Programozás 2

3.

PARANCSSORI ARGUMENTUMOK

Parancssori argumentumok

- A C nyelvet támogató környezetekben lehetőség van arra, hogy a végrehajtás megkezdésekor a programnak parancssori argumentumokat vagy paramétereket adjunk át.

Parancssori argumentumok

- Amikor az operációs rendszer elindítja a programot, azaz meghívja a **`main`**-t, a hívásban két argumentum szerepelhet:
  - Az első (általában **`argc`**) azoknak a parancssori argumentumoknak a darabszáma, amelyekkel a programot meghívtuk.
  - A második argumentum (általában **`argv`**) egy mutató egy sztring-tömbre, amely a parancssori argumentumokat tartalmazza. Egy karakterlánc egy argumentumnak felel meg.

Parancssori argumentumok

- Megállapodás szerint **`argv[0]`** az a név, amellyel a programot hívták, így az **`argc`** értéke legalább 1.
- Számíthatunk arra is, hogy **`argv[argc]==NULL`**.
- Mivel **`argv`** mutatótömböt megcímző mutató, többféleképpen is megírhatjuk azt a programot, amely kiírja a parancssorban lévő argumentumokat.

Parancssori argumentumok

```
/* Kiírjuk a parancssorban lévő argumentumokat.
 * 1998. Április 16. Dévényi Károly, devenyi@inf.u-szeged.hu
 */

#include <stdio.h>

main(int argc, char **argv)
/* vagy így is lehet az argv-t deklarálni
main(int argc, char *argv[])
*/
{
    int i;
    printf("argc = %d\n\n",argc);
    for (i=0; i<argc; ++i) {
        printf("argv[%d]: %s\n", i, argv[i]);
    }
}
```

Parancssori argumentumok

- Mentsük el a fenti programot **`arg.c`** néven és fordítsuk le!

```
$ gcc -o arg arg.c
$ ./arg alma korte szilva barack palinka
argc = 6

argv[0]: ./arg
argv[1]: alma
argv[2]: korte
argv[3]: szilva
argv[4]: barack
argv[5]: palinka
```

REKURZIÓ

Hanoi tornyai

- Problémafelvetés:
  - Egy oszlopon egyre csökkenő átmérőjű korongok vannak. Pakoljuk át a korongokat egy másik oszlopra úgy, hogy
    - Egyszerre csak egy korongot mozgatunk, amelyik valamelyik oszlop tetején van
    - Nagyobb átmérőjű korong nem kerülhet kisebbre
    - Rendelkezésre áll egy kezdetben szabad oszlop is

Hanoi tornyai

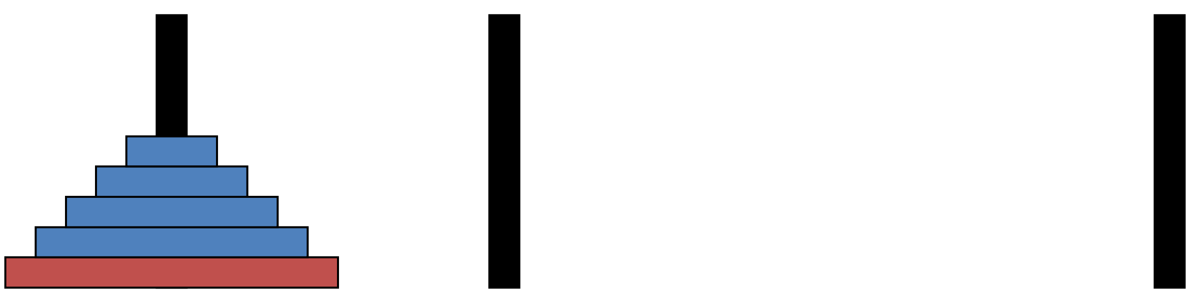
- Specifikáció:
  - Input
    - Db pozitiv egész szám, a torony magassága
    - Két különböző, pozitiv egész szám: Honnan és Hova (1<=Honnan, Hova<=3), melyek jelentése, hogy melyik toronyról melyik toronyra kell átpakolni.
  - Output
    - Egy tevékenységsorozat szövegesen, amit mechanikusan végrehajtva ténylegesen átpakolhatjuk a tornyot.

Hanoi tornyai

- Algoritmustervezés:
  - Készítsünk egy rekurzív eljárást, amelyik az N magasságú torony átpakolását visszavezeti az N-1 magasságú torony átpakolására.
  - Az 1 magasságú torony átpakolása nem igényel előkészületet, azonnal elvégezhető.

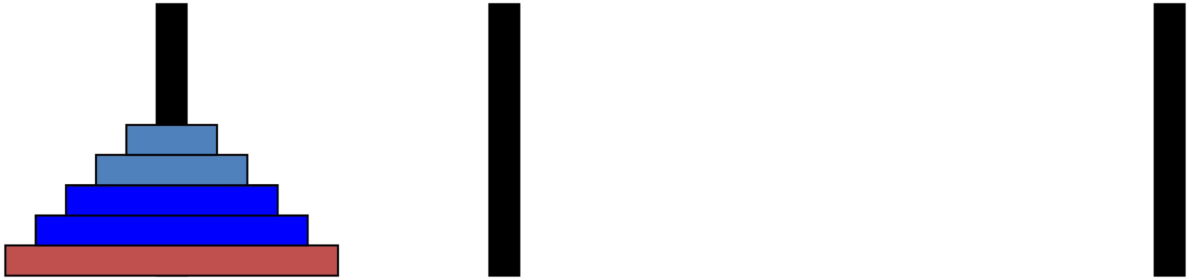
Hanoi tornyai

- Algoritmustervezés:
  - Az N magasságú torony átpakolását visszavezetjük az N-1 magasságú torony átpakolására.



