Alapvető algoritmusok

6. előadás

Dr. Pătcaș Csaba

BABEŞ-BOLYAI TUDOMÁNYEGYETEM
Matematika és Informatika Kar

Tartalom

Harmadik változat

Ötlet

- Észrevesszük, hogy a sorozat első bejárása után, a legnagyobb elem a helyére kerül, a következő alkalommal a második legnagyobb elem is a helyére kerül és így tovább.
- Tulajdonképpen ez a tulajdonság adja a rendezés nevét, mert a nagy elemek buborékként „szállnak fel" a tömb végére.
- Emiatt nem szükséges mindig a tömb végéig ellenőrizni a felcserélendő elemeket, hanem mindig csökkenthetjük eggyel az értéket ameddig az `i` ciklusváltozó nő, ezt `nn`-el jelöljük.

Buborékrendezés (Bubblesort)

Harmadik változat

```
ALGORITMUS Buborékrendezés3(n, a)
  nn = n - 1
  ISMÉTELD
    rendben = IGAZ
    MINDEN i = 1, nn végezd el:
      HA (a[i] > a[i + 1]) akkor
        Felcserél(a[i], a[i + 1])
        rendben = HAMIS
      VÉGE(Ha)
    VÉGE(Minden)
    nn = nn - 1
  AMEDDIG (rendben)
VÉGE(Algoritmus)
```

Harmadik változat

Példa, bonyolultság

Példa: [9 2 7 3 5]

Harmadik változat

Példa, bonyolultság

Példa: [9 2 7 3 5]

- Legrosszabb eset, ha a legkisebb elem a sorozat végén van, ekkor a bonyolultság: $\Theta(n^2)$
- Az átlag eset szintén négyzetes.
- A legjobb eset, ha a sorozat rendezett, ekkor a bonyolultság: $\Theta(n)$
- Stabil és helyben rendez.

Negyedik változat

Ötlet

- Láttuk, hogy ha a sorozat végén lévő elemek már a helyükön vannak, azokat nem kell többet ellenőrizni.
- Innen jön az ötlet, hogy a sorozatot csak az utolsó csere helyéig vizsgáljuk, ezt a `k` változóban jegyezzük meg.

Buborékrendezés (Bubblesort)

Negyedik változat

```
ALGORITMUS Buborékrendezés4(n, a)
  k = n
  ISMÉTELD
    nn = k - 1
    rendben = IGAZ
    MINDEN i = 1, nn végezd el:
      HA (a[i] > a[i + 1]) akkor
        Felcserél(a[i], a[i + 1])
        rendben = HAMIS
        k = i
      VÉGE(Ha)
    VÉGE(Minden)
  AMEDDIG (rendben)
VÉGE(Algoritmus)
```

Negyedik változat

Példa, bonyolultság

Példa magyar néptánccal szemléltetve: `https://youtu.be/lyZQPjUT5B4`

- Legrosszabb eset $\Theta(n^2)$, átlag eset $\Theta(n^2)$, legjobb eset $\Theta(n)$.
- Helyben rendez, stabil.

Ötödik változat (cocktail shaker sort)

Ötlet

- Elegendő a sorozatot csak az első és az utolsó csere helye között vizsgálni.
- Mivel eddig mindig balról jobbra jártuk be a tömböt, mindig a legnagyobb elemek kerültek a helyükre.
- Ha viszont jobbról balra is bejárnánk, akkor a legkisebb elemek is hamar a helyükre kerülnének.
- A `régibal` és `régijobb` változók tárolják az első az utolsó cserék helyét.
- A `bal` és `jobb` változókba mentjük ezeknek az újonnan megtalált értékeit.

Koktélrendezés

```
ALGORITMUS Buborékrendezés5(n, a)
  régibal = 1
  régijobb = n - 1
  ISMÉTELD
    rendben = IGAZ
    jobb = 0
    MINDEN i = régibal, régijobb végezd el:
      HA (a[i] > a[i + 1]) akkor
        Felcserél(a[i], a[i + 1])
        rendben = HAMIS
        jobb = max(jobb, i)
      VÉGE(Ha)
    VÉGE(Minden)
```

Koktélrendezés

```
HA (NEM rendben) akkor
  régijobb = jobb
  rendben = IGAZ
  bal = n
```

Koktélrendezés

```
      MINDEN i = régijobb, régibal, -1 végezd el:
        HA (a[i] > a[i + 1]) akkor
          Felcserél(a[i], a[i + 1])
          rendben = HAMIS
          bal = min(bal, i)
        VÉGE(Ha)
      VÉGE(Minden)
      régibal = bal
    VÉGE(Ha)
  AMEDDIG (rendben)
VÉGE(Algoritmus)
```

Megjegyzés: Tulajdonképpen felesleges a minimum- és maximumszámolás, lehetne egyszerű értékadás is: `jobb = i`, illetve `bal = i`. Ekkor hiányozhatna a `bal` és `jobb` változók inicializálása is.

