kapunk. A mellékelt ábrán az *a* változót az eljáráson belül hozzuk létre (3. sor), csak ott használhatjuk fel. Ha a főprogramból megpróbáljuk kiíratni az értékét (11. sor), hibaüzenetet kapunk, mert a változó ott nem létezik.

```
1  def proc():
2      #global a
3      a = 5
4      print(b)
5      c = 3
6      print(c)
7
8  b = 0
9  c = 4
10 proc()
11 print(a)
```

A modulokban létrehozott változókat lokálisnak nevezik. Érvényességi körük (hatókörük) az a modul, ahol létrejöttek.

A főprogramban létrehozott változók viszont alapvetően a teljes programban elérhetők. Az ábrán létrehozzuk a *b* változót 0 értékkel (8. sor), amit aztán kiíratha-tunk az eljárásból (4. sor), mivel ott is elérhető.

Ha létezik ugyanolyan nevű lokális és globális változó, a lokális változó érhető el. Az ábrán ilyen a c változó, amit létrehozunk 4 értékkel a főprogramban (9. sor), és 3 értékkel az eljárásban (5. sor). Az eljáráson belüli kiíratás (11. sor) a 3-at, azaz a lokális változót fogja megjeleníteni. Ha az 5. sort kivesszük az eljárásból, például egy #-ot írunk elé, a c értékeként 4 fog megjelenni a kiíratáskor.

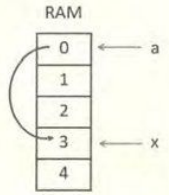
A főprogramban létrehozott változókat globálisnak nevezik. Érvényességi körük a teljes program, ha nincs egy azonos nevű lokális változó.

A Python lehetőséget biztosít rá, hogy létrehozzunk globális változót a modulon belül. Ilyenkor a változó elé a **global** kulcsszót kell írni. Ha az ábrán kivesszük a 2. sorban a #-ot, már nem kapunk hibaüzenetet, és kiírható a főprogramból az *a* változó értéke.

Érték szerinti paraméterátadás

Az előzőekben megismert paraméterátadást érték szerinti átadásnak nevezik, mert a főprogramból csak a változó értéke kerül át az alprogramba.

```
1 def proc(x):
2     x += 1
3     print(x)
4
5 proc(3)
6 a = 10
7 proc(a)
8 print(a)
```



A program a futásakor lefoglal egy memóriarészt a főprogramban található változónak (az *a* változónak a 0. memóriarészt). Aztán a *proc* eljárás a meghívásakor lefoglal egy másik memóriarészt (az *x* lokális változónak a 3. memóriarészt). Az eljárás meghívása során a 0. memóriarészből áttöltődik a változó tartalma a 3.-ba. Így a két változó innentől kezdve teljesen független életet él. Az *x*-et módosítva az *a* nem változik meg.

Az eljárás befejeztével az eljáráshoz tartozó változó *(x)* megszűnik.

Az eljárást meghívhatjuk úgy is, hogy nem változó, hanem konkrét érték van megadva paraméterként (például az 5. sorban: *proc(3)*). Ebben az esetben az alprogram változójának lefoglalt memóriaterületre a konkrét számérték kerül (a példában 3).

Kövessük végig az ábrán látható példát.

1. Az 5. sorban indul a végrehajtás, mert ez a főprogram első sora. Meghívjuk a *proc* eljárást 3-as paraméterrel.
2. A program végrehajtása az 1. sortól folytatódik: A 3 az x változóba kerül. Az x értéke eggyel növekszik a 2. sorban, tehát 4 lesz. Ezt is fogja kiírni az eljárás a 3. sorból: *4*.
3. Mivel az eljárás végére értünk, a program a főprogram következő sorától (6.) folytatódik.
4. Az *a* változónak a 10 értéket adjuk (6. sor), majd meghívjuk ismét a proc eljárást (7. sor), most az *a* változót használva paraméterül.
5. A program végrehajtása ismét az 1. sortól folytatódik, ahol az *a* változóból átmásolódik a 10 az *x* változóba.
6. A 2. sorban az x értéke eggyel növekszik, így 11 lesz, amit meg is jelenítünk a képernyőn (3. sor).
7. A program végrehajtása a főprogram következő sorától (8.) folytatódik, ahol kiírjuk az a változó értékét. Mivel az a változó értéke nem változott, így 10 lesz a kiírt szám.