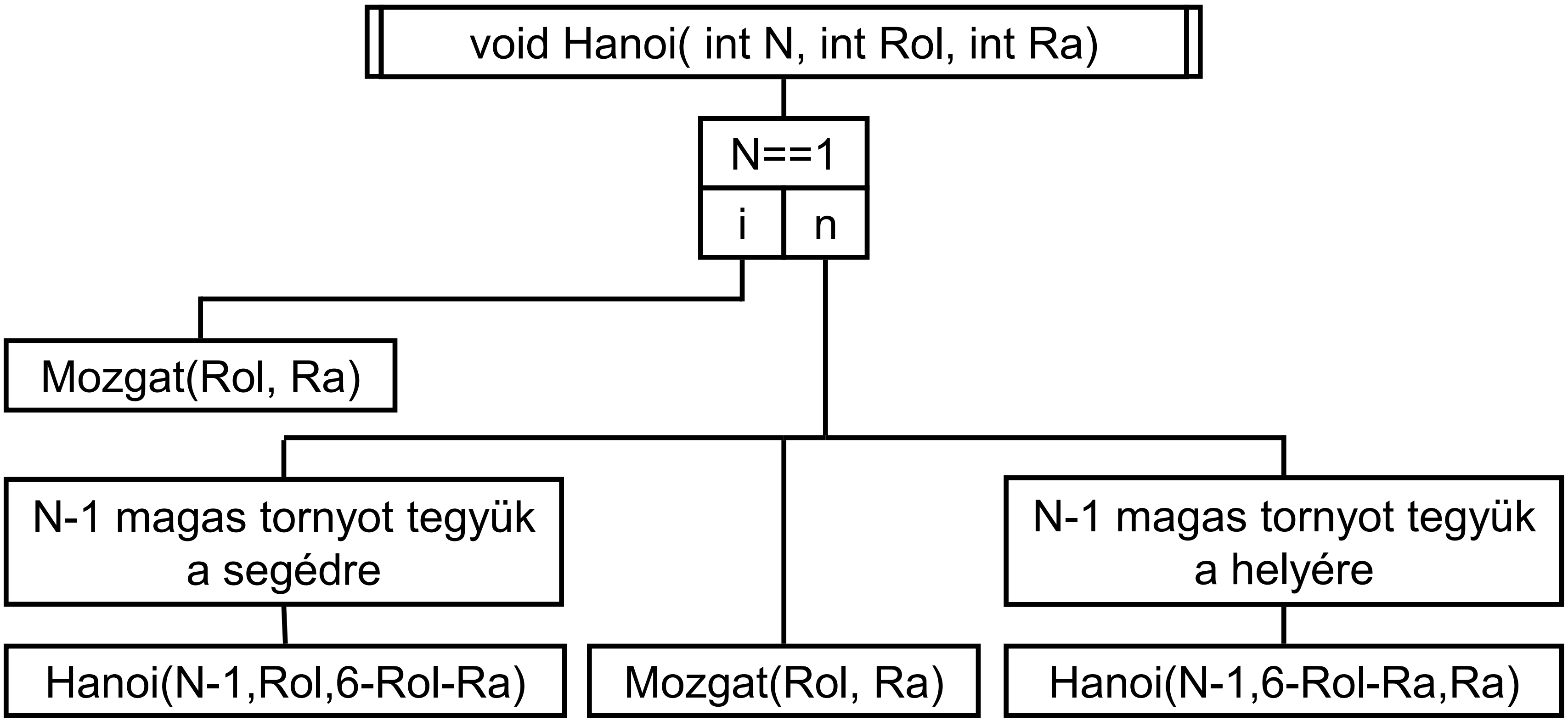
Hanoi tornyai

- Megjegyzés:
  – Mivel az X magasságú torony, amit az egyik rúdról a másikra pakolunk mindig az X legkisebb korongból áll, a harmadik rudat akkor is használhatjuk segédrúdként, ha azon van korong, mivel ez biztosan nagyobb, mint a legnagyobb, amit mi át szeretnénk pakolni.



Hanoi tornyai

- Algoritmustervezés:



Hanoi tornyai

```
/* A Hanoi tornyai játék megvalósítása rekurzív eljárással.
 * 1997. Október 31.  Dévényi Károly, devenyi@inf.u-szeged.hu
 */

#include <stdio.h>

int Honnan;                         /* erről a toronyról kell átrakni */
int Hova;                           /* erre a toronyra */
int Db;                             /* a torony ennyi korongból áll */

void Mozgat(int Innen, int Ide)
{  /* Átrak egy korongot Innen Ide */
  printf(" Tegyünk át egy korongot");
  printf(" a %d. oszlopról a %d. oszlopra!\n", Innen, Ide);
}
```

>>>

Hanoi tornyai

```
void Hanoi(int N,            /* ilyen magas a torony */
           int Rol,          /* erről a toronyról */
           int Ra)           /* erre a toronyra */
{
  if (N == 1) {
    Mozgat(Rol, Ra);
  } else {
    Hanoi(N - 1, Rol, 6 - Ra - Rol);
    Mozgat(Rol, Ra);
    Hanoi(N - 1, 6 - Ra - Rol, Ra);
  }
}
```

>>>

Hanoi tornyai

```
main(int argc, char *argv[])
{
  printf("Kérem adja meg a torony magasságát: ");
  scanf("%d%*[^\n]", &Db);
  getchar();
  printf("Kérem adja meg, hogy a torony hol áll? (1,2,3) ");
  scanf("%d%*[^\n]", &Honnan);
  getchar();
  printf("Kérem adja meg, hogy melyik oszlopra tegyük át? ");
  scanf("%d%*[^\n]", &Hova);
  getchar();
  if (Db > 0 && 1 <= Honnan && Honnan <= 3 && 1 <= Hova &&
      Hova <= 3 && Honnan != Hova) {
    Hanoi(Db, Honnan, Hova);
  } else {
    printf("Hibás adat\n");
  }
}
```

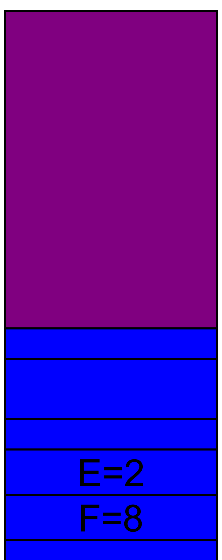
Rekurzió

- Az előző példában rekurzív függvénydeklarációt láthattunk.
- A C nyelven bármelyik függvény lehet rekurzív illetve részt vehet rekurzív függvényrendszerben.

Végrehajtás

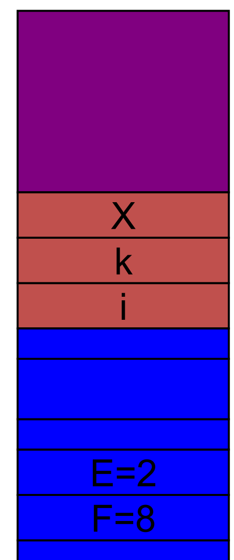
- Az F(A1, ... ,An) függvényművelet végrehajtása sorrendben a következő tevékenységeket jelenti

1.) Memória helyfoglalás a függvényblokk paraméterei és lokális változói számára.




```
int A(int X)
{
  int i, k;
  ...
  i=3;
  k=A(X-1);
  X=5;
  ...
}
```

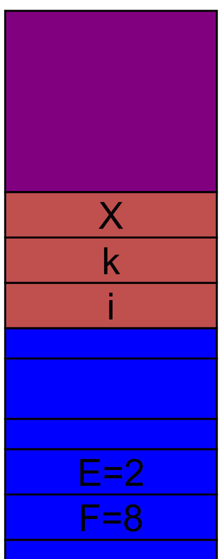
```
int E,F;

int main()
{
  ...
  E=A(E+F);
  ...
}
```



Végrehajtás

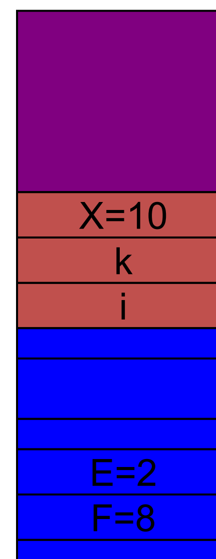
2.) Paraméterátadás.

