



Eötvös Loránd Tudományegyetem  
Informatikai Kar

## **Alkalmazott modul: Programozás**

---

### **6. fejezet**

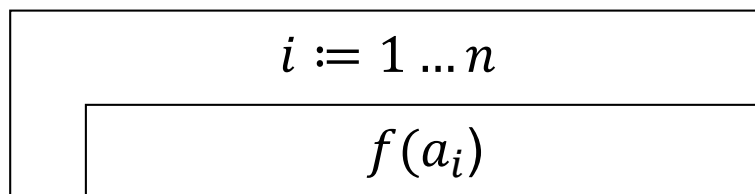
### **Procedurális programozás: iteratív és rekurzív alprogramok**

---

**Giachetta Roberto**

A jegyzet az ELTE Informatikai Karának 2015. évi  
Jegyzetpályázatának támogatásával készült

- *Elemenkénti feldolgozásnak* nevezzük azon algoritmusokat, amely a bemenetként megadott adatsorozat minden egyes elemére alkalmaznak egy adott műveletet



- pl. összegzés, lineáris keresés, maximumkeresés
- az algoritmus magját ekkor egy ciklus adja, amely valamilyen sorrendben (szekvenciában) bejárja az adatsorozatot, ezt *iterációnak* nevezzük
- az iterációt végző alprogramokat nevezzük *iteratív alprogramoknak*

# Alprogramok

## Iteratív alprogramok

---

- Az iteráció sorrendjét az adatsorozat szerkezetének megfelelően a programozó adja meg
  - egyes adatsorozatok (pl. konzol képernyő) csak egyféleképpen, más mások többféleképpen is feldolgozhatóak
  - sokszor eleve a sorozat szerkezete sem szekvenciális
  - pl. mátrixok esetén két gyakori módszer:
    - *sorfolytonos* bejárás (a sorban vízszintesen feldolgozzuk az összes elemet, majd haladunk a következő sorra)
    - *oszlopfolytonos* bejárás (az oszlopban függőlegesen feldolgozzuk az összes elemet, majd haladunk a következő oszlopra)

# Alprogramok

## Iteratív alprogramok

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

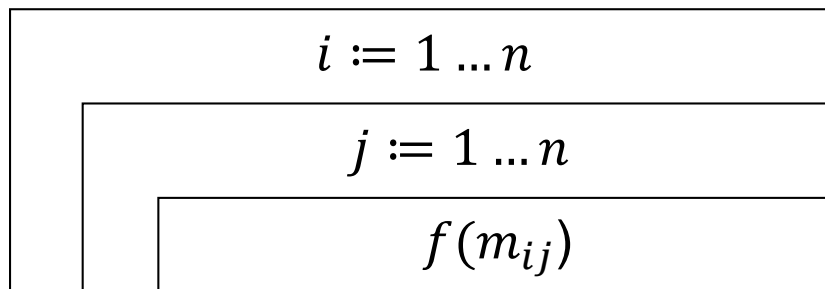
sorfolytonos bejárás:

1, 2, 3, 4, 5, 6, 7, 8, ...

oszlopfolytonos bejárás:

1, 5, 9, 13, 2, 6, 10, 14, ...

- mivel a mátrixok sor és oszlop szerint indexeltek, ezért a bejárás két futóindex szerint halad, vagyis két, egymásba ágyazott ciklussal