45. mintafeladat – eljárás készítése

A matematikában tökéletes számoknak nevezik azokat a pozitív egész számokat, amiknek a számnál kisebb pozitív osztóinak összege magát az egész számot adja. Például a 28 tökéletes szám, mert 1+2+4+7+14 = 28. Írjunk eljárást, ami paraméterül kap egy egész számot, majd arról eldönti, hogy tökéletes szám-e vagy sem, és ezt megjeleníti a képernyőn. Ha kész az eljárás, írjunk programot az eljárást felhasználva, ami a 2...1000 számok mindegyikéről kiírja, tökéletes szám-e, majd bekér a felhasználótól egy egész számot, és kiírja, tökéletes szám-e.

```
1  def pn(num):
2      end = int(num/2)
3      s = 1
4      for i in range(2,end+1):
5          if num % i == 0:
6              s += i
7      if s == num:
8          print("Tökéletes szám")
9      else:
10         print("Nem tökéletes szám")
11
12 for i in range(2,1001):
13     print(i, end=' ')
14     pn(i)
15 ob = int(input("Kérem a vizsgálandó számot: "))
16 pn(ob)
```

1. Az eljárás létrehozásához a **def** utasítást használjuk. Mögötte megadjuk az eljárás nevét *(pn)*, és zárójelek között a paraméter nevét *(num)*, majd beírjuk a vezérlési szerkezetek végén kötelező kettőspontot. A paraméterben fog átadódni a vizsgált szám (1. sor).
2. Az eljárás "lelke" a 4–6. sorok, ahol egy ciklussal végigvizsgáljuk, hogy 2-től kezdve (4. sor – **range** első paramétere) mik a szám osztói. Azért 2-től kezdjük, mert az 1 biztosan osztója mindegyik számnak, így azt nem kell vizsgálni, elég hozzáadni. Ezért indul az osztók összege *(s)* 1-ről (3. sor).
3. Meghatározzuk, meddig kell a ciklusban számlálni. A vizsgált szám feléig elegendő elmenni, hiszen annál nagyobb valódi osztója nem lesz.[37] Az **int** függvény az eredmény egészrészét veszi, amire akkor van szükség, ha a vizsgált szám páratlan, mivel az kettővel osztva törtet adna eredményül, amit nem adhatunk meg a **range**-ben (2. sor).
4. A számláló ciklus végigpörgeti az *i* változóban a lehetséges osztókat (4. sor).
5. Ha a vizsgált szám *(num)* osztható az épp aktuális lehetséges osztóval *(i)*, mert az osztási maradék 0 (5. sor), az eddigi osztók összegét megnöveli az aktuális osztóval (6. sor).

37 Igazából elég a vizsgált szám négyzetgyökéig próbálkozni, de ennek belátásától most eltekintünk.

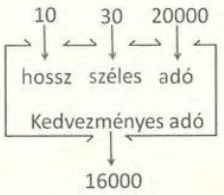
6. Ha az osztók összege *(s)* megegyezik a vizsgált számmal *(num)* (7. sor), akkor kiíratjuk, hogy "Tökéletes szám" (8. sor).
7. Különben (9. sor) kiíratjuk, hogy "Nem tökéletes szám" (10. sor).
8. A főprogramban indítunk egy ciklust, ami a 2-től 1000-ig számlál (12. sor).
9. A ciklus minden egyes lefutásakor kiíratjuk az aktuális számot (13. sor), majd meghívjuk a *pn* eljárást, ami eldönti és kiírja, tökéletes számról van-e szó (14. sor).
10. Bekérjük a vizsgálandó számot a felhasználótól az *ob* változóba (15. sor).
11. Meghívjuk a *pn* eljárást a vizsgálandó számra, hogy megvizsgálja és kiírja, tökéletes számról van-e szó.

Feladatok

1. Készítsünk eljárást, ami paraméterül kap három egész számot, ami megfelel egy óra, perc és másodperc értéknek, majd kijelzi az időt 21:03:12 (egy másik példa: 4:11:07) formátumban. A kipróbálásához írjunk olyan főprogramot, ami bekéri a felhasználótól a három adatot, és meghívja az eljárást.
2. Készítsünk eljárást, ami paraméterül egy számot kap, ami egy másodpercben mért idő. Ezt kell átváltani óra, perc, másodpercre, majd megjelenítenie 21:03:12 (egy másik példa: 4:11:07) formátumban. Használjuk fel az előző pontban megírt eljárást. A kipróbálásához írjunk olyan főprogramot, ami bekéri a felhasználótól a másodperc adatot, és meghívja az eljárást.
3. A matematikában barátságos számoknak nevezzük azokat a pozitív egész számpárokat, amelyekre teljesül, hogy az egyik szám felírható a másik szám (saját magánál kisebb) osztóinak összegeként, és fordítva. Például a 220 és a 284 barátságos számpár, mert 220 osztói: 1 + 2 + 4 + 5 + 10 + 11 + 20 + 22 + 44 + 55 + 110 = 284, illetve 284 osztói: 1 + 2 + 4 + 71 + 142 = 220. Írjunk eljárást, ami a paraméteréül kapott két számról eldönti, hogy barátságos számpárt képeznek-e, és ezt ki is írja a képernyőre. Teszteléshez további ilyen számpárok: (1184; 1210), (2620;2974), (5020;5564).
4. Készítsünk eljárást, ami kiír a képernyőre egy véletlenszerű számjegyvariációt egy "kombinációs zárhoz" (pl. 34512, vagy 043). A számjegyek számát a paraméterben lehessen meghatározni. A kipróbálásához írjunk olyan főprogramot, ami bekéri a felhasználótól a számjegyvariáció hosszát, és meghívja az eljárást.