– Először tetszőleges sorrendben kiértékelődnek az aktuális paraméterek. Mivel mind értékparaméter, az i-edik kiértékelt aktuális paraméter értéke átadódik az i-edik formális paraméternek, vagyis az aktuális paraméter értéke bemásolódik a formális paraméter számára foglalt memóriahelyre.



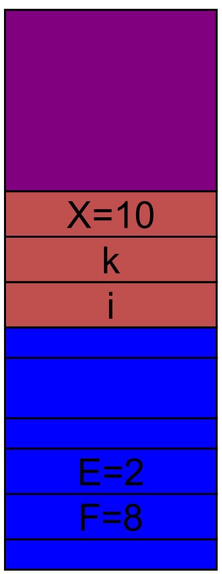

```
int A(int X)
{
  int i, k;
  ...
  i=3;
  k=A(X-1);
  X=5;
  ...
}
```

```
int E,F;

int main()
{
   ...
   E=A(E+F);
   ...
}
```



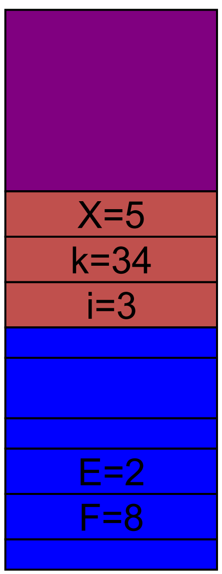
Végrehajtás

3.) A függvényblokk utasításrészének végrehajtása.



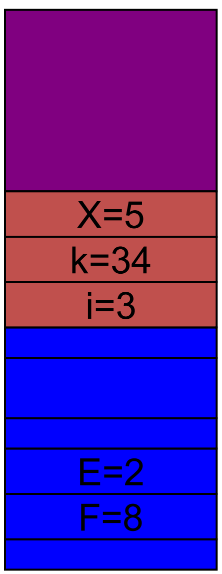

```
int A(int X)
{
  int i, k;
  ...
  i=3;
  k=A(X-1);
  X=5;
  ...
}
```

```
int E,F;

int main()
{
  ...
  E=A(E+F);
  ...
}
```



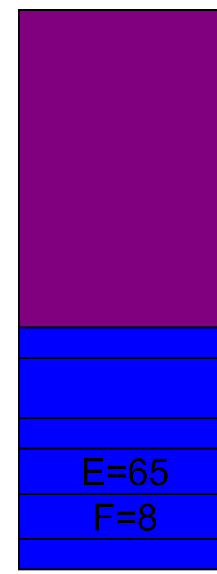
Végrehajtás

4.) A függvényblokk formális paraméterei és lokális változói számára foglalt memória felszabadítása.



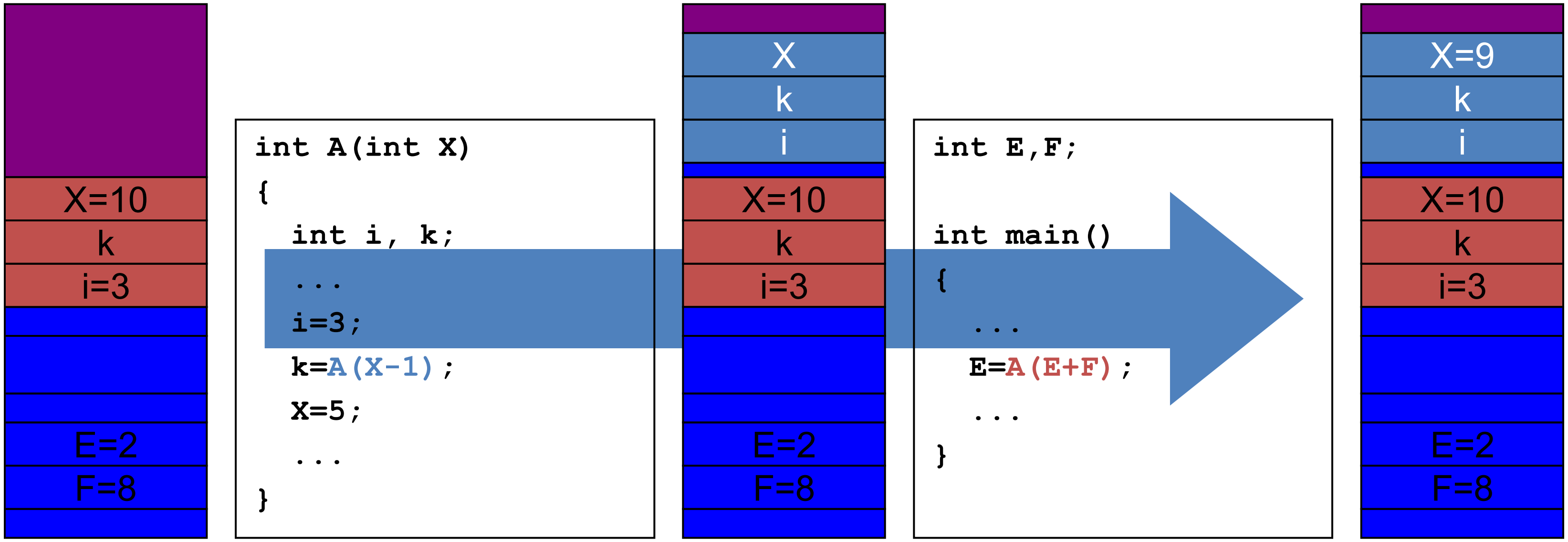

```
int A(int X)
{
  int i, k;
  ...
  i=3;
  k=A(X-1);
  X=5;
  ...
}
```

```
int E,F;

int main()
{
  ...
  E=A(E+F);
  ...
}
```