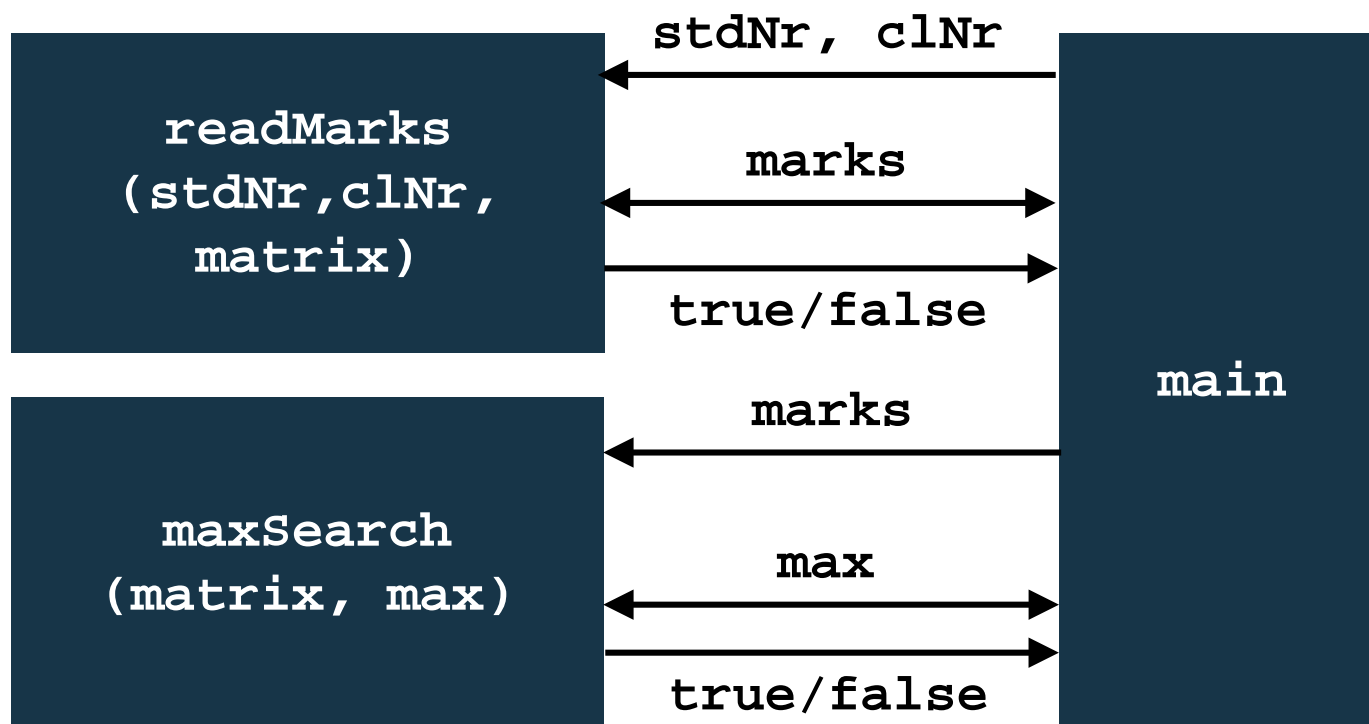
# Alprogramok

## Példa

*Feladat:* Mátrixban tároljuk egy osztály adatait, minden sora egy diák, minden oszlopa egy tantárgy, a mátrix értékei a jegyek. Adjuk meg a legjobb jegyet.

- maximumkeresést kell végeznünk a teljes mátrixra, az elemek maximumát keressük, ezt kihelyezzük egy alprogramba
- mivel újrafelhasználható alprogramot készítünk, fel kell készülnünk arra, hogy a mátrix üres, ezért feltételes maximumkeresést végzünk
- a mátrix feltöltését is elemenkénti feldolgozással végezzük, és sorfolytonosan haladunk
- a diákok és tárgyak számát bekérjük a főprogramban

*Tervezés:*



# Alprogramok

## Példa

*Megoldás:*

```
bool maxSearch(vector<vector<int> > matrix,
               int& max){
    // feltételes maximumkeresés

    bool l = false;
    // ha üres a mátrix, akkor hamissal térünk
    // vissza

    // maximumkeresés a teljes mátrixra
    for (int i = 0; i < matrix.size(); i++)
        for (int j = 0; j < matrix[i].size(); j++)
        {
```

*Megoldás:*

```
        if (!l) {
            l = true;
            max = matrix[i][j];
        }
        else {
            if (matrix[i][j] > max)
                max = matrix[i][j];
        }
    }

    return l;
}
```