13. FÜGGVÉNYEK

A függvények olyan eljárások, amik egyetlen értéket *visszaadnak* a főprogramnak.



Úgy szokták megfogalmazni, hogy a függvénynek visszatérési értéke van. Természetesen a függvény is elvégezheti mindazokat a feladatokat, amiket az eljárás. Azonban különlegességét a visszatérési érték adja.

Szokás a függvényeket mint átalakító gépeket szemléltetni, hasonlóan az alsó tagozatos "mit csinál a gép" típusú feladatokhoz.

A gép neve, ami szerencsés esetben utal a feladatára (pl. húsdaráló), lehet a függvény neve. Amit a gépbe rakunk, az a bemenő paraméter (pl. hús). Ami kijön a gépből (darált hús), az eredmény, a visszatérési érték.

```
1) Függvény kedv(h:egész, sz:egész, a:valós) : valós
2)     Ha h<=25 vagy sz<=15 akkor a:=0,8*a
3)     kedv:=a
4) Függvény vége
5) Főprogram
6)     a:=kedv(10,30,20000)
7)     Ki: a
8)     Ki: kedv(20,30,30000)
9)
10)    kedvez(20,25,10000)
```

Visszatérési érték típusa

Mondatszerű leírásnál[38] a függvényen belül a függvény nevével (mindenféle paraméter nélkül) jelöljük meg a visszatérési értéket (3. sor).

A visszatérési érték típusát mondatszerű leírásban a paraméterlista után írjuk :-tal elválasztva (1. sor).

A főprogramban a visszatérési értékkel valamit kezdeni kell: kiíratni (8. sor), egy változóba rakni (6. sor), egy másik függvénnyel vagy eljárással felhasználni, stb. Értelmetlen, és hibaüzenetet okoz önmagában a függvényt használni utasításként (10. sor), hiszen a visszatérési érték létrejön, azzal valaminek történni kell. Olyan lenne, mintha a (húsdaráló) gépünk kimeneti nyílását behegesztenénk.

A függvények kódolása alig tér el az eljárásokétól. A visszatérési értéket a **return** kulcsszó mögé kell írni.

A függvény csak egyetlen értéket adhat vissza.

38 És pár programnyelvben is, mint például a Pascal, de a Pythonban nem!

46. mintafeladat – függvény készítése

Az önkormányzat a 15 m vagy annál keskenyebb, illetve 25 m vagy annál rövidebb telkekre 20% kedvezményt ad az adóból. Írjunk függvényt, ami megkapja a telek szélességét, hosszúságát, és a kedvezmény nélküli adót, visszatérési értékként azt az adót adja, amiben már figyelembe vettük a kedvezményt is, ha volt. Ha nem jár kedvezmény, akkor az eredeti adót adja vissza.

```
1 def red(t, w, l):
2     if l<=25 or w<=15:
3         t = 0.8 * t
4     return t
5
6 tax = int(input("Kérem a kedvezmény nélküli adót: "))
7 length = int(input("Kérem a telek hosszát: "))
8 width = int(input("Kérem a telek szélességét: "))
9 print("A kedvezményes adó: ", red(tax,width,length))
```

1. Az eljárás létrehozásához a **def** utasítást használjuk. Mögötte megadjuk a függvény nevét (*red*), és zárójelek között a paraméterek nevét (*t* – eredeti adó, *w* – szélesség, *l* – hosszúság), majd beírjuk a vezérlési szerkezetek végén kötelező kettőspontot (1. sor).
2. Ha a hosszúság *(l)* kisebb vagy egyenlő 25, vagy (**or**) a szélesség kisebb vagy egyenlő 15 (2. sor), vesszük az eredeti adó *(t)* 80%-át (3. sor). (Ha az eredeti adó 100%, és abból 20% a kedvezmény, akkor marad 80%, amit be kell fizetni.)
3. A **return** kulcsszó mögött megadjuk a visszatérési értéket, ami a kedvezményes adó *(t)*.
4. A kipróbáláshoz bekérjük a főprogramban (6–9. sor) a szükséges adatokat (6–8. sor).
5. A **print** utasításnak arra a helyére, ahová a kiírandó változót szoktuk írni, most a függvény meghívása kerül (red...). Miután a függvény lefutott, a visszatérési értéke, azaz a kedvezményes adó a függvény helyére kerül visszsza, így a print utasítás azt fogja kiírni (9. sor).