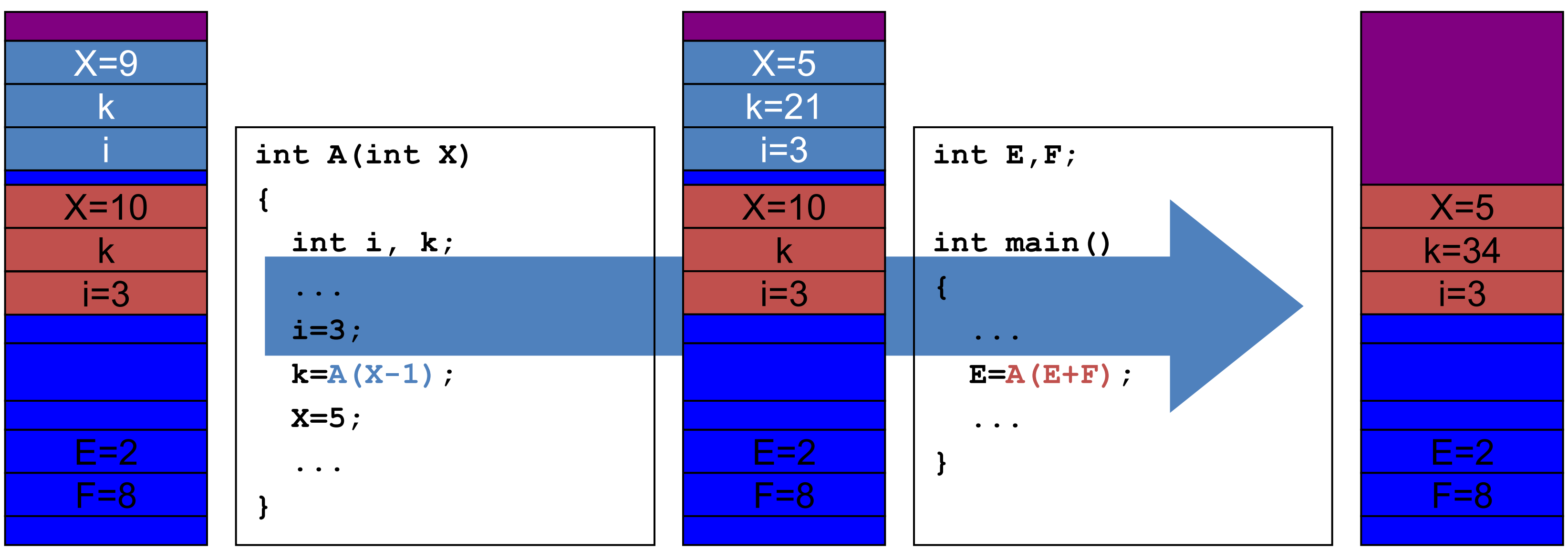
Végrehajtás (rekurzió)

R:1-2.) Rekurzió esetén (mint bármely függvényhívásnál) ugyanezek a lépések fognak végrehajtódni, tehát minden függvényhíváshoz saját változók tartoznak.



Végrehajtás (rekurzió)

R:3-4.) A rekurzióból (mint bármely függvényhívásból) visszatérve az elhagyott függvény paraméterei és lokális változói szűnnek meg, majd a hívó függvény folytatódik.



FOLYAMATÁBRA, LEÍRÓNYELV

Algoritmusok tervezése, szemléltetése

Az algoritmusok tervezésére, szemléltetésé-re sokféle eszköz létezik, pl.

- *folyamatábra:* az algoritmus szerkezetét, a lépések sorrendjét teszi áttekinthetővé;
- *leíró nyelv* (mondatszerű leírás): az így meg-fogalmazott algoritmus közvetlenül átírható egy általános célú programozási nyelvre.

Folyamatábra

- Szerkezeti ábrával az algoritmusok tervezése során a lépésenkénti finomításokat és a kifejlesztett algoritmust egyaránt kifejezhetjük.
- Ha csak a kész algoritmus vezérlési szerkezetét akarjuk leírni, akkor használhatjuk a folyamatábrát, mint leíró nyelvezetet.
- Ez olyan ábra, amelyben az algoritmus egyes lépéseit és a lépések közötti vezérlési viszonyt szemléltethetjük irányított vonalakkal.

Folyamatábra

- Legyen M={M_1, ... ,M_k} műveletek és F={F_1, ... ,F_l} feltételek.
- Az (M,F) feletti folyamatábrán olyan irányított gráfot értünk, amelyre teljesül a következő 5 feltétel:
  1.) Egy olyan pontja van, amely a Start üres művelettel van címkézve és ebbe a pontba nem vezet él.
  2.) Egy olyan pontja van, amely a Stop üres művelettel van címkézve és ebből a pontból nem indul él.

Folyamatábra

3.) Minden pontja M-beli művelettel vagy F-beli feltétellel van címkézve, kivéve a Start és a Stop pontokat.

4.) Ha egy pont

- M-beli művelettel van címkézve, akkor belőle egy él indul ki
- F-beli feltétellel van címkézve, akkor belőle két él indul ki és ezek az i(igen) illetve n(nem) címkét viselik

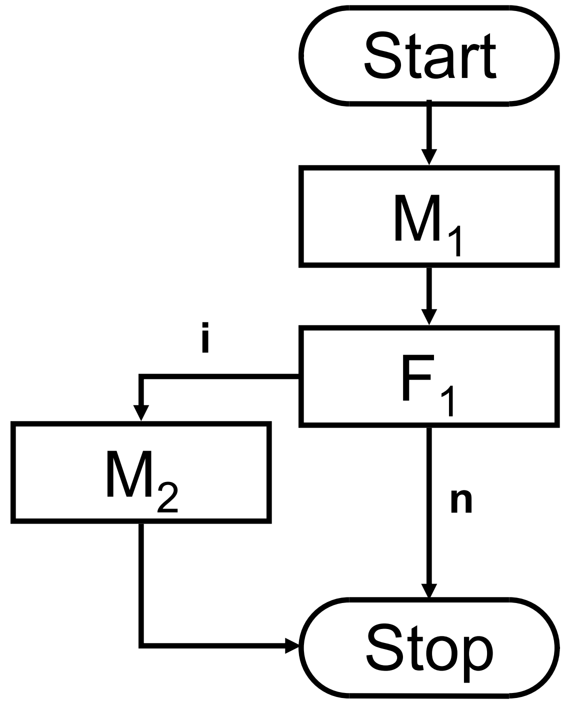
5.) A gráf miden pontja elérhető a Start címkéjű pontból.

Folyamatábra

- Egy folyamatábra a következő összetett vezérlési előírást jelenti
  - i.) A végrehajtás a Start pontból indul. Az összetett művelet végrehajtása akkor ér véget, ha a Stop pont kap vezérlést.
  - ii.) A gráf egy pontjának végrehajtása azt jelenti, hogy
    - Ha a pontban M-beli művelet van, akkor a művelet végrehajtódik és a vezérlés a gráf azon pontjára kerül, amelybe a pontból kiinduló él vezet
    - Ha a pont F-beli feltétellel van címkézve, akkor kiértékelődik a feltétel. Ha értéke igaz, akkor az a pont kap vezérlést, amelybe az i(igen) címkéjű él vezet, egyébként az a pont kap vezérlést, amelybe az n(nem) címkéjű él vezet.