Ötödik változat

Példa, bonyolultság

Példa: [5 2 3 4 1]

Ötödik változat

Példa, bonyolultság

Példa: [5 2 3 4 1]

- Itt már nem elég a legrosszabb esethez ha a legkisebb elem a sorozat végén van, a sorozat ellenkezően rendezett kell legyen, ekkor a bonyolultság $\Theta(n^2)$ marad.
- Átlag eset $\Theta(n^2)$, legjobb eset $\Theta(n)$.
- Helyben rendez, stabil.
- Általában a buborékrendezés változatai abban az esetben is jól működnek, ha a sorozat „majdnem rendezett”.
- Specifikusan a koktélrendezés k-szor járja be a sorozatot, ha minden elem legtöbb k pozicióra van a helyétől.

Egyszerű felcseréléses rendezés

- Hasonlít a buborékrendezéshez.
- A különbség, hogy kötelezően elvégzi az összehasonlítást minden indexpárra.
- Ha a sorrend nem megfelelő, ugyanúgy cserél.
- Az első bejárás után helyére kerül a legkisebb elem, majd a második legkisebb és így tovább.
- A belső ciklus az aktuális elem utáni pozíciótól indul.

Egyszerű felcseréléses rendezés

```
ALGORITMUS FelcserélésesRendezés(n, a)
  MINDEN i = 1, n - 1 végezd el:
    MINDEN j = i + 1, n végezd el:
      HA (a[i] > a[j]) akkor
        Felcserél(a[i], a[j])
      VÉGE(Ha)
    VÉGE(Minden)
  VÉGE(Minden)
VÉGE(Algoritmus)
```

Egyszerű felcseréléses rendezés

Mennyi a bonyolultság a legjobb esetben?

Egyszerű felcseréléses rendezés

Példa, bonyolultság

Példa: [2 3 4 1]

- Az algoritmus minden esetben $\frac{n(n-1)}{2}$ összehasonlítást végez, tehát bonyolultsága $\Theta(n^2)$.
- Helyben dolgozik és nem stabil.

Számlálva szétosztó (válogatásos) rendezés

- Minden elemet összehasonlítunk a többi elemmel.
- Minden elemre megszámoljuk a `k` változóban, hogy hány nála kisebb elem van.
- Ezzel tulajdonléppen meghatároztuk mindegyik elem sorszámát a rendezett sorozatban.
- Ha a feladat követelményei szerint az eredeti sorozat kell tartalmazza a rendezett sorozatot, akkor az algoritmus végén szükséges rámásolni az eredeti tömbre a rendezett tömböt.

Számlálva szétosztó (válogatásos) rendezés

```
ALGORITMUS VálogatásosRendezés(n, a)
  MINDEN i = 1, n végezd el:
    k = 0
    MINDEN j = 1, n végezd el:
      HA (a[i] > a[j]) akkor
        k = k + 1
      VÉGE(Ha)
    VÉGE(Minden)
    rendezett[k + 1] = a[i]
  VÉGE(Minden)
  Másol(n, rendezett, n, a)
VÉGE(Algoritmus)
```

Számlálva szétosztó (válogatásos) rendezés

Helyben rendez-e a számlálva szétosztó rendezés? Indokold a választ!
Mondj példát egy olyan sorozatra, amelyre nem működik a bemutatott rendezés!

Számlálva szétosztó (válogatásos) rendezés

Példa, bonyolultság

Példa: [1 3 2 5 3]

- Mint látjuk a példából, a bemutatott változat nem működik egyenlő elemek esetén.
- Viszont könnyen módosítható úgy, hogy működjön. Hogyan?
- Minden esetben $\Theta(n^2)$ bonyolultságú.
- Nem dolgozik helyben, memóriabonyolultsága $\Theta(n)$.
- A megfelelő módosítással stabillá válik.