# Alprogramok

## Példa

*Megoldás:*

```
int main(){
    int stdNr, clNr, max;
    vector<vector<int> > marks;

    cout << "Hallgatók száma: "; cin >> stdNr;
    cout << "Tantárgyak száma: "; cin >> clNr;

    if (!readMarks(stdNr, clNr, marks)){
        // beolvasás, amely lehet sikertelen
        cout << "A beolvasás sikertelen!" << endl;
        return 1;
    }
    // sikeres volt a beolvasás
```

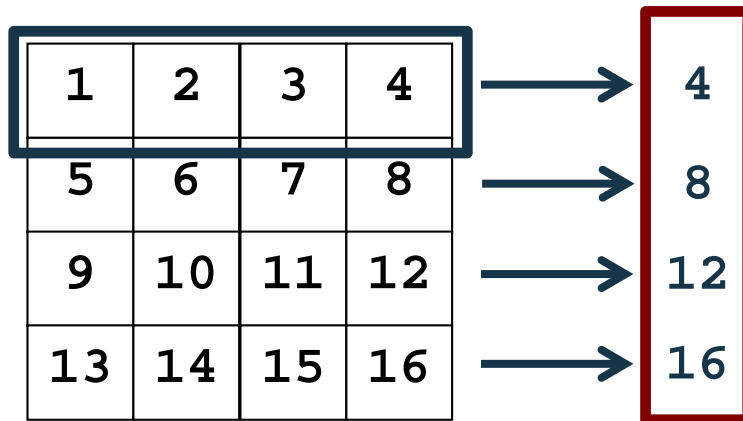
*Megoldás:*

```
if (!maxSearch(marks, max)){  
    // maximumkeresés, amely lehet sikertelen  
    cout << "Nincsenek adatok!" << endl;  
    return 2;  
}  
// sikeres maximumkeresés  
cout << "A legjobb jegy: " << max << endl;  
  
return 0;  
}
```

# Alprogramok

## Iteratív alprogramok

- Az adatsorozatot nem csak egy, de több lépésben is feldolgozhatjuk
  - egyes részsorozatokot külön feldolgozunk, majd a kapott részeredményeket ismételten feldolgozhatjuk
  - pl. egy mátrix esetén külön végezhetünk maximumkeresést a sorokra, majd a sorok maximumára