Feladatok

1. Készítsen függvényt *hetnapja* néven, amely a paraméterként megadott dátumhoz (hónap, nap) megadja, hogy az a hét melyik napjára esik (hétfő, kedd...). Tudjuk, hogy az adott év nem volt szökőév, továbbá azt is, hogy január elseje hétfőre esett. Használja az alábbi algoritmust, ahol a tömbök indexelése 0-val kezdődik.

```
Függvény hetnapja(honap:egesz, nap:egesz): szöveg
    napnev[ ]:= ("vasarnap", "hetfo", "kedd", "szerda", "csutortok", "pentek", "szombat")
    napszam[ ]:= (0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 335)
    napsorszam:= (napszam[honap-1]+nap) MOD 7
    hetnapja:= napnev[napsorszam]
Függvény vége
```

2. N természetes szám faktoriálisa alatt 1-től N-ig a természetes számok szorzatát értjük. Írjunk függvényt, ami paraméterként megkap egy természetes számot, és visszaadja a faktoriálisát. (0 faktoriálisa 1, 1 faktoriálisa 1, 2 faktoriálisa 2, 3 faktoriálisa 6, 4 faktoriálisa 24)
3. A Fibonacci-számsorozat első és második eleme 1. A következő elemet mindig úgy kapjuk, hogy az előző két elemet összeadjuk: 1, 1, 2, 3, 5, 8, 13, ... Készítsünk függvényt, ami meghatározza a bekért sorszámú Fibonacci-számot. Például: ha 7-et adunk meg, 13-at ad vissza.
4. A futár az egyes utakra az út hosszától függően kap fizetést a mellékelt táblázatnak megfelelően. Írjunk függvényt, ami a megadott út hosszának függvényében meghatározza a díjazás mértékét.

1–2 km	500 Ft
3–5 km	700 Ft
6–10 km	900 Ft
11–20 km	1400 Ft
21–30 km	2000 Ft

14. REKURZIÓ

Fogalma

A rekurzív szó jelentése önmagát tartalmazó, önmagára hivatkozó. Ennek alapján a rekurzió alatt olyan eljárásokat, függvényeket értünk, amik önmagukra hivatkoznak.

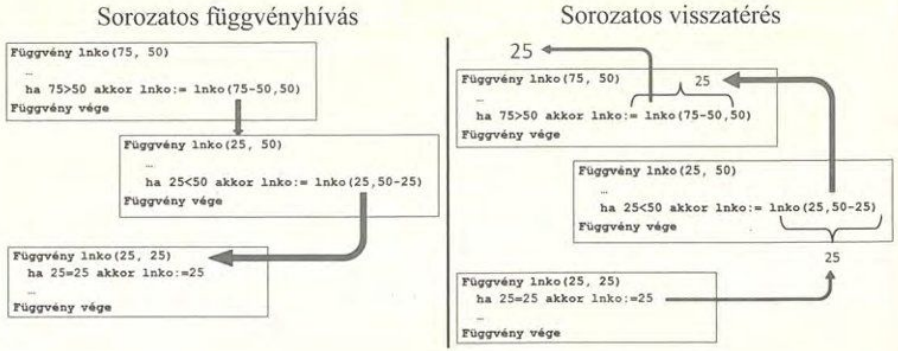
Bár a szakirodalom zömében az előző meghatározás elterjedt valamilyen formában, azért ezt az önmagukat kicsit pontosítanám: *önmaguk egy másik példányára.*

Tanulmányozzuk a mellékelt ábrát, ahol rekurzív algoritmussal keressük a legnagyobb közös osztót. Kövessük nyomon a működését. A 3. és 4. sor végén látható, hogy a függvény önmaga egy újabb példányát hívja meg más bemeneti értékekkel (a program ott folytatódik).

Működése

Nézzük meg a rekurzió működését az előbb látott példán keresztül.

Keressük a 75 és az 50 legnagyobb közös osztóját, így a függvény első meghívásakor ez a két paraméter. Az elágazás három ága közül az utolsó feltétele teljesül (4. sor). Ebben az ágban folytatódik a vezérlés, és újabb függvényhívás következik 75-50 és 50 paraméterekkel. Létrejön a függvény egy újabb példánya, ahol a középső ág feltétele teljesül, így ebben az ágban folytatódik a vezérlés, ahol újabb függvényhívás következik 25 és 50-25 paraméterekkel. A függvény immár harmadik példányának első sorában igaz a feltétel, tehát meg is van a visszatérési érték, a 25, és nincs további függvényhívás.