Minimum (maximum) kiválasztásra épülő rendezés (Selection sort)

- Megkeressük a sorozat legkisebb elemét.
- Ezt az első helyre tesszük úgy, hogy felcseréljük az első helyen található elemmel.
- Folytatjuk hasonlóan, de a következő lépésben már a második helytől kezdődően keressük a minimumot.
- Addig folytatjuk, míg a sorozat végére nem érünk.

Példa tánc formában: `https://youtu.be/Ns4TPTC8whw`

Minimum (maximum) kiválasztásra épülő rendezés (Selection sort)

```
ALGORITMUS MinimumKiválasztásosRendezés(n, a)
  MINDEN i = 1, n - 1 végezd el:
    ind_min = i
    MINDEN j = i + 1, n végezd el:
      HA (a[ind_min] > a[j]) akkor
        ind_min = j
      VÉGE(Ha)
    VÉGE(Minden)
    Felcserél(a[i], a[ind_min])
  VÉGE(Minden)
VÉGE(Algoritmus)
```

Minimum (maximum) kiválasztásra épülő rendezés (Selection sort)

Példa, bonyolultság

Példa: [2 8 9 2 1]

- Bonyolultsága minden esetben $\Theta(n^2)$.
- Helyben rendez.
- Amint a fenti példából látszik, nem stabil.

Beszúró rendezés (Insertion sort)

- Ahhoz hasonlóan dolgozik, mint ahogy kártyázáskor rendezzük a lapokat a kezünkben.
- Kezdve a második elemmel mindegyik elemnek megkeressük a helyét az eddig rendezett sorozatban és beszúrjuk oda.
- A beszúrás miatt az újonnan beszúrt elemtől jobbra eső elemeket egy pozícióval jobbra kell tolnunk, ezzel csinálunk helyet az új elemnek.

Példa tánc formában: `https://youtu.be/ROalU37913U`

Beszúró rendezés (Insertion sort)


```
ALGORITMUS BeszúróRendezés(n, a)
  MINDEN j = 2, n végezd el:
    segéd = a[j]
    i = j - 1
    AMÍG ((i > 0) ÉS (a[i] > segéd)) végezd el:
      a[i + 1] = a[i]
      i = i - 1
    VÉGE(Amíg)
    a[i + 1] = segéd
  VÉGE(Minden)
VÉGE(Algoritmus)
```

Beszúró rendezés (Insertion sort)

Példa, bonyolultság

Példa: [7 4 9 5 2]

- A legjobb esetben a tömb rendezett, ekkor a bonyolultság $\Theta(n)$.
- A legrosszabb eset amikor a sorozat fordított sorrendben van, ekkor a bonyulultság $\Theta(n^2)$.
- Átlag esetben szintén négyzetes: $\Theta(n^2)$
- Helyben rendez, stabil.
- Kis n értékekre (általában valahol 7 és 50 között) gyorsabb mint a jobb bonyolultsággal rendelkező rendezőalgoritmusok, mint a Gyorsrendezés vagy az Összefésüléses rendezés. Hogyan lehetséges ez?
- Hatékony majdnem rendezett bemenetekre, ha minden elem legtöbb k pozícióra van a végső helyétől, bonyolultsága $\Theta(kn)$.
- Alkalmas ún. online üzemmódban rendezni, ahol egyenként kapjuk meg az elemeket.

Buborékrendezés vs Minimumkiválasztásos rendezés vs Beszúró rendezés

- A gyakorlatban a beszúrásos rendezés gyorsabb, mint minimumkiválasztásos rendezés, amely gyorsabb, mint a buborékrendezés.
- Átlagosan a beszúrásos rendezés fele annyi összehasonlítást végez, mint a minimumkiválasztásos.
- Viszont a beszúrásos rendezés lényegesen többször ír a tömbbe a minimumkiválasztásos rendezéshez képest (négyzetes vs lineáris nagyságrend).
- Emiatt azokban az esetekben, amikor az írás lényegesen költségesebb, mint az olvasás, a minimumkiválasztásos rendezés a hatékonyabb.
- A minimumkiválasztásos és a beszúró rendezésnek is számos változata ismert.
- Bizonyos programozási nyelvekbe beépített rendezések több klasszikus rendezési algoritmust kombinálnak, ilyen az STL-ben megtalálható Introsort, vagy a Python-ban implementált Timsort.

Tartalom

Lineáris rendezések

- Az eddig tárgyalt rendezési algoritmusok két elem összehasonlítására alapultak.
- Láttuk, hogy ez $\Omega(n \log n)$ futási időt feltételez.
- Ahhoz, hogy ennél hatékonyabban rendezhessünk, nem az összehasonlítást kell használnunk, hanem a rendezendő sorozat bizonyos tulajdonságait.
- Ezeket a rendezési algoritmusokat csak az adott tulajdonsággal rendelkező sorozatokkal végezhetjük.

Leszámláló rendezés (Countsort)

- Akkor alkalmazhatjuk, ha a bemeneti kulcsok mindegyike 1 és k közötti egész szám.
- Használunk egy `darab` segédtömböt, melynek az i. eleme azt tartja nyilván, hogy hány darab i-vel egyenlő elemet találtunk az eredeti tömbben.
- A lineáris feldolgozás után felülírjuk az eredeti tömb elemeit a segédtömb elemeinek értékei alapján.

Leszámláló rendezés (Countsort)


```
ALGORITMUS LeszámlálóRendezés(n, a, k)
  MINDEN i = 1, k végezd el:
    darab[i] = 0
  VÉGE(Minden)
  MINDEN j = 1, n végezd el:
    darab[a[j]] = darab[a[j]] + 1
  VÉGE(Minden)
```

Leszámláló rendezés (Countsort)

```
  q = 0
  MINDEN i = 1, k végezd el:
    MINDEN j = 1, darab[i] végezd el:
      q = q + 1
      a[q] = i
    VÉGE(Minden)
  VÉGE(Minden)
VÉGE(Algoritmus)
```

Leszámláló rendezés (Countsort)

Példa, bonyolultság

Példa $= [1\ 6\ 1\ 2\ 3\ 2\ 7\ 2]$, $k = 10$

- Az időbonyolultság minden esetben $\Theta(n + k)$.
- Memóriabonyolultság $\Theta(k)$, tehát nem dolgozik helyben.
- Ebből következik, hogy k nem lehet túl nagy.
- Az algoritmus egyszerűen módosítható úgy, hogy más sorszámozható típusokra (pl. karakterek), vagy $[x, y]$ intervallumban lévő egész számokra is működjön.
- Általánosított változatai a pigeonhole sort és a ládarendezés (bucket sort vagy bin sort).

Leszámláló rendezés stabilan

- Az előző változatra nem tudjuk a megszokott módon értelmezni, hogy stabil rendezés-e, mivel az egyenlő elemeket nem tároljuk (csak számoljuk).
- Erre egy lehetséges megoldás, hogy a segédtömbben nem csak darabszámot tárolunk, hanem egymás után láncolva az elemeket is.
- Ez tulajdonképpen a pigeonhole sort ötlete, amely stabil.
- Viszont a leszámláló rendezést is módosíthatjuk erre a célra, anélkül, hogy szükség legyen dinamikus helyfoglalásra.
- A `b` segédtömbbe építjük fel a rendezett sorozatot, a végén ezt visszamásoljuk az `a`-ba.

Leszámláló rendezés stabilan

```
ALGORITMUS StabilLeszámlálóRendezés(n, a, k)
  MINDEN i = 1, k végezd el:
    darab[i] = 0
  VÉGE(Minden)
  MINDEN j = 1, n végezd el:
    darab[a[j]] = darab[a[j]] + 1
  VÉGE(Minden)
```

Leszámláló rendezés stabilan

```
  MINDEN i = 2, k végezd el:
    darab[i] = darab[i] + darab[i - 1]
  VÉGE(Minden)
  MINDEN j = n, 1, -1 végezd el:
    b[darab[a[j]]] = a[j]
    darab[a[j]] = darab[a[j]] - 1
  VÉGE(Minden)
  a = b
VÉGE(Algoritmus)
```

Megjegyzés: apró módosításokkal a fenti algoritmust lehetséges úgy is implementálni, hogy az utolsó `MINDEN` ciklusban növekvő sorrendben járjuk be az elemeket és a `darab` tömb elemeit növeljük.

Leszámláló rendezés stabilan

Példa, bonyolultság

Példa: [4 1 3 4 3], $k = 4$

- Az időbonyolultság minden esetben $\Theta(n + k)$ marad.
- Ennek a változatnak a memóriabonyolultsága $\Theta(n + k)$.

Számjegyes rendezés (Radixsort)

- Feltételezzük, hogy mind az n szám legtöbb d számjegyű lehet.
- Sorban dolgozzuk fel a számjegyeket jobbról balra, a legkevésbé fontos számjeggyel kezdve.
- Először rendezzük a számokat az utolsó számjegyük alapján, úgy, hogy ha csak ezt a számjegyet tekintjük, növekvő sorrendet lássunk.
- Folytatjuk az utolsó előtti számjeggyel és addig folytatjuk, ameddig a számokat mind a d számjegy szerint nem rendeztük.
- Fontos, hogy egy adott számjegy szerinti rendezés stabil legyen, hogy ne rontsuk el az addigi munkánkat.

Számjegyes rendezés (Radixsort)

Vázlat


```
ALGORITMUS SzámjegyesRendezés(n, a, d)
  MINDEN szj = 1, d végezd el:
    //Stabil leszámlálással rendezzük az "a" tömböt
    //a szj. legjelentéktelenebb számjegy szerint
    //Természetesen k = 10 lesz
  VÉGE(Minden)
VÉGE(Algoritmus)
```

Számjegyes rendezés (Radixsort)

Példa, bonyolultság

Példa: [329, 457, 657, 839, 436, 720, 355], $d = 3$

- d-szer vizsgáljuk meg a sorozatot, így az időbonyolultság $\Theta(dn)$.
- A memóriabonyolultság $\Theta(n)$.
- A számjegyes rendezést először lyukkártya-rendező berendezéseknél használták.

Tartalom

Mit nevezünk rekurziónak?

- A rekurzió egy programozási stílus, inkább technika, vagy implementálási mód, mint módszer.
- A rekurzív programozás, mint fogalom, a matematikai értelmezéshez közel álló módon került közhasználatba.
- Programozásban általában az alprogramok kontextusában használjuk a rekurzió fogalmát, de léteznek rekurzívan definiált adatszerkezetek is.

Definíció

Olyan alprogramokat nevezünk rekurzívnak, amelyek meghívják önmagukat.

Példák rekurzióra

1. Az aranyhal és a három kívánság: a harmadik kívánság az, hogy legyen még három kívánságom.
2. Ha a definíción belül felhasználjuk magát a definiálandó fogalmat, rekurzív definícióról beszélünk. Pl. a faktoriális rekurzív definíciója:

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & \text{egyébként} \end{cases}$$

3. Knuth így definiálta a bináris fákat: *Egy bináris fa vagy üres, vagy tartalmaz egy csomópontot, amelynek van egy bal meg egy jobb utódja, amelyek szintén bináris fák.*

Rekurzív és iteratív algoritmusok

- Bármely algoritmus megvalósítható iteratívan és rekurzívan is.
- Mindkét technikának a lényege: bizonyos utasítások ismételt végrehajtása.
- Az iteratív algoritmusokban az ismétlést ciklusokkal valósítjuk meg.
- A rekurzív algoritmusokban az ismétlés azáltal valósul meg, hogy az illető alprogram meghívja önmagát.

Rekurzív és iteratív algoritmusok

Mi szól a rekurzív algoritmusok mellett?

Általában tömörebbek és olvashatóbbak, bizonyos algoritmusokat természetesebb íly módon implementálni.

Mi szól a rekurzív algoritmusok ellen?

Igénybe veszik a végrehajtási vermet, a hívások által bevezetett implicit műveletek miatt a futási idejük is nagyobb. Bizonyos esetekben gondatlanságból is exponenciálissá válhat a bonyolultságuk (pl. Fibonacci).

Rekurzív és iteratív algoritmusok

Példa megállási feltétellel


```
AMÍG (i <= n) végezd el:
  \\utasítások
VÉGE(Amíg)
```

```
ALGORITMUS RekurzívWhile1(i, n)
  HA (i > n) akkor
    VISSZATÉRÍT \\return; vagy exit;
  VÉGE(Ha)
  \\utasítások
  RekurzívWhile1(i + 1, n)
VÉGE(Algoritmus)
```

Rekurzív és iteratív algoritmusok

Példa továbblépési feltétellel


```
AMÍG (i <= n) végezd el:
  \\utasítások
VÉGE(Amíg)
```

```
ALGORITMUS RekurzívWhile2(i, n)
  HA (i <= n) akkor
    \\utasítások
    RekurzívWhile2(i + 1, n)
  VÉGE(Ha)
VÉGE(Algoritmus)
```

Miért van szükségünk rekurzióra?