Folyamatábra

- Példa
  - $M=\{M_1, M_2\}$
  - $F=\{F_1\}$



Vezérlési szerkezetek

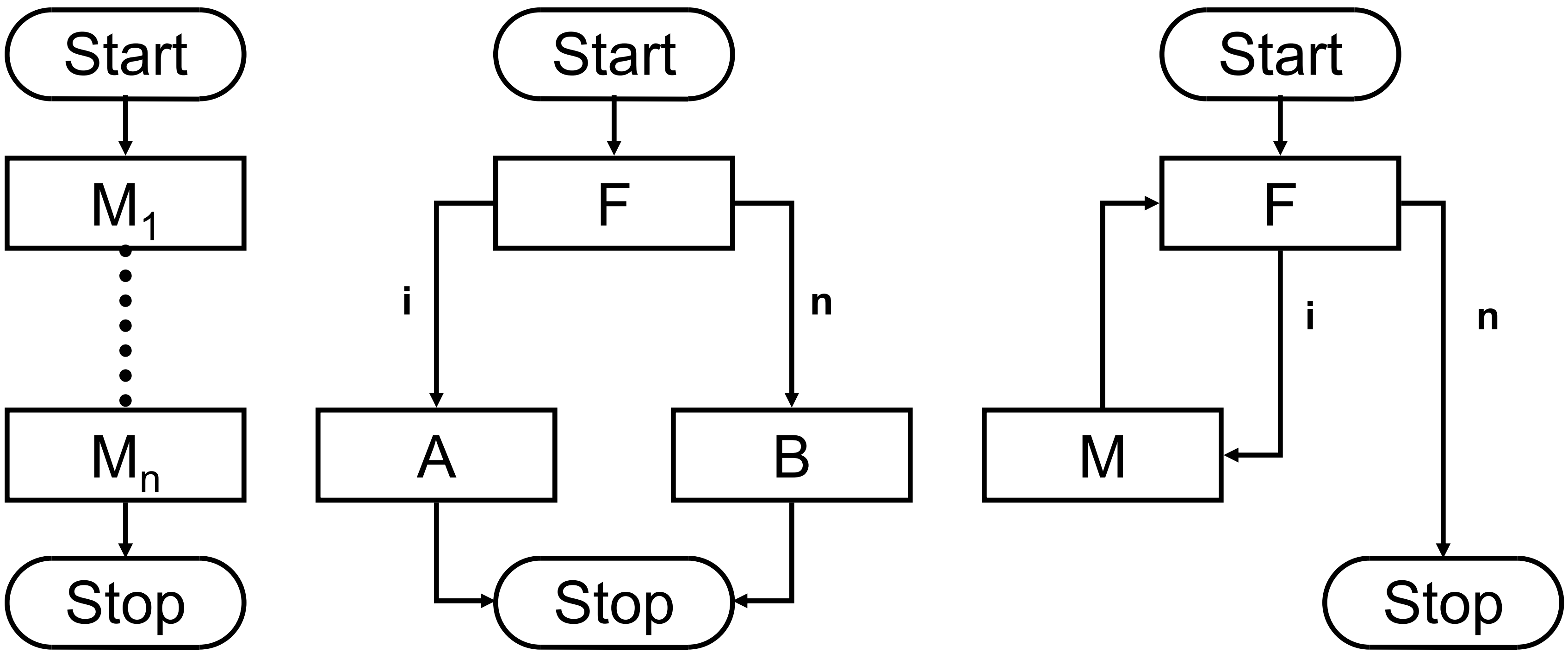
- Az 1960-as években bebizonyították (Dijkstra, strukturált programozás), hogy bármely algoritmus leírható szekvencia, szelekció és iteráció segítségével.
  - szekvencia: utasítások egymás utáni végrehajtási sorrendje, külön utasítást nem használunk a jelölé-sére;
  - szelekció: egy feltétel igaz vagy hamis voltától függ, hogy bizonyos utasítások végrehajtódnak-e vagy sem;
  - iteráció: lehetővé teszi meghatározott utasítások tet-szőleges számú ismételt végrehajtását.

Folyamatábra és struktúradiagram

- Nevezzük szabályosnak az eddig bevezetett vezérlési módokat, ezen belül alapvetőnek a szekvenciális, egyszerű szelekciós és a kezdőfeltételes ismétléses vezérlést.
- Beláttuk, hogy a szabályos vezérlési módok megvalósíthatók az alapvető vezérlési módokkal.

Folyamatábra és struktúradiagram

- Az alapvető vezérlési módokat használó algoritmusok kifejezhetők folyamatábrával is.



Folyamatábra és struktúradiagram

- Megmutatjuk, hogy fordítva is igaz, tehát a folyamatábrával leírt algoritmusok megadhatók szabályos és így alapvető vezérlési módokat használva, feltéve, hogy bevezethetünk egész típusú segédváltozót.

Folyamatábra és struktúradiagram

- Legyen G egy folyamatábra (M,F) felett, amely pontjainak száma n.
- Sorszámozzuk meg a gráf pontjait úgy, hogy a Start pont sorszáma 1 és a Stop pont sorszáma n legyen.
- Vegyünk fel egy olyan új egész típusú változót. Legyen ennek az azonosítója **`pont`**.

Folyamatábra és struktúradiagram

- Tekintsük az alábbi C programot

```
{
    pont = 1;
    while (pont != n) {
        switch (pont) {
        case 1  : U1; break;
        ...
        case n-1: Un-1; break;
        case n  : /* Stop */ break;
        } /* switch */
    }
}
```

Folyamatábra és struktúradiagram

- Ahol az U_i utasítás:
  - Ha az i. pontban az M_i művelet volt és a belőle kiinduló él a j. pontba vezetett:

    ```
    { Mi; pont = j; }
    ```
  - Ha az i. pontban az F_i feltétel volt és az igennel címkézett él a j., a nemmel címkézett él pedig a k. pontba vezetett

    ```
    { if (Fi) { pont = j; } else { pont = k; } }
    ```
- Az így megalkotott program a G folyamatábrával adott algoritmussal ekvivalens algoritmus kódolása lesz.

Egy általános leírónyelv szintaxisa

- *Változódeklarálás:*

  `változó` *`változónév1, változónév2...: típus`*

pl:

```
változó Darabszám: egész
változó Összeg, Átlag: valós
```

- *Értékadó utasítás:*

  `változó``:=`*`kifejezés`*

pl:

```
Átlag:=Összeg/Darabszám
```

Beolvasó és kiíró utasítások

`be`: *`vált1`*, *`vált2`*...