A függvény harmadik példányát a második példány hozta létre, így a harmadik példány visszatérési értékét is a második példány kapja. Behelyettesítődik a függvényhívás helyén, és íme, már a második függvénynek is van visszatérési értéke. A második példányt az első példány hozta létre, így a második példány visszatérési értékét az első példány veszi át. Az érték behelyettesítődik a függvényhívás helyén. Így már az első függvénynek is van visszatérési értéke, amit végül a főprogramnak ad vissza. A már nem használt függvény példány a paraméter visszaadása után megszűnik.

A most tárgyalt működésből következik, hogy a függvényhívások fázisában az éppen nem futó függvénypéldányt is tárolni kell, hiszen később használjuk. Ez sok hívás esetén memóriaigényes lehet.

Felépítése

Érettségin eddig nem kellett önállóan rekurzív algoritmust írni, de a követelmények szerint akár ez is megtörténhet. Ezért nézzük meg, milyen egy rekurzív függvény felépítése.

A függvénynek hívnia kell önmagát, tehát lesz legalább egy rekurzív hívást tartalmazó rész. A példában kettő is van, a 3. és 4. sorok végén.

A függvénynek egyszer be kell fejeződnie, tehát lesz egy rész, ahol a függvény konkrét visszatérési értéket ad (2. sor vége).

Lesz egy bázisfeltétel (a példában a = b), aminek a segítségével el lehet dönteni, hogy vége van-e a rekurzív hívások sorozatának.

Lesz egy elágazás a bázisfeltétel szerint, aminek egyik ágában a függvényhívás, másik ágában a konkrét értékadás lesz. Persze az értékadást és függvényhívást is megelőzhetik további műveletek.

Végül pedig szükség van egy olyan műveletre, ami minden egyes függvényhívással közelebb visz a rekurzió végéhez. A példában ez az a - b, illetve a b - a a 3. és 4. sorok függvényhívásaiban.

Ha rosszul írjuk meg a rekurzív alprogramot, előfordulhat, hogy végtelen sokszor kellene meghívnia magát. Így olyan sokszor jön létre új példány belőle, hogy megtelik az erre a célra használható memória, és a program hibaüzenettel megáll. Esetleg előtte látszólag sokáig nem tesz semmit, és csak utána áll meg hibaüzenettel.

Rekurzív és iteratív formák

A rekurzió is egyfajta ismétlés, a ciklusok méltó vetélytársának látszik. Amikor a feladatot rekurzió alkalmazása nélkül, ciklusokkal oldjuk meg, iteratív formáról beszélünk.

Ugyanannak a problémának létezhet rekurzív és iteratív megoldása is. Az iteratív megoldás általában gyorsabban fut, és kevésbé memóriaigényes. A rekurzív forma néha rövidebb, és az ember számára jobban megérthető.

A rekurzív függvények alkalmazása nem igényel új kódolási ismereteket.

Feladatok

1. Kódoljuk az alábbi, legnagyobb közös osztó kiszámolására szolgáló függvényt. Próbáljuk is ki.

```
Függvény lnko(a, b : egész számok) : egész szám
   ha a=b akkor lnko := a
   ha a<b akkor lnko := lnko(a, b-a)
ha a>b akkor lnko := lnko(a-b, b)
Függvény vége
```

2. Kódoljuk az alábbi, legkisebb közös többszörös kiszámolására szolgáló függvényt. Használjuk fel az előző feladat függvényét. Próbáljuk is ki.

```
Függvény lkkt(a, b : egész számok) : egész szám
   lkkt := a * b / lnko(a, b)
Függvény vége
```

3. N természetes szám faktoriálisa alatt 1-től N-ig a természetes számok szorzatát értjük. Írjunk rekurzív függvényt, ami paraméterként megkap egy természetes számot, és visszaadja a faktoriálisát. (0 faktoriálisa 1, 1 faktoriálisa 1, 2 faktoriálisa 2, 3 faktoriálisa 6, 4 faktoriálisa 24)
4. A Fibonacci-számsorozat első és második eleme 1. A következő elemet mindig úgy kapjuk, hogy az előző két elemet összeadjuk: 1, 1, 2, 3, 5, 8, 13, ... Készítsünk rekurzív függvényt, ami meghatározza a bekért sorszámú Fibonacci-számot. Például, ha 7-et adunk meg, 13-at ad vissza.

15. MODULÁRIS PROGRAMOZÁS ÉS TESZTELÉS

Az előzőekben láthattuk, hogy egy feladat modulokra bontása eljárások, illetve függvények segítségével áttekinthetőbbé teszi és megrövidíti a kódot, felosztathatóvá teszi a munkát, ráadásul könnyebb a tesztelés is.