- Mert rekurzív gondolkodásmód nélkül bizonyos programozási nyelvekben nem tudunk dolgozni.
- Mert szükségünk van a rekurzív gondolkodásmódra:
  1. Az oszd meg és uralkodj (divide et impera) módszerrel megoldható feladatok esetén. Ezeket a gyakorlatban mindig rekurzívan implementáljuk.
  2. A dinamikus programozás módszerével megoldható feladatok esetén. Ezeket a gyakorlatban mindig iteratívan implementáljuk.
  3. A visszalépéses keresés (backtracking) módszerével megoldható feladatok esetén. Ezeket a gyakorlatban nagyon gyakran rekurzívan implementáljuk.
- Mert bizonyos algoritmusokat sokkal egyszerűbb rekurzívan implementálni. Ide tartoznak például a rekurzívan definiálható adatszerkezeteket (pl. gyökeres fák) feldolgozó algoritmusok is és számos más gráfalgoritmus.

Közvetlen és közvetett rekurzió

- Akkor beszélünk közvetlen (direkt) rekurzióról, amikor egy alprogram a saját utasításblokkjából hívja meg önmagát, erre láttunk példát az imént.
- Ha egy rekurzív alprogramot egy másik alprogram hív meg, amelyet ugyanakkor az illető alprogram hív (közvetve vagy közvetlenül), akkor közvetett (indirekt) rekurzióról beszélünk. Pl.

```
ALGORITMUS Első()
  Második()
VÉGE(Algoritmus)

ALGORITMUS Második()
  Első()
VÉGE(Algoritmus)
```

A megállási feltétel szükségessége

- Minden alprogramhíváskor a verembe kerülnek a híváskontextushoz tartozó adatok: az utasítás címe ahová vissza kell térni (`CS:IP` regiszterpár), a paraméterek és a lokális változók.
- Mivel a verem mérete véges, bizonyos számú hívás után bekövetkezhet a túlcsordulás. Ekkor a program `Stack overflow` hibaüzenettel leáll.
- Ennek gyakran oka a végtelen rekurzió, amikor az alprogramok rekurzív hívássorozata nem áll le soha.
- Ezt feltétlenül el kell kerülnünk a megfelelő megállási feltételek és/vagy továbblépési feltételek megállapításával és bevezetésével.
- Az újrahívások számát a rekurzió mélységének nevezzük.
- A túlcsordulás másik oka lehet, hogy a programozási környezetben a verem mérete túl kicsire van állítva.

Gyakorlati tanácsok a rekurzív gondolkodásmód kialakításához

Amikor rekurzív algoritmust kell terveznünk, próbáljunk válaszolni a következő kérdésekre:

1. Mik a paraméterek? Milyen értékek jellemeznek egy részfeladatot, állapotot? Mi az, ami változik egyik hívásról a másikra?
2. Mi a megállási feltétel? Melyik az a legkisebb részfeladat, amit már meg tudunk oldani újabb hívás nélkül?
3. Mi lesz a kezdeti hívás, aminek eredményeképpen megkapjuk a megoldást? Milyen paraméterek jellemzik a kezdeti feladatot?
4. Mik a rekurzív összefüggések? Hogyan függnek egymástól a részfeladatok?
5. Milyen lokális változókra van szükségünk az előző pontok megvalósításához?

Szavak sorrendjének megfordítása

Feladat

Olvassunk be szavakat a billentyűzetről, ameddig egy üres szó nem kerül beolvasásra. Ezután, írassuk ki a beolvasott szavakat fordított sorrendben, tömb használata nélkül!

1. Paraméterek?
2. Megállási feltétel?
3. Kezdeti hívás?
4. Rekurzív hívások?
5. Lokális változók?

Szavak sorrendjének megfordítása

Az alábbi megoldás didaktikai célt szolgál, ezért nem tartja be a beolvasás és kiíratás alprogramokba való leválasztását a megoldás többi részéről.

```
ALGORITMUS Fordít()
  BE: szó
  HA (szó NEM üres) akkor
    Fordít()
  VÉGE(Ha)
  KI: szó
VÉGE(Algoritmus)
```

Gyakorlati tanács: Használjuk a programozási környezet hibakeresési eszközeit a rekurzív algoritmusok működésének megértéséhez!