# Alprogramok

## Iteratív alprogramok

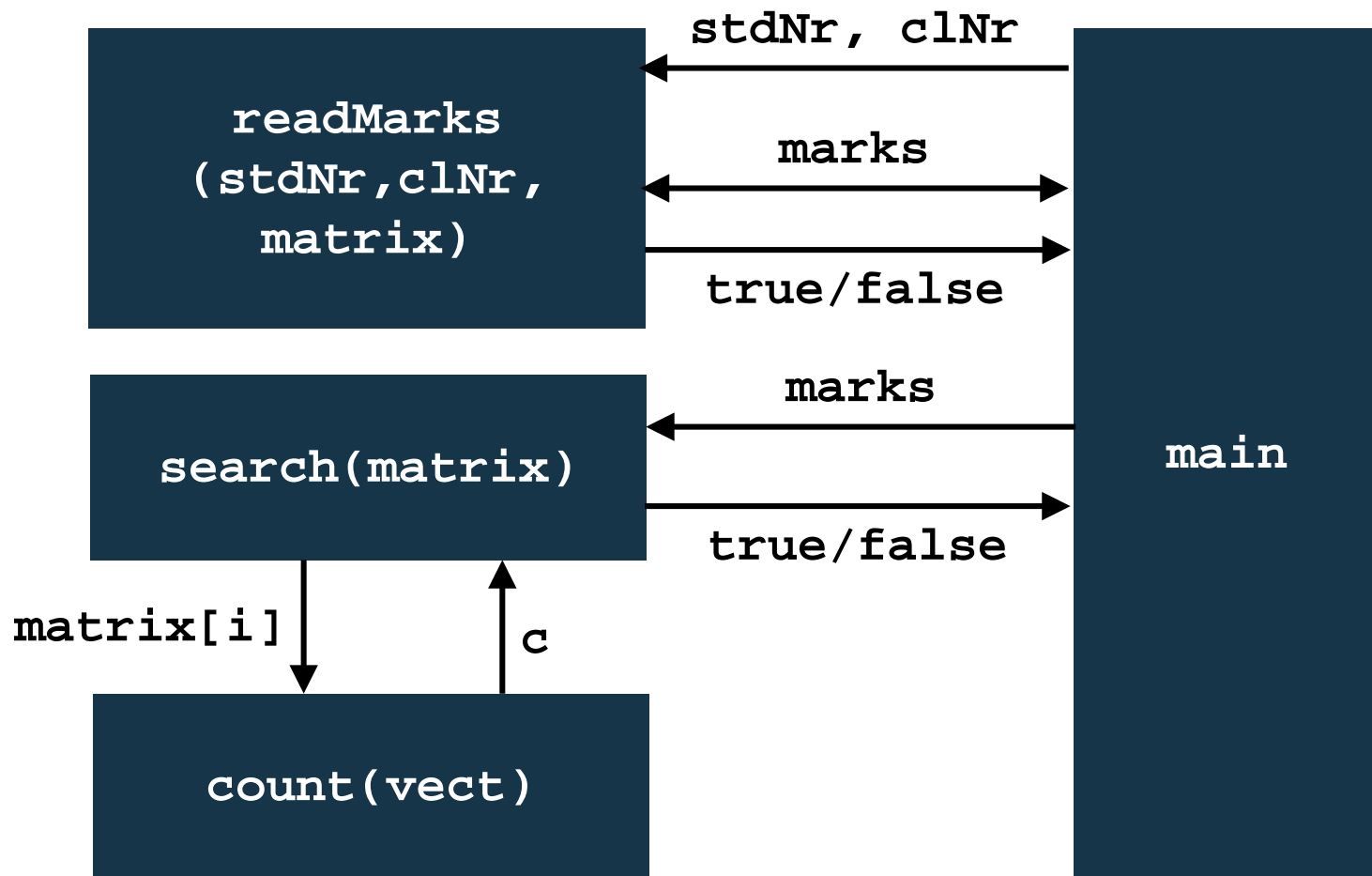
---

- A részsorozatok feldolgozására és a részeredmények feldolgozására használhatunk külön programozási tételt
  - pl. ha a sorok átlagának maximumát akarjuk megállapítani, a sorokon összegzést, az átlagokon maximumkeresést végzünk
  - ekkor *beágyazzuk* az egyik tételt a másikba, és keletkezik egy *külső* és egy *belső programozási tétel*
  - ezeket a tételeket külön alprogramokba helyezhetjük, mindkét alprogram egyszerű ciklussal végez elemenkénti feldolgozást
  - a belső alprogram általában csak a megfelelő részsorozatot kapja meg, pl. mátrix egy sorát

*Feladat:* Mátrixban tároljuk egy osztály adatait, minden sora egy diák, minden oszlopa egy tantárgy, a mátrix értékei a jegyek. Adjuk meg, van-e olyan hallgató, akinek évet kell ismételnie (legalább három tárgyból bukott).

- egy számlálást kell végeznünk a hallgatókon (a mátrix sorai), amelynek feltétele a legalább három tárgyból bukás
- lineáris kereséssel megállapítjuk, hogy van-e olyan hallgató, akire a számlálás legalább hármat adott
- tehát egy lineáris keresésbe ágyazunk számlálást
- a lineáris keresés a mátrix sorain megy végig, míg a számlálás a sorokon belül dolgozik (tehát elég a mátrix sorát átadni paraméterként)

*Tervezés:*



# Alprogramok

## Példa

*Megoldás:*

```
int count(vector<int> vect)
{
    int c = 0; // számlálás tétele

    for (int i = 0; i < vect.size(); i++)
    {
        if (vect[i] == 1)
            c++;
    }

    return c;
}
```