A kérdés csak az, hogyan bonthatunk fel egy összetett feladatot modulokra, majd hogyan illeszthetjük őket ismét össze.

Modulokra bontás a közös részek kiemelésével

Először érdemes megfigyelni a feladatban fellelhető részleges ismétlődéseket. A mellékelt ábrán a feladatok a következők: a megadott tört egyszerűsítése, két tört szorzása, illetve két tört összeadása. A feladathoz rendelkezésünkre áll a legnagyobb közös osztót (lnko), valamint a legkisebb közös többszöröst (lkkt) meghatározó függvény.

1. Egyszerűsítés
   24/32 = 3/4
   24/6 = 4 (ha a tört értéke egész)
2. Tört szorzása
   24/32 * 12/15 = 288/480 = 3/5
   24/32 * 8/3 = 192/96 = 2
3. Tört összeadása (rendelkezésre áll az lkkt függvény)
   24/32 + 8/3 = 72/96 + 256/96 = 328/96 = 41/12
   22/4 + 27/6 = 66/12 + 54/12 = 120/12 = 10

Az ábrán jól látható, hogy a törtek egyszerűsítését minden művelet után el kell végezni. Ezért kár lenne minden művelethez megírni azt. Írjunk egy *egyszerűsít* nevű függvényt, és azt mindhárom esetben felhasználhatjuk.

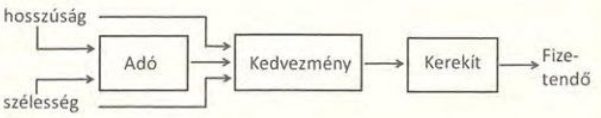
Az *egyszerűsít* függvény a legnagyobb közös osztót, az összead függvény pedig a legkisebb közös többszöröst meghatározó függvényt használja fel. Így elkészíthetjük a feladatsorunk "függvénytérképét", egy olyan ábrát, ahol látszik, melyik függvény melyiket használja. Ez az ábra mutatja, milyen sorrendben kell a függvényeket elhelyezni a kódban. (A függvényre történő hivatkozás előtt meg kell történnie a deklarációnak.)

Összefoglalva: Ezzel a módszerrel a közös részeket egy önálló függvénybe (eljárásba) emeljük ki.

Modulokra bontás a feladat szétszedésével

Egyetlen feladat önmagában is túl bonyolult lehet az egyben történő megíráshoz, vagy éppen a teszteléshez.

Általános tanács, hogy a feladat szövegében szorosan összetartozó dolgokat érdemes egyben hagyni, és a kevésbé összetartozók mentén szétbontani.




```
1) Függvény kerekit(fab : egész)
2) Függvény kedv(h: egész, sz: egész, p: egész)
3) Függvény ado...
4)   ...
5)   ado := kerekit(kedv(hossz,szel,f))
6) Függvény vége
7) Főprogram
8)     Ki: ado(40, 20)
```

Tekintsük például az alábbi feladatot:

Az önkormányzat telekadót fog kivetni. Az adót Fabatkában számolják. A 700 négyzetméteres és annál kisebb telkek esetén ez 51 Fabatka négyzetméterenként, az ennél nagyobb telkeknél az első 700 négyzetméterre vonatkozóan szintén 51 Fabatka, 700 négyzetméter felett egészen 1000 négyzetméterig 39 Fabatka a négyzetméterenkénti adó. Az 1000 négyzetméter feletti részért négyzetméterárat nem, csak 200 Fabatka egyösszegű átalányt kell fizetni. A 15 m vagy annál keskenyebb, illetve a 25 m vagy annál rövidebb telkek tulajdonosai 20% adókedvezményben részesülnek. Az adó meghatározásánál 100 Fabatkás kerekítést kell használni (pl. 6238 esetén 6200, 6586 esetén 6600). Írjunk függvényt, ami meghatározza egy telek adóját, ha megadjuk az oldalainak méretét.

A feladatot többször figyelmesen elolvasva rájöhetünk, hogy maga a szöveg is három részre tagolódik. Az elején a legnagyobb rész az adó meghatározásáról szól. Utána következik a kedvezmény megállapítása, majd végül a kerekítés.

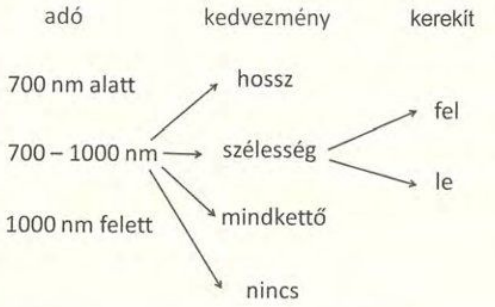
Ennek alapján érdemes megrajzolni a függvények egymásra épülésének térképét, feltüntetve a paramétereket és a visszaadott értékeket. Ennek segítségével már könnyen deklarálhatjuk a függvényeket, határozhatjuk meg a sorrendjüket a kódban (1–3. sor).