`ki`: *`kif1`*, *`kif2`*...

pl:

`be`: `Szám`

`ki`: `Szám*Szám`

Szelekciós (feltételes) utasítások

```
ha feltétel akkor
   utasítás...
hvége
```

```
ha feltétel akkor
   utasítás...
különben
   utasítás...
hvége
```

pl:
```
ha a>b akkor
   c:=a
különben
   c:=b
hvége
```

Többszörös elágazás

```
elágazás
  amikor feltétel1:
    utasítás…
  amikor feltétel2:
    utasítás…
    ...
  különben
    utasítás
evége
```

Iterációs (ciklus) utasítások

1. elöltesztelő feltételes ciklus (while)

```
amíg feltétel ismétel
  utasítás...
avége
```

pl:

```
be: R
amíg R<>0 ismétel
  ki: R*R*pi
  be: R
avége
```

Iterációs (ciklus) utasítások

2. elöltesztelő előírt lépésszámú (léptető, növekményes) ciklus (for)

```
ciklus cv:=ké..vé lépésköz: lk ismétel
  utasítás...
cvége
```

A használt rövidítések: ciklusváltozó, kezdőérték, végérték, lépésköz (lépésköz: 1 elhagyható).

pl:

```
ciklus I:=1..100 ismétel
  ki: I*I
cvége
```

Iterációs (ciklus) utasítások

3. hátultesztelő feltételes ciklus

ismétel
utasítás...
ivége *feltétel* **esetén**

pl:

ismétel
be: Jegy
ivége (Jegy>=1) és (Jegy<=5) **esetén**

RENDEZŐ ALGORITMUSOK

Rendezések

- A rendezési feladatok általános problémája: adott egy *n* elemű sorozat, készítsük el ezeknek az elemeknek egy olyan permutációját, amelyre igaz, hogy a sorozat *i.* eleme kisebb (egyenlő) az *i+1*-ediktől.
- Ez a növekvő rendezés. Az ilyen algoritmusok értelemszerűen átalakíthatók csökkenő sorrendűvé.

- A különböző rendezések leginkább a következő két művelet alkalmazásában térnek el egymástól:
  - összehasonlítás – két sorozatelem viszonyának meghatározása,
  - mozgatás – az elemek sorrendjének változtatá-sa a rendezettségnek megfelelően.
- A hatékonyság vizsgálatakor is ez a két művelet, valamint az igényelt tárterület érdekes számunkra.

Minimumkiválasztásos-rendezés

- Először megkeressük az egész tömb legkisebb elemét, és ezt kicseréljük az első elemmel.
- Most már csak a maradék: 2..N részt kell rendeznünk. Megkeressük itt is a legkisebb elemet, majd ezt kicseréljük a másodikkal.
- Folytassuk ezt a procedúrát. Utolsó lépésként az utolsó előtti helyre kell kiválasztanunk a legkisebb elemet, hiszen ezzel az utolsó is a helyére kerül.

```
algoritmus Minimumkiválasztásos_rendezés
 konstans N=maximális_elemszám
 változó A:tömb[1..N] Elemtípus
 változó I, J, MinIndex: egész
 változó Csere: Elemtípus

 ciklus I:=1..N-1 ismétel
   MinIndex:=I
   ciklus J:=I+1..N ismétel
     ha A[J]<A[MinIndex] akkor
       MinIndex:=J
     hvége
   cvége
   ha MinIndex<>I akkor
     Csere:=A[I]; A[I]:=A[MinIndex]; A[MinIndex]:=Csere;
   hvége
 cvége
algoritmus vége
```

Minimumkiválasztásos-rendezés – hatékonyság

- Helyfoglalás: n+1 (n elemű tömb és a cserékhez szükséges segédváltozó)
- Összehasonlítások száma: $n*(n-1)/2$
  nagyságrendben n^2: $O(n^2)$
- Cserék száma: 0..n-1
  értékadások száma: $0..3*(n-1)$

Nagy ordó – $O(f(n))$

- Az $O(f(n))$ jelölést (*nagy ordó*) rendszerint pozitív egész n-eken értelmezett f függvény esetén használjuk.
- Az $O(f(n))$ jelölés az n-től függő mennyiségek becslésére szolgál.
- Egy X mennyiség helyére akkor írhatunk $O(f(n))$-t, ha létezik olyan konstans, mondjuk $\mathcal{M}$, hogy minden elég nagy n-re, $|X| \leq \mathcal{M} \cdot |f(n)|$, azaz aszimptotikusan felső becslést adunk egy konstansszorzótól eltekintve a lépésszámra.

- A **ha** `Min<>I` **akkor** elágazást kihagyhatjuk az algoritmusból. Ezt akkor célszerű megtenni, ha az elemek kevesebb, mint a negyede van eleve a helyén.
- A minimumkiválasztásos rendezés már egy javításának tekinthető az egyszerű cserés rende-zésnek, mely az egyik legkevésbé hatékony rendezés, túl sok felesleges cserét tartalmaz:

```
ciklus I:=1..N-1 ismétel
  ciklus J:=I+1..N ismétel
    ha A[J]<A[I] akkor
      Csere:=A[I]; A[I]:=A[J]; A[J]:=Csere
    hvége
  cvége
cvége
```

Buborékrendezés

- Az előzőhöz képest a különbség az össze-hasonlításokban van.
- Ennél mindig két szomszédos elemet hasonlítunk össze, és ha nem megfelelő a sorrendjük, megcseréljük őket.

Buborékrendezés algoritmusa

```
algoritmus Buborékrendezés
 változó I, J: egész
 változó Csere: Elemtípus

 ciklus I:=N..2 lépésköz -1 ismétel
   ciklus J:=1..I-1 ismétel
     ha A[J]>A[J+1] akkor
       Csere:=A[J]; A[J]:=A[J+1]; A[J+1]:=Csere
     hvége
   cvége
 cvége
algoritmus vége
```

Buborékrendezés – hatékonyság

- Helyfoglalás: n+1 (n elemű tömb és a cserékhez szükséges segédváltozó)
- Összehasonlítások száma: n*(n-1)/2 nagyságrendben n^2: $O(n^2)$
- Cserék száma: 0..n*(n-1)/2; értékadások száma: 0..3*n*(n-1)/2

Cserék átlagos számának meghatározása

- A cserék száma megegyezik az *A* tömb elemei között fennálló inverziók számával.
- Valóban, minden csere pontosan egy inverziót szüntet meg a két szomszédos elem között, újat viszont nem hoz létre. A rendezett tömbbben pedig nincs inverzió.
- Feltesszük, hogy a rendezendő számok minden permutációja egyformán valószínű (vehetjük úgy, hogy az 1, 2, ..., *n* számokat kell rendeznünk).
- A cserék számának átlagát nyilvánvalóan úgy kapjuk, hogy az 1, 2, ..., *n* elemek minden permutációjának inverziószámát összeadjuk és osztjuk a permutációk számával (n!).

- Célszerű párosítani a permutációkat úgy, hogy mindegyikkel párba állítjuk az inverzét, pl. az 1423 és a 3241 alkot egy ilyen párt.
- Egy ilyen párban az inverziók száma együtt éppen a lehetséges $\binom{n}{2}$-t teszi ki:
  - pl. *inv*(1423) + *inv*(3241) = 2+4 = 6.
- Tehát a keresett hányados:

$$(n!/2)\binom{n}{2}/n! = \binom{n}{2}/2 = n(n-1)/4 = O(n^2)$$

- Tehát a buborékrendezés nagyságrendben n^2-es algoritmus.
- Szerencsétlen esetben tehát ez a módszer rosszabb a minimumkiválasztásosnál.

Javított buborékrendezés I.

- Ötlet: ha egy teljes belső ciklus lefutása alatt egyetlen csere sem volt, akkor az ez utáni menetekben sem lehet, tehát a sorozat már rendezetté vált.
- Ezzel kiküszöböltük a már feleslegessé vált összehasonlításokat.

Javított buborékrendezés I.

```
algoritmus JavítottBuborékrendezés1
 változó I, J: egész
 változó Csere: Elemtípus
 változó Vége: logikai

 I:=N; Vége:=hamis
 amíg (I>=2) és (nem Vége) ismétel
   Vége:=igaz;
   ciklus J:=1..I-1 ismétel
     ha A[J]>A[J+1] akkor
       Csere:=A[J]; A[J]:=A[J+1]; A[J+1]:=Csere
       Vége:=hamis
     hvége
   cvége
   I:=I-1
 avége
algoritmus vége
```

Javított buborékrendezés – hatékonyság

- Helyfoglalás: n+1 (n elemű tömb és a cserékhez szükséges segédváltozó)
- Összehasonlítások száma: n-1..n*(n-1)/2
- Értékadások száma: 0..3*n*(n-1)/2

Javított buborékrendezés II.

- Ötlet: ha a belső ciklusban volt csere, de a legutolsó valahol a sorozat belsejében volt, akkor azon túl már rendezett a sorozat.
- Jegyezzük meg az utolsó csere helyét, és legközelebb csak addig rendezzünk.
- Ez a megoldás tartalmazza az előző javítást is, ha nem volt csere, akkor befejeződik.

Javított buborékrendezés II.

```
algoritmus JavítottBuborékrendezés2
 változó I, J, UtolsóCsere: egész
 változó Csere: Elemtípus
  I:=N;
 amíg I>=2 ismétel
   UtolsóCsere=0;
   ciklus J:=1..I-1 ismétel
     ha A[J]>A[J+1] akkor
       Csere:=A[J]; A[J]:=A[J+1]; A[J+1]:=Csere
       UtolsóCsere:=J
     hvége
   cvége
   I:=UtolsóCsere
 avége
algoritmus vége
```

A konkrét számokat tekintve a hatékonysági mutatók ugyanazok, mint az előzőnél, azonban az előző javításhoz képest az átlagos végrehajtási idő tovább csökkenhet.

Beszúró rendezés

- Más néven beillesztő vagy kártyás rendezés.
- A működést leginkább a kártyalapok egyenként való kézbe vételéhez és a helyükre igazításához hasonlíthatjuk.
- Vesszük a soron következő elemet, és megkeressük a helyét a tőle balra lévő, már rendezett részben.
- A kereséssel párhuzamosan a nagyobb elemeket rendre eggyel jobbra mozgatjuk.
- Az aktuális elemet egy segédváltozóban tároljuk, mert a mozgatások során értéke felülíródhat egy nagyobb elemmel.

```
algoritmus Beszúrórendezés
 változó I, J: egész
 változó X: Elemtípus

 ciklus I:=2..N ismétel
   J:=I-1; X:=A[I]
   amíg (J>=1) és (X<A[J]) ismétel
     A[J+1]:=A[J]
     J:=J-1
   avége
   A[J+1]:=X
 cvége
algoritmus vége
```

- Hatékonyság:
  - Helyfoglalás: n+1 (n elemű tömb és a cserékhez szük-séges segédváltozó)
  - Összehasonlítások száma: n-1..n*(n-1)/2
  - Értékadások száma: 2*(n-1).. 2*(n-1)+n*(n-1)/2

Beszúró rendezés - előnyök

- Hatékony kis adatsorok esetén
- Hatékony, ha az adatsorok már részben rendezettek
- Gyakorlatban hatékonyabb a többi $O(n^2)$-es rendezésnél
- Online algoritmus, képes rendezni egy listát az új elemek felvételekor

Shell rendezés

- Nem önálló módszer, hanem több, már megismert módszerhez illeszthető.
- Donald Shell, 1959
- Elve: sokat javíthat a rendezésen, ha először az egymástól nagy távolságra lévő elemeket hasonlítjuk, cseréljük, mert így az egyes elemek gyorsabban közel kerülhetnek a végleges helyükhöz.
- Így az eredeti módszer hatékonyabbá válhat.
- Különösen igaz ez a beszúró rendezésnél.

Shellsort

Original	32	95	16	82	24	66	35	19	75	54	40	43	93	68	
After 5-sort	32	35	16	68	24	40	43	19	75	54	66	95	93	82	6 swaps
After 3-sort	32	19	16	43	24	40	54	35	75	68	66	95	93	82	5 swaps
After 1-sort	16	19	24	32	35	40	43	54	66	68	72	82	93	95	15 swaps

- Az elemek közötti távolságot jelöljük D-vel.
- Első értéke: N/3+1, majd D:=D/3+1.
- Pl. 10 elemű sorozatnál az alábbi részsorozatokat rendezzük:
  - D=4 1,5,9 2,6,10 3,7 4,8
  - D=2 1,3,5,7,9 2,4,6,8,10
  - D=1 1,2,3,4,5,6,7,8,9,10
- Úgy tűnhet, mintha D=1 esetén lezajlana az egész beszúró rendezés, ekkor azonban már a korábbi menetek miatt minimális számú össze-hasonlítás és mozgatás történik.

```
algoritmus ShellBeszúrórendezés
 változó I, J, D, E: egész
 változó X: Elemtípus
 D:=N
 ismétel
   D:=D/3+1
   ciklus E:=1..D ismétel
     I:=E+D
     amíg I<=N ismétel
       J:=I-D; X:=A[I]
       amíg (J>=1) és (X<A[J]) ismétel
         A[J+D]:=A[J]
         J:=J-D
       avége
       A[J+D]:=X
       I:=I+D
     avége
   cvége
 ivége D=1 esetén
algoritmus vége
```

Shell rendezés hatékonysága

- Kevesebb, mint $O(n^2)$ összehasonlítást és cserét igényel
- Nehéz megállapítani a műveletigényét:
  - Megvalósítástól függően $O(n^{1.25})$ és $O(n^{1.5})$ között van

Indexvektoros rendezés

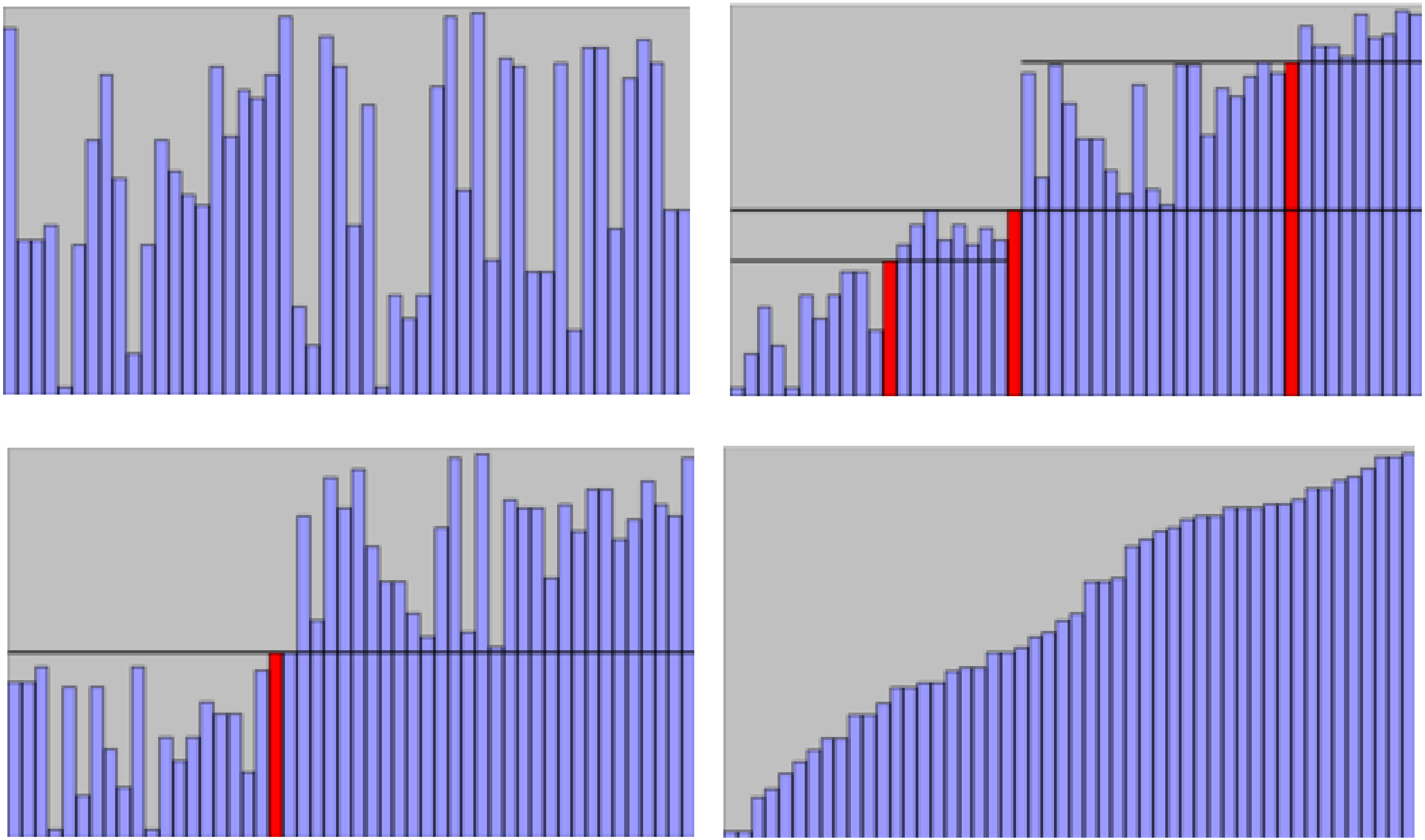
- A rendezendő tömb elemeit nem mozgatjuk, hanem a tömbhöz egy indexvektort rendelünk, melyben a tömb elemeire mutató indexek a tömb rendezettségének megfelelően követik egymást.
- Az eljárás során az indextömb elemeit az indexelt adatok rendezettségének függvényében sorba rakjuk.
- A hasonlítást tehát mindig az indexelt adatokra végezzük el, de csere esetén csak az indexeket cseréljük.
- Így az eredeti tömbünk változatlan maradhat.
- Közvetett módon így egyszerre több szempont szerint is módunkban áll adatainkat rendezni.

```
algoritmus Indexvektoros_Minimumkiválasztásos_rendezés
 konstans N=maximális_elemszám
 változó A:tömb[1..N] Elemtípus
 változó Index:tömb[1..N] egész
 változó I, J, Min, Csere: egész
 ciklus I:=1..N ismétel
   Index[I]:=I;
 cvége
 ciklus I:=1..N-1 ismétel
   Min:=I
   ciklus J:=I+1..N ismétel
     ha A[Index[J]]<A[Index[Min]] akkor
       Min:=J
     hvége
   cvége
   ha Min<>I akkor
     Csere:=Index[I]; Index[I]:=Index[Min]; Index[Min]:=Csere;
   hvége
 cvége
algoritmus vége
```

Gyorsrendezés (Quicksort)

- Elve rekurzív:
  - osszuk két részre a rendezendő sorozatot úgy, hogy az egyik rész minden eleme kisebb legyen a másik rész összes eleménél;
  - a két részre külön-külön ismételjük meg az előbbi lépést, míg mindkét rész 0 vagy 1 elemű lesz.
- A feldolgozási művelet egy szétválogatás, melynek során egy megadott értékhez viszonyítva a tőle kisebb elemeket elé, a nagyobbakat mögé helyezzük.
- Viszonyítási értéknek a gyakorlatban a sorozat középső vagy első elemét választják ki.
- Hatékonyság: O(n log n)

Quicksort



```
eljárás GyorsRendezés(Alsó, Felső: egész)
 változó I, J: egész
 változó: X, Csere: ElemTípus
 I:=Alsó; J:=Felső
 X:=A[(Felső+Alsó)/2]
 ismétel
   amíg A[I]<X ismétel
     I:=I+1
   avége
   amíg A[J]>X ismétel
     J:=J-1
   avége
   ha I<J akkor
     Csere:=A[I]; A[I]:=A[J]; A[J]:=Csere
   hvége
   ha I<=J akkor
     I:=I+1; J:=J-1
   hvége
 ivége I>J esetén
 ha Alsó<J akkor GyorsRendezés(Alsó, J) hvége
 ha I<Felső akkor GyorsRendezés(I, Felső) hvége
eljárás vége
```

Felhasznált anyagok

- Dévényi Károly (SZTE): Programozás alapjai
- Simon Gyula (PE): A programozás alapjai
- Pohl László (BME): A programozás alapjai
- B. W. Kernighan - D. M. Ritchie: A C programozási nyelv