*Megoldás:*

```
bool search(vector<vector<int> > matrix)
{
    // lineáris keresés
    bool l = false;

    for (int i = 0; i < matrix.size() && !l; i++)
    {
        l = (count(matrix[i]) >= 3);
        // átadjuk a mátrix egy sorát
    }

    return l;
}
```



*Megoldás:*

```
int main(){
    ...
    if (!search(marks)) // keresés
    {
        cout << "Nincs évismétlő diák!" << endl;
    }
    else
    {
        cout << "Van évismétlő diák!" << endl;
    }

    return 0;
}
```

# Alprogramok

## Deklaráció és definíció szétválasztása

- Lehetőségünk van szétválasztani az alprogram *deklarációját* a *definíciójától*
  - a deklaráció tartalmazza az alprogram nevét, típusát, valamint paramétereinek típusát (a nevük nem kötelező):  
`<típus> <név>(<paraméterek típusa>);`
  - a definíció megegyezik magával az alprogrammal:  
`<típus> <név>(<paraméterek>){ <utasítások> }`
  - ezáltal átláthatóbbá tehető a programkód, pl.:  
`void func(int); // deklaráció a főprogram előtt`  
`int main(){ ... /* itt használhatjuk Func-t */ }`  
  
`void func(int a){ ... /* utasítások */ }`

- Lehetőségünk van ugyanolyan névvel különböző paraméterezésű alprogramok létrehozására, ezt az alprogram *túlterhelésének* nevezzük
  - az alprogramok neve megegyezik, visszatérési típusa különbözhet, formális paramétereinek száma, vagy típusa kötelezően különböző kell, hogy legyen
  - a program az átadott aktuális paraméterek száma, illetve típusa függvényében dönti el, melyik túlterhelt változatot futtatja
  - akkor hasznos, ha ugyanazon tevékenységet különböző értékekkel, vagy befolyásoló tényezőkkel szeretnénk elvégezni

- Pl.:

```
int max(int a, int b) { // két szám maximuma
    return a > b ? a : b;
}
```

```
string max(string a, string b) { // két szövegé
    return a.length() > b.length() ? a : b;
}
```

```
int max(int a, int b, int c) { // 3 szám maximuma
    return max(a, max(b, c));
}
```

```
int max(vector<int> v){ ... } // egy vektor maximuma
```

...

```
x = max(10, 3, 8); // a 3. változatot hívja meg
```

```
y = max("hello", "world"); // ez a 2. változatot
```