Ha a függvényeket külön-külön megírtuk, valahogy meg kell oldani az egymás után történő meghívásukat. Ezt megtehetjük például a függvények egymásba ágyazásával, azaz a kódban előrébb szereplő függvényt meghívjuk a későbbivel (5. sor).

Moduláris és funkcionális tesztelés

A modulokra bontás általában nagyban leegyszerűsíti a tesztelést is. Számoljuk át, hogy a teljes lefedés elve alapján történő teszteléshez hány tesztesetre van szükség modulokra bontva, és hányra egyben a telekadós példában.

A kerekít függvény teszteléséhez két teszteset elegendő, mivel felfelé (2. sor) vagy lefelé (1. sor) kerekíthetünk. A kedvezményt számoló függvényt négy esetből tudjuk tesztelni a feltételek alapján: hosszúság alapján jár kedvezmény (3. sor), szélesség alapján jár kedvezmény (4. sor), mindkettő alapján jár kedvezmény (5. sor), egyik alapján sem jár kedvezmény (6. sor). Végül az adó-függvény három esettel fedhető le: 700 négyzetméter alatt (7. sor), 700 és 1000 négyzetméter között (8. sor), és 1000 négyzetméter felett (9. sor). Ha ehhez hozzávesszük a függvények összeillesztésének tesztelését, összesen 2+4+3+1 = 10 tesztesetet kapunk.




```
1) Ki: kerekit(6238)
2) Ki: kerekit(6586)
3) Ki: kedv(20, 20, 1000)
4) Ki: kedv(30, 10, 1000)
5) Ki: kedv(20, 10, 1000)
6) Ki: kedv(30, 20, 1000)
7) Ki: ado(30, 20)
8) Ki: ado(40, 20)
9) Ki: ado(40, 30)
```

Ha egyben hagytuk volna a feladatot (vagy csak egyben tesztelnénk), ez a szám 2*4*3=24 lenne, hiszen például a 700 négyzetméter alatti telek lehet hossz miatt kedvezményes, szélesség miatt kedvezményes, mindkettő miatt kedvezményes, vagy egyik miatt sem. S mindezek az esetek kerekedhetnek felfelé vagy lefelé. (Lásd bal oldali ábra.) Ami talán ennél is időigényesebb, az összes megfelelő teszteset megtalálása, hiszen honnan tudnánk előre, hogy például az 1000 négyzetméter feletti, kedvezményes hosszúságú telek adója lefelé vagy felfelé fog kerekedni a végén?

Mikor a modulokat önállóan, azaz a többi modultól függetlenül teszteljük, modultesztelésről beszélünk.

Időnként azonban a modulteszt nem elegendő. Egyrészt, mert ritkán történik mindenre kiterjedő alaposságú tesztelés. Másrészt a modulok összeillesztésében is lehet hiba. Ezért általában szükség van a teljes feladat összefüggő tesztjére is.

Az egymással összekapcsolt modulok együttes tesztelését funkcionális tesztelésnek nevezzük.

Mint fentebb láthattuk, a funkcionális tesztelés jóval több esetet jelentene, amit ráadásul gondos tervezéssel kellene kiválasztani. Mindez nagyon időigényes, ezért általában kiválasztanak néhány (szélsőséges esetben egyetlen) jellemző bemeneti paraméterekkel rendelkező tesztesetet, és csak ezekkel (ezzel) végzik el a funkcionális tesztet.

Az adó-kiszámítós példában a 7–9. sorok bármelyikét választhatjuk funkcionális tesztnek. A különbség a moduláris teszteléshez képest az, hogy itt már elvégeztük a függvények egymásba ágyazását, míg a modultesztelésnél csak a kedvezmény és kerekítés nélküli adó kiszámítását teszteljük.

Néhány funkcionális teszt nagyon alapos modulteszteléssel párosítva a legtöbb esetben már elfogadható üzembiztonságot ad.

Természetesen egy érettségin nem lesz időnk arra, hogy ilyen alaposan megtervezzük a tesztelést. A modulteszteket általában a modulok írása közben többé-kevésbé letudjuk, a funkcionális tesztből általában beérjük eggyel az egyes feladatok elkészülte után.

A folyamatosan fejlesztés alatt álló szoftvereket a cégek gyakran automatikusan tesztelik. Mind a modul-, mind a funkcionális teszteket rendszeres időközönként egy program futtatja, s hiba esetén jelzést küld a fejlesztőknek. (A folyamatos fejlesztés miatt a modulok változhatnak.) Így takarítják meg a tesztelésre szánt drága munkaórák egy tekintélyes részét[39].

Végül megemlítendő, hogy a szoftverhibák (majd a különböző javítócsomagok) közvetlen oka a nem megfelelő alaposságú tesztelés, ami mögött egyrészt a szoros határidők állnak, másrészt az a szemlélet, hogy az ügyfél nem fizet a tesztelésért[40]. Az sem mellékes, hogy a szoftverfejlesztők általában nem szeretnek tesztelni. Mára gyakran a szoftvertesztelő nemcsak külön munkakörként, hanem külön foglalkozásként jelenik meg.

39 Magyarországi főiskola is alkalmazta a tesztautomatizálást a hallgatók által beadott programok értékelésére. Maga az automatizálás is munkaidőt igényel, így alaposan meg kell gondolni, megéri-e.

40 Az ügyfél igyekszik minél alacsonyabb árat kiharcolni, míg a szoftverfejlesztő cég a magas óradíjban (kevés munkával sok pénzt) érdekelt. Ennek eredménye a kevesebb ráfordított munkaóra. A legegyszerűbb a tesztelési időt megnyirbálni.

Feladatok

1. Az önkormányzat telekadót fog kivetni. Az adót Fabatkában számolják. A 700 négyzetméteres és annál kisebb telkek esetén ez 51 Fabatka négyzetméterenként, az ennél nagyobb telkeknél az első 700 négyzetméterre vonatkozóan szintén 51 Fabatka, 700 négyzetméter felett egészen 1000 négyzetméterig 39 Fabatka a négyzetméterenkénti adó. Az 1000 négyzetméter feletti részért négyzetméterárat nem, csak 200 Fabatka egyösszegű általányt kell fizetni. A 15 m vagy annál keskenyebb, illetve a 25 m vagy annál rövidebb telkek tulajdonosai 20% adókedvezményben részesülnek. Az adó meghatározásánál 100 Fabatkás kerekítést kell használni (pl. 6238 esetén 6200, 6586 esetén 6600). Írjunk függvényt, ami meghatározza egy telek adóját, ha megadjuk az oldalainak méretét. Bontsuk több részfeladatra, függvényekre.)
2. Végezzük el az előző feladat modultesztelését a teljes lefedés elve szerint. A tesztesetek legyenek ott a főprogramban megjegyzésként.
3. Végezzük el a 1. feladat funkcionális tesztelését. A tesztesetek legyenek ott a főprogramban megjegyzésként.
4. Írjunk programot, ami bekéri két közönséges tört számlálóját és nevezőjét. A törteket külön-külön leegyszerűsíti. Ha felírhatók egész számként, az egész értéküket jeleníti meg. A két bekért törtet összeszorozza, illetve összeadja, és az eredményeket leegyszerűsítve, ha lehetséges egész számként megadja. Felhasználható a Rekurzió fejezetnél felírt legnagyobb közös osztó, illetve legkisebb közös többszörös függvény. Az ismétlődéseket emeljük ki függvénybe.
5. Végezzük el az előző feladat modultesztelését a teljes lefedés elve szerint. A tesztesetek legyenek ott a főprogramban megjegyzésként.
6. Végezzük el a 4. feladat funkcionális tesztelését. A tesztesetek legyenek ott a főprogramban megjegyzésként.

V. ÖSSZETETT ADATSZERKEZETEK

1. LISTA

Bevezetés

Ha sok változóra van szükségünk (pl. 100-ra), hosszú időbe telne, mire mindegyiknek saját nevet adnánk. Problémát okoz a nevek megjegyzése, valamint az, hogy a későbbiekben hogyan hivatkozzunk ezekre. Megoldást jelent, ha a sok változónak egyetlen nevet adunk, de mellette sorszámozzuk őket. Például: honap(2). Ezek lesznek a listák.

A lista névvel és sorszámokkal ellátott összetett változó. A lista elemeire a nevével és a sorszámukkal együtt hivatkozunk.

Elképzelhetjük a listákat, mint egy fiókos szekrényt. A szekrénynek nevet adunk, például hónap, a fiókjait pedig megszámozzuk. A fiókok tartalma pedig a változóban tárolt érték lesz.

A listaelem sorszámát **index**nek nevezzük.

Felfoghatunk egy listát egy számozott felsorolásként, aminek neve van: például a *honap* nevű felsorolás (lista), aminek 0. eleme a január, 1. eleme a február, stb.

A Pythonban a lista elemei akár különböző típusúak is lehetnek, sőt akár egy másik lista is lehet listaelem.

Más programnyelvekben és az általános programozás módszertanban létezik a **tömb** fogalma. A fenti példa lehetne akár tömb is. A lista egy sokoldalúbb adatszerkezet: olyan műveleteket hajthatunk végre, amiket egy tömbön nem (hozzáfűzés, törlés, rendezés, stb.), va-