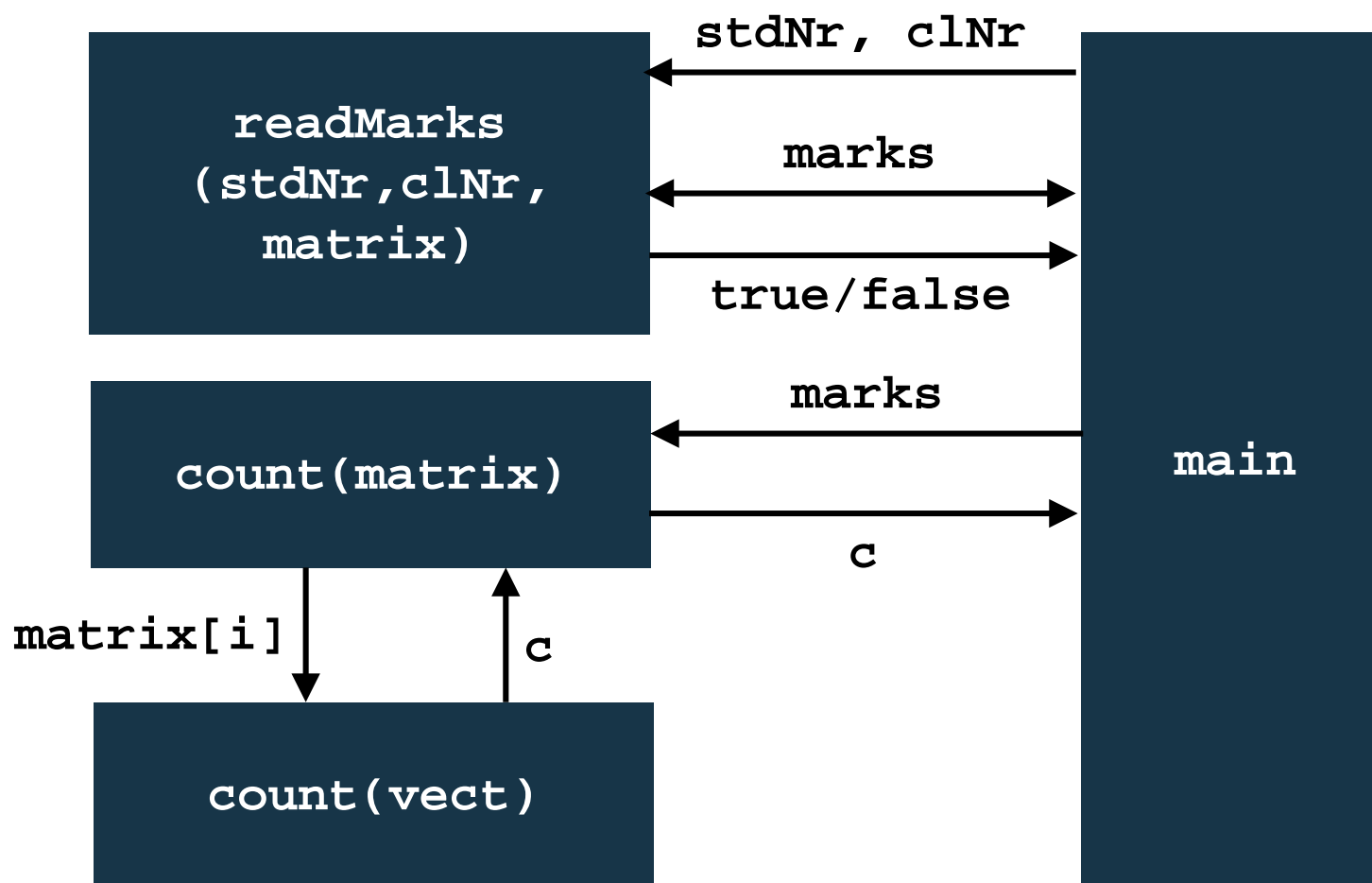
# Alprogramok

## Példa

*Feladat:* Mátrixban tároljuk egy osztály adatait, minden sora egy diák, minden oszlopa egy tantárgy, a mátrix értékei a jegyek. Adjuk meg, hány hallgatónak kell évet ismételnie (legalább három tárgyból bukott).

- egy számlálást kell végeznünk a hallgatókon (a mátrix sorai), amelynek feltétele a legalább három tárgyból bukás
- a bukások számát szintén számlálással tudjuk megadni, az előző számlálás eredményeiből kiindulva
- tehát egy számlálásba ágyazunk másik számlálást
- a két alprogramot nevezhetjük ugyanúgy, hiszen más a paraméterezésük, így túlterhelés lép életbe

*Tervezés:*



# Alprogramok

## Példa

*Megoldás:*

```
// alprogram deklarációk:
bool readMarks(int stdNr, int clNr,
               vector<vector<int> >& matrix);
int count(vector<int> vect);
    // túlterhelést használunk a count függvényre
int count(vector<vector<int> > matrix);

// főprogram:
int main()
{
    ...
}
```

# Alprogramok

## Példa

*Megoldás:*

```
int count(vector<vector<int> > matrix){  
    int c = 0;  
    // számlálás tétele a teljes mátrixon  
  
    for (int i = 0; i < matrix.size(); i++)  
    {  
        if (count(matrix[i]) >= 3)  
            c++;  
        // átadjuk a mátrix egy sorát  
    }  
  
    return c;  
}
```



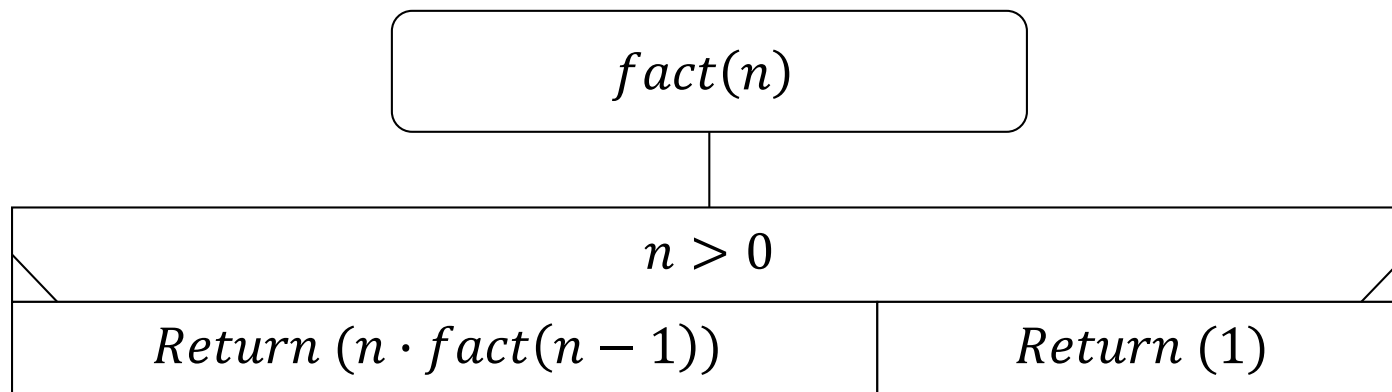
- Mivel alprogram hívhat másik alprogramot, az is engedélyezett, hogy meghívja saját magát, ezt *rekurzió*nak nevezzük, pl.:

```
void func(...) { ... func(...) ... }
```

- Rekurzió esetén be kell tartanunk bizonyos szabályokat:
  - az alprogram lefutása nem vezethet minden esetben rekurzív híváshoz
  - az átadott paraméterek nem egyezhetnek meg a kapott paraméterekkel
  - a paramétereknek az átadáskor úgy kell változnia, hogy idővel befejeződjön a rekurzió

- Pl. a faktoriális számítás megadható az  $n! = n \cdot (n - 1)!$  képlettel
  - itt mindkét oldalon szerepel a faktoriális, így kifejezhető rekurzióval:

$$fact(n) = \begin{cases} n \cdot fact(n - 1) & \text{ha } n > 0 \\ 1 & \text{ha } n = 0 \end{cases}$$



- itt teljesülnek a rekurzív függvény követelményei
  - egy rekurzív hívás van, elágazás után, amely a paramétert vizsgálja
  - a paraméter értéke minden lépésben csökken, idővel eléri a nem rekurzív ágot
- megvalósítás:

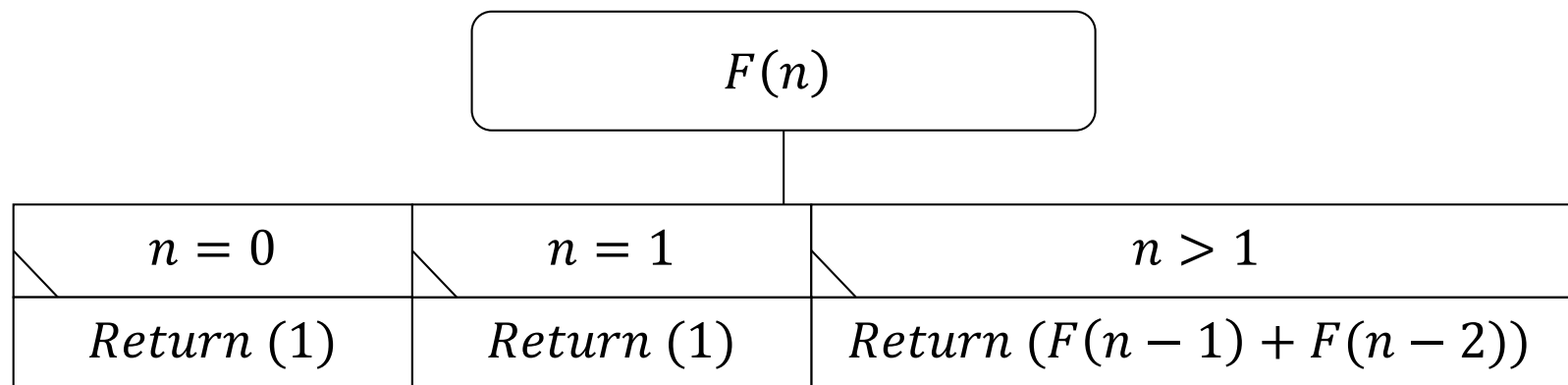
```
int fact(int n)
{
    if (n > 1)
        return n * fact(n - 1); // rekurzív hívás
    else
        return 1;
}
```

# Alprogramok

## Rekurzió

- Pl. a Fibonacci-számok esetén az előző kettő összegét vesszük, vagyis  $F_n = F_{n-1} + F_{n-2}$
- itt jobb oldalon kétszer is szerepel a bal oldali eredmény:

$$F(n) = \begin{cases} 0 & \text{ha } n = 0 \\ 1 & \text{ha } n = 1 \\ F(n-1) + F(n-2) & \text{ha } n > 1 \end{cases}$$



- megvalósítás:

```
int F(int n)
{
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;
    // az elágazás elhanyagolható

    return F(n - 1) + F(n - 2);
    // rekurzív hívás
}
```

- A rekurzió lehet
  - *egyszeres*: egy rekurzív hívás történik, pl. elemenkénti feldolgozások esetén (faktoriális)
  - *többszörös*: több rekurzív hívás történik, pl. Fibonacci-számok, elágazó adatszerkezetek bejárása (fa, gráfok)
- Sok feladatra iteratív és rekurzív megoldás is adható, azonban hatékonyságuk különbözhet
  - pl. a Fibonacci számok esetén a rekurzív változat a köztes értékeket többször is kiszámítja:
$$F(n) = F(n - 1) + F(n - 2) =$$
$$F(n - 2) + F(n - 3) + F(n - 2) = \dots$$

- Elemenkénti feldolgozás esetén a rekurzió könnyen átfogalmazható iteratívra
  - az iteratív ciklus léptetése megfelel a rekurzív hívásnak
  - pl. a faktoriális iteratív megfogalmazása (összegzéssel):

$$fact(n) = \prod_{i=1}^n i$$

$s := 1, i := 1$	
$i \leq n$	
	$s := s \cdot i$
	$i := i + 1$

# Alprogramok

## Rekurzió átalakítása iterációra

- A rekurzív alprogramok iteratív átalakítására szolgál a *rekurzív függvények kibontásának* programozási tétele
  - lényege, hogy az értékeket iteratívan számoljuk, a korábban kiszámolt értékeket pedig ne számoljuk ki újra, hanem tároljuk el, és léptessük az értékeket
  - pl. a Fibonacci számítás iteratív megfogalmazása:

$f := 1, x := 0, i := 2$	
$i \leq n$	
$y := f + x, x := f, f := y$	
$i := i + 1$	



- megvalósítás:

```
int F(int n){  
    int f = 1, x = 0; // inicializálás  
    for (int i = 2; i <= n; i++)  
    {  
        int y = f + x;  
        // a következő érték számítása  
        x = f; // értékek léptetése  
        f = y;  
    }  
    return f; // eredmény  
}
```

# Alprogramok

## Rekurzió átalakítása iterációra

- működése:

