Alapvető algoritmusok

3. előadás

Dr. Pătcaș Csaba

BABEȘ-BOLYAI TUDOMÁNYEGYETEM
Matematika és Informatika Kar

Tartalom

Maximális összegű tömbszakasz

Feladat

Adott egy n egész számból álló számsorozat, amely biztosan tartalmaz legalább egy pozitív számot. Írjunk programot, amely meghatározza azt a tömbszakaszt, amelynek összege a lehető legnagyobb.

- Egy tömb részsorozatának nevezzük a tömb egy olyan rendezett részhalmazát, mely nem feltétlenül egymás utáni poziciókon található elemeket tartalmaz.
  Példa: 1, 2, -6, 3, 4, 5, -2, 10, -5, -6
- Egy tömbszakasz olyan részsorozat, amely csak egymás utáni poziciókon található elemeket tartalmaz.
  Példa: 1, 2, -6, 3, 4, 5, -2, 10, -5, -6
- Más elnevezéseket ne használjunk a továbbiakban (pl. vizsgán)!

Leghatékonyabban hogyan tudod megoldani a feladatot?

Maximális összegű tömbszakasz

Első megoldás

- Generáljuk az a tömb összes lehetséges tömbszakaszát, mindegyiknek kiszámoljuk az összegét és kiválasztjuk a legnagyobbat.
- A `maxOsszeg` változóba számoljuk ki a maximális összegű tömbszakasz összegét, ezt mindig frissítjük mikor jobb megoldást találtunk az eddiginél.
- A `bal` változóval jelöljük az aktuálisan generált tömbbszakasz legbaloldalibb elemének indexét.
- A `jobb` változóval jelöljük az aktuálisan generált tömbbszakasz legjobboldalibb elemének indexét.
- Az `osszeg` változóba számoljuk ki az aktuálisan generált tömbszakasz hosszát.
- Az `i` változóval járjuk be az aktuálisan generált tömszakaszt az összeg kiszámításához.

Forráskód: `maxTombszakasz1.cpp`

Maximális összegű tömbszakasz

Második megoldás

- Kiszámoljuk a részösszegek `sum` tömbjét, melynek `i`. eleme az `a` tömb első `i` elemének összegével lesz egyenlő.
- Észrevesszük, hogy `a[bal..jobb] = sum[jobb] - sum[bal - 1]`
- Átírjuk az első megoldást úgy, hogy megszabadulunk a belső `for` ciklustól, melyet helyettesítünk a fenti képlettel.

Forráskód: `maxTombszakasz2.cpp`

Maximális összegű tömbszakasz

Harmadik megoldás

- Megszabadulhatunk a `sum` tömbtől és a belső (`i` ciklusváltozót használó) ciklustól is, ha észrevesszük a következőt.
- `a[bal..jobb] = a[bal..jobb-1] + a[jobb]`
- Ezt felhasználva, az `osszeg` változót frissíthetjük egyetlen összeadással minden lépésben mikor a `jobb` változó növekszik és nem kell a teljes intervallumra újraszámolni.

Forráskód: `maxTombszakasz3.cpp`

Maximális összegű tömbszakasz

Negyedik megoldás, Kadane algoritmusa

- A tömb egyetlen bejárásával fogjuk a megoldást megkeresni.
- Minden lépésben lesz egy jelöltünk a maximális összegű tömbszakaszra, ennek az összegét tárolja az `osszeg` változó (mint eddig).
- Két esetünk van: az aktuális jelölt vagy hozzájárulhat egy optimális megoldáshoz, vagy nem
- Mikor áll fenn a második eset?

Maximális összegű tömbszakasz

Negyedik megoldás, Kadane algoritmusa

- A tömb egyetlen bejárásával fogjuk a megoldást megkeresni.
- Minden lépésben lesz egy jelöltünk a maximális összegű tömbszakaszra, ennek az összegét tárolja az `osszeg` változó (mint eddig).
- Két esetünk van: az aktuális jelölt vagy hozzájárulhat egy optimális megoldáshoz, vagy nem
- Mikor áll fenn a második eset? Amikor $\texttt{osszeg} < 0$, hiszen egy negatív összegű tömbszakaszt folytatva csak rosszabb megoldást kaphatunk, mint ha újat kezdenénk.
- Ezért a második esetben az `osszeg` változót 0-ra állítjuk.

Forráskód: `maxTombszakasz4.cpp`

Bónusz

Tervezhetünk-e lineáris megoldást a második megoldás ötletéből kiindulva?
Mennyi? 2 pont a H1 jegybe (40-es skálán)
Mit? Forráskódot és az ötlet rövid magyarázatát
Hova? Canvas privát üzenet
Meddig? Október 22. 23:59

Tartalom

Hogyan hasonlíthatjuk össze két algoritmus hatékonyságát?

Mi a probléma a direkt időméréssel?

Hogyan hasonlíthatjuk össze két algoritmus hatékonyságát?

Mi a probléma a direkt időméréssel?

- Függ a számítógép(hálózat) hardware-jétől.
- Függ a használt programozási nyelvtől.
- Függ a fordítóprogram verziójától.
- Függ a használt optimalizálási szinttől (lásd *Debug* és *Release* mód közötti különbségek).
- Függ az implementálás módjától (pl. paraméterátadás).
- Függ az operációs rendszertől.

Hogyan hasonlíthatjuk össze két algoritmus hatékonyságát?

Mi a probléma a direkt időméréssel?

- Függ a bemeneti adatok méretétől és szerkezetétől.
- Ezt a problémát részben kiküszöbölhetjük, ha véletlenszerűen generált bemeneti adatokra futtatjuk többször az algoritmusokat és így hasonlítjuk össze a futási időket (`randomIdomeres.cpp`).
- Ezt a módszert használhatjuk arra is, hogy empirikusan ellenőrizzük, hogy az algoritmusunk helyes eredményt ad-e, összehasonlítva az eredményt más algoritmusok által adott eredményekkel (`randomHelyesseg.cpp`).

Hogyan hasonlíthatjuk össze két algoritmus hatékonyságát?

- Az előbbi módszeren kicsit finomíthatunk, ha időmérés helyett kinevezünk egy „alapműveletet” és azt számoljuk, hogy ez hányszor hajtódik végre.
- A maximális összegű tömbszakasz feladatában ez az alapművelet lehet az a feltétel, amelyik a `maxOsszeg` változót potenciálisan frissíti (`randomMuveletek.cpp`).
- Ezzel a megközelítéssel nem tudunk különbséget tenni az első és a második megoldás között, pedig az időmérésnél láttuk, hogy a második gyorsabb.
- Továbbra sem küszöböltük ki a véletlenszerű bemenetek és az elvégzett kísérletek száma által bevezetett szórást a mért számokban.
- Egy pontosabb „mértékegységre” van szükségünk, amelyet nem csak a futási idő, hanem más hatékonysági szempontok kifejezésére is használhatunk.

Milyen szempontból lehet „hatékony” egy algoritmus?

Milyen szempontból lehet „hatékony” egy algoritmus?

- Futási idő
- Memóriaigény
- Tárhelyigény
- Klasszikus algoritmusok esetén nem szoktuk említeni, de párhuzamosított algoritmusok esetén ide tartozhat a kommunikációhoz szükséges sávszélesség is (lásd pl. bitcoin bányászat). Ez fontos szempont lehet bármilyen esetben ahol adatokat küldünk interneten keresztül, például webes alkalmazások esetén, kliens-szerver architektúrákban stb.

Még egy probléma az időméréssel

Legrosszabb, legjobb, átlagos esetek

- Láttuk, hogy a futási idő függ a bemeneti adatok méretétől és szerkezetétől.
- A véletlengenerálós módszerrel megtehettük volna, hogy az `n` értékét nem változtatjuk, ekkor tulajdonképpen csak a bemenet szerkezetén változtattunk volna, a méretén nem, így átlagos futási időt közelítettünk volna egy adott `n`-re.
- Nézzünk egy egyszerűbb példát, ahol könnyebben beláthatjuk a legrosszabb, legjobb és átlagos eseteket!

Legrosszabb, legjobb, átlagos esetek

Feladat

Adott egy n elemű a tömb és egy x érték. Állapítsuk meg, hogy x szerepel-e a tömbben!

- Válasszuk itt is alapműveletnek az összehasonlítást!
- A legjobb esetben rögtön az első pozición megtaláljuk a keresett elemet, ekkor a műveletek száma 1.
- A legrosszabb esetben x nem szerepel a tömbben, de ezt csak n összehasonlítás után tudjuk megállapítani.
- Kis matekezéssel könnyedén levezethető, hogy az átlagos esetben $\frac{n+1}{2}$ összehasonlításra van szükségünk (lásd jegyzet).

Algoritmusok növekedési rendje

- A fentiekből lekövetkeztethetjük, hogy nem érdemes, de nem is lehetséges pontos, precíz értékeket keresni futási idő megadására (ez sok esetben a többi hatékonysági szempontra is igaz).
- Az esetek túlnyomó többségében arra kell szorítkoznunk, hogy a végrehajtási idő nagyságrendjét határozzuk meg.
- Láttuk, hogy ezt kézenfekvő a bemeneti adatok méretének függvényében kifejezni, ezt általában n-el jelöljük (vannak esetek, amikor nem elég egy paraméter a bemeneti adatok méretének kifejezéséhez).

Algoritmusok növekedési rendje

Példák a bemeneti adatok méretére

- Az eddigi két példában egy vektor volt a bemeneti adat, ilyenkor n a tömb elemeinek számát jelöli.
- Két mátrix összeszorzásakor a mátrixok sorainak és oszlopainak számát tekintjük a bemeneti adatok méretének.
- Nagy számok összeadásakor a számok számjegyeinek száma a méret.
- Gráfelméleti feladatokban általában a csomópontok számát (n) és sokszor az élek számát (m) is figyelembe vesszük.

Algoritmusok növekedési rendje

Az alapművelet

- Amikor egy algoritmus hatékonyságát próbáljuk meghatározni futási idő szempontjából, annak a nagyságrendjét próbáljuk kifejezni, hogy egy adott alapművelet hányszor hajtódik végre.
- Láttuk, hogy fontos, hogy hogyan választjuk meg ezt a műveletet, sőt előfordulhat, hogy több alapműveletet is kell választanunk a pontos eredmény érdekében.
- Jellemzően a „legbelsőbb” műveletekre kell gondolnunk, amelyek a legtöbbször hajtódnak végre.

- Az előző két példában az összehasonlítás volt az alapművelet.
- Ahhoz, hogy pontos eredményt kapjunk a maximális összegű tömbszakasz feladatának esetében, az első változatban a legbelsőbb `for` cikluson belüli összeadást is alapműveletnek kell kineveznünk.
- Ha két mátrixot kell összeszoroznunk, akkor két szám szorzása lesz az alapművelet, de érdemes lehet az összeadásokat is figyelembe venni.
- Két nagy szám összeadásakor két számjegy feldolgozása lesz az elemi művelet.

Algoritmusok növekedési rendje

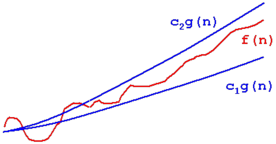
- Az algoritmusok bonyolultságát (komplexitását) a bemeneti adatok méretének (n, m stb.) függvényeként írjuk le.
- A képletnek csak a fő tagját tartjuk meg, pl. $an^2 + bn + c$-ből csak az an^2-et, mivel az alacsonyabb rendű tagok nagy n-re kevésbé lényegesek.
- Szintén figyelmen kívül hagyjuk a fő tag konstans szorzóját, mivel nagy bemenetekre ez is elhanyagolható, így csak n^2 marad a fenti példában.
- Az így kapott kifejezést nevezzük az algoritmus növekedési rendjének.
- Általában akkor mondjuk, hogy egy algoritmus hatékonyabb egy másiknál, ha a legrosszabb esetben való növekedési rendje kisebb.
- Mennyi a maximális összegű tömbszakasznál tárgyalt megoldások növekedési rendje?

Algoritmusok növekedési rendje

- Az algoritmusok bonyolultságát (komplexitását) a bemeneti adatok méretének (n, m stb.) függvényeként írjuk le.
- A képletnek csak a fő tagját tartjuk meg, pl. $an^2 + bn + c$-ből csak az an^2-et, mivel az alacsonyabb rendű tagok nagy n-re kevésbé lényegesek.
- Szintén figyelmen kívül hagyjuk a fő tag konstans szorzóját, mivel nagy bemenetekre ez is elhanyagolható, így csak n^2 marad a fenti példában.
- Az így kapott kifejezést nevezzük az algoritmus növekedési rendjének.
- Általában akkor mondjuk, hogy egy algoritmus hatékonyabb egy másiknál, ha a legrosszabb esetben való növekedési rendje kisebb.
- Mennyi a maximális összegű tömbszakasznál tárgyalt megoldások növekedési rendje? Az első köbös (n^3), a második és a harmadik négyzetes (n^2), a negyedik lineáris (n).

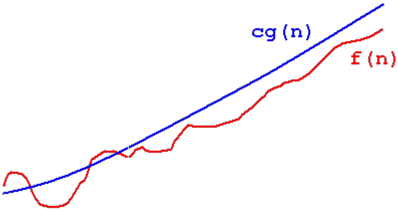
- Formálisan
  $\Theta(g(n)) = \{f(n) | \exists c_1, c_2, n_0 > 0 : 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0\}$
- Az aszimptotikus jelölések függvények halmazát jelölik, vagyis matematikailag úgy lenne pontos írni, hogy $3n^2 + 5n + 2 \in \Theta(n^2)$, de a jelöléseket kicsit feloldva $f(n) = \Theta(g(n))$-t szoktuk használni.
- A Θ jelölést szavakkal úgy is leírhatjuk, hogy minden $n \geq n_0$ esetén az $f(n)$ függvény – egy állandó szorzótényezőtől eltekintve – egyenlő $g(n)$-el.
- Ezt úgy mondjuk, hogy $g(n)$ aszimptotikusan éles korlátja $f(n)$-nek.

- Más szóval az $f(n)$ függvény hozzátartozik a $\Theta(g(n))$ halmazhoz, ha léteznek c_1 és c_2 állandók úgy, hogy $f(n)$ – elég nagy n-re – beszorítható $c_1 g(n)$ és $c_2 g(n)$ közé.
- Grafikusan



A konstans (állandó) bonyolultságot $\Theta(1)$-el jelöljük, ekkor az algoritmus futási ideje nem függ a bemeneti adatok méretétől, vagy ha memóriáról beszélünk nem használ n-el arányos méretű plusz memóriát.

- Az *O* jelölés aszimptotikus felső korlátot ad.
- $O(g(n)) = \{f(n) | \exists c, n_0 > 0 : 0 \leq f(n) \leq cg(n), \forall n \geq n_0\}$



Aszimptotikus jelölések
Ω jelölés

- Aszimptotikus alsó korlátot ad.
- $\Omega(g(n)) = \{f(n) | \exists c, n_0 > 0 : 0 \leq cg(n) \leq f(n), \forall n \geq n_0\}$

Aszimptotikus jelölések és legrosszabb, legjobb, átlag esetek kapcsolata

- Mivel a legjobb és legrosszabb esetben általában pontosan kiszámítható a lépések száma, ezek leírásakor a Θ jelölés adja a legtöbb információt. Például mondhatjuk, hogy egy érték megkeresése egy tömbben legrosszabb esetben $\Theta(n)$ időben fut.
- Ugyanakkor intuitív a legrosszabb eset leírásakor a O és a legjobb eset leírásakor a Ω jelöléseket használni. Ez nem hibás, csak kevesebb információt nyújt, mint a Θ, hiszen ezek a jelölések nem adnak éles korlátot.
- Az átlag eset leírásakor általában a Θ-t használjuk.
- Néha hivatkozunk egy algoritmus futási idejének bonyolultságára, anélkül, hogy kiemelnénk, hogy legjobb, legrosszabb vagy átlag esetről beszélünk (esetleg úgy is mondhatjuk, hogy „teljes futási idő”). Ekkor egy olyan aszimptotikus jelölést keresünk, amely az összes esetet magában foglalja. Például egy érték megkeresése egy tömbben $O(n)$ időben fut (mert sokszor $\Theta(n)$, de a legjobb esetben csak $\Theta(1)$).

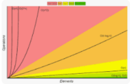
Vagyis: néha még a Google nulladik találata is helytelen

algorithmic complexity

All | Images | Videos | News | Maps | More | Tools

About 27,800,000 results (0.71 seconds)

Algorithmic complexity is **a measure of how long an algorithm would take to complete given an input of size n**. If an algorithm has to scale, it should compute the result within a finite and practical time bound even for large values of n. ... Algorithmic complexity is also called complexity or running time.



https://devopedia.org › algorithmic-complexity

Algorithmic Complexity - Devopedia

About featured snippets • Feedback

DEVOPEDIA®
for developers. by developers.

Summary

Discussion

What notations are used to represent algorithmic complexity?

What does it mean to state best-case, worst-case and average time complexity of algorithms?

Why should we care about an algorithm's performance when processors are getting faster and memories are getting cheaper?

Are there techniques to figure out the complexity of algorithms?

If an algorithm is inefficient,

Order of growth of algorithms specified in Big-O notation. Source: Big-O Cheat Sheet, 2016.

Big-O notation is the prevalent notation to represent algorithmic complexity. ~~It gives an upper bound on complexity and hence it signifies the worst-case performance of the algorithm.~~ With such a notation, it's easy to compare different algorithms because the notation tells clearly how the algorithm scales when input size increases. This is often called the *order of growth*.

Az algoritmus által feldolgozott adatok számára szükséges memória mérete

- Gyakran előfordul, hogy egy program a bemeneti és kimeneti adatokon kívül, ideiglenesen létrehozott adatszerkezetekkel is dolgozik.
- Amikor egy algoritmus memóriabonyolultságáról beszélünk, akkor általában ezekre gondolunk, tehát a bemeneti adatok tárolására szükséges memóriát nem vesszük figyelembe. Például a maximális összegű tömbszakasz második megoldásában használt `sum` tömb $\Theta(n)$ plusz memóriát jelent, tehát azt mondhatjuk, hogy ennek az algoritmusnak a memóriabonyolultsága $\Theta(n)$.
- Hasonlóképpen, ha egy segédmátrixot használna az algoritmusunk, a memóriabonyolultság $\Theta(n^2)$ lenne.
- Ha egy algoritmus feldolgozás közben csak konstans méretű plusz memóriát vesz igénybe (vagyis memóriabonyolultsága $\Theta(1)$), azt mondjuk, hogy helyben dolgozik. Ekkor a lefoglalt memória mérete nem függ a bemeneti adatok méretétől.

Tartalom

Az „algoritmus” szó eredete

- Abu-Ja far Mohammed ibn Mura al Kvarîzmi matematikus nevének latinos átírásából származik.
- Nevét írják még Muhammad ibn Musa al-Khwarizmi-nek is.
- A mai Üzbegisztán területén született 780-ban, Bagdadban halt meg 850-ben.
- Munkáit arabul írta, származása valószínűleg perzsa.

Az algoritmus fogalma

- Lépések sorozata, mely megold egy konkrét feladatot.
- Véges számú lépés
- Bemeneti adatok (lehet üres is)
- Kimeneti adatok (lehet üres is)
- Mechanikusan elvégezhető, vagyis anélkül, hogy az ember arra szorulna, hogy ő hozzon döntéseket
- Ennek ellenére lehet determinisztikus, vagy nemdeterminisztikus

Böhm és Jacopini tétele

Tétel

Bármely algoritmus megvalósítható a következő három alapstruktúrával:

1. Szekvencia
2. Elágazás (döntés)
3. Elöltesztelő ismeretlen lépésszámú ciklus (iteráció)

Tartalom

Miért van szükség pszeudokódra?

Pszeudokód
Utasítások

- Lineáris struktúra: művelet feltétel nélküli elvégzése
- Elágazási (alternatív) struktúra (`if`, `then`, `else`)

```
HA (feltétel) akkor
   ...
KÜLÖNBEN
   ...
VÉGE(Ha)
```

- Elöltesztelő ismétlő struktúra / ciklus (`while`)

```
AMÍG (feltétel) végezd el:
   ...
VÉGE(Amíg)
```

- Hátultesztelő ismétlő struktúra / ciklus (`do`, `while` / `repeat`, `until`)

```
ISMÉTELD
   ...
AMEDDIG (feltétel)
```

- Ismert lépésszámú ismétlő struktúra / ciklus (`for`)

```
MINDEN i = ké, vé[, lépés] végezd el:
   ...
VÉGE(Minden)
```

Ha a `lépés`-t nem adjuk meg, akkor +1-nek tekintjük.

Megjegyzés: a szögletes zárójelek közé írt paraméterek opcionálisak

Pszeudokód

Utasítások

- Algoritmus első utasítása: `ALGORITMUS Név(formális paraméterek)`
- Algoritmus utolsó utasítása: `VÉGE(Algoritmus)`
- Beolvasás: `BE: változólista`
- Kiírás: `KI: kifejezéslista`
- Értékadás:
  `változónév` $\leftarrow$ `kifejezés` vagy
  `változónév = kifejezés` vagy
  `változónév := kifejezés`
- Alprogramhívás: `Alprogramnév(aktuális paraméterek)`
- Érték visszatérítése alprogramból való kilépéssel: `VISSZATÉRÍT[ érték]`
- `\\Ez itt egy megjegyzés`

Tartalom

Programozási tételek

- Gyakran ismétlődő (rész)feladatokra adott mintamegoldások, „kapszulák”.
- Garantáltan helyesek és optimálisak.
- Annyira gyakori feladatokat oldanak meg, hogy bizonyos programozási nyelvekben külön utasítással valósították meg egy részüket (pl. C++ STL, Java 8 streamek)
- Az egyszerűbb feladatok megoldásai összerakhatóak belőlük mint egy puzzle.

Sorozatszámítás

Feladat

Egy n elemű a sorozathoz kell egy s értéket hozzárendelnünk. Az s értéket az egész sorozaton értelmezett f függvény adja, mely felbontható értékpárokon kiszámított függvények sorozatára: $f(a_1, a_2, a_3, \ldots, a_n) = f(\ldots f(f(a_1, a_2), a_3) \ldots a_n)$

A megoldás az f_0 semleges elemre épül.

Sorozatszámítás

```
ALGORITMUS Sorozatszámítás(n, a, s)
  s = f0
  MINDEN i = 1, n végezd el:
    s = f(s, a[i])
  VÉGE(Minden)
VÉGE(Algoritmus)
```

Sorozatszámítás

Speciális eset: összegszámítás

```
ALGORITMUS Összegszámítás(n, a, s)
  s = 0
  MINDEN i = 1, n végezd el:
    s = s + a[i]
  VÉGE(Minden)
VÉGE(Algoritmus)
```

Döntés

Feladat

Adott egy n elemű a sorozat és az elemein értelmezett T tulajdonság. Döntsük el, hogy van-e a sorozatban T tulajdonságú elem!

A válasz egy logikai változó (`igaz` vagy `hamis`).

Az első megközelítésben a sorozatszámítást használjuk a logikai `VAGY` művelettel, az eredményt a `talált` változó tartalmazza.

```
ALGORITMUS Döntés1(n, a, talált)
  talált = HAMIS
  MINDEN i = 1, n végezd el:
    talált = talált VAGY T(a[i])
  VÉGE(Minden)
VÉGE(Algoritmus)
```

Döntés

Második változat

- Észrevesszük, hogy ha a `talált` változó `IGAZ`zá vált, többet biztosan nem fog `HAMIS`sá válni.
- Ebben az esetben nincs értelme tovább folytatni a ciklust.
- Ezért `MINDEN` típusú ciklus helyett ismeretlen lépésszámú struktúrát választunk.
- Mi a legrosszabb eset?

Döntés

Második változat

- Észrevesszük, hogy ha a `talált` változó `IGAZ`zá vált, többet biztosan nem fog `HAMIS`sá válni.
- Ebben az esetben nincs értelme tovább folytatni a ciklust.
- Ezért `MINDEN` típusú ciklus helyett ismeretlen lépésszámú struktúrát választunk.
- Mi a legrosszabb eset? Ha nincs T tulajdonságú elem a sorozatban.

Döntés

Második változat

```
ALGORITMUS Döntés2(n, a, talált)
  i = 1
  talált = HAMIS
  AMÍG (NEM talált ÉS (i <= n)) végezd el:
    HA (NEM(T(a[i])))
      i = i + 1
    KÜLÖNBEN
      talált = IGAZ
    VÉGE(Ha)
  VÉGE(Amíg)
VÉGE(Algoritmus)
```

Döntés

Harmadik változat: a második változat tömörebben

```
ALGORITMUS Döntés3(n, a, talált)
  i = 1
  talált = HAMIS
  AMÍG ((i <= n) ÉS NEM T(a[i])) végezd el:
      i = i + 1
  VÉGE(Amíg)
  talált = (i <= n)
VÉGE(Algoritmus)
```

Döntés

Negyedik változat

- Változik a feladat, itt azt kell eldöntenünk, hogy az adatok teljességükben rendelkeznek-e az adott tulajdonsággal.
- Más szavakkal: nem létezik egyetlen elem sem, amely ne lenne T tulajdonságú.
- A bemenet minden elemét meg kell vizsgálnunk.
- Mivel a kimenet jelentése minden adatra érvényes, a `talált` változót `mind`-re cseréljük.

Döntés

Negyedik változat

```
ALGORITMUS Döntés4(n, a, mind)
  i = 1
  mind = HAMIS
  AMÍG ((i <= n) ÉS T(a[i])) végezd el:
      i = i + 1
  VÉGE(Amíg)
  mind = (i > n)
VÉGE(Algoritmus)
```

Kiválasztás

Feladat

Adott egy n elemű a sorozat és az elemein értelmezett T tulajdonság. Tudván, hogy a sorozatban garantáltan létezik legalább egy T tulajdonságú elem, adjuk meg egy ilyen elem sorszámát!

Ha megelégszünk egyetlen elem sorszámával, akkor az eredmény egy sorszám, balról nézve az első adott tulajdonságú elem helye a sorozatban.

Kiválasztás

```
ALGORITMUS Kiválasztás(n, a, hely)
  hely = 1
  AMÍG (NEM T(a[hely])) végezd el: //nem kell a hely <= n
      hely = hely + 1
  VÉGE(Amíg)
VÉGE(Algoritmus)
```

Szekvenciális (lineáris) keresés

Feladat

Adott egy n elemű a sorozat és az elemein értelmezett T tulajdonság. Vizsgáljuk meg, hogy van-e T tulajdonságú elem a sorozatban és ha igen, adjuk meg az egyiknek a pozícióját!

- A megoldáshoz kombináljuk a Döntést és a Kiválasztást.
- Az összehasonlítások száma legkevesebb 1, legtöbb n, az átlag pedig $\frac{n+1}{2}$

Szekvenciális (lineáris) keresés

Első változat

```
ALGORITMUS Keres1(n, a, hely)
  hely = 0
  i = 1
  AMÍG ((hely = 0) ÉS (i <= n)) végezd el:
      HA (T(a[i])) akkor
        hely = i
      KÜLÖNBEN
        i = i + 1
      VÉGE(Ha)
  VÉGE(Amíg)
VÉGE(Algoritmus)
```

Szekvenciális (lineáris) keresés

Második változat

```
ALGORITMUS Keres2(n, a, hely)
  i = 1
  AMÍG ((i <= n) ÉS NEM T(a[i])) végezd el:
    i = i + 1
  VÉGE(Amíg)
  HA (i <= n) akkor
    hely = i
  KÜLÖNBEN
    hely = 0
  VÉGE(Ha)
VÉGE(Algoritmus)
```

Szekvenciális (lineáris) keresés

Harmadik változat

- A Döntéshez hasonlóan a követelmény kérheti minden olyan eleme megkeresését, amely rendelkezik a T tulajdonsággal
- Ekkor bejárjuk a teljes adathalmazt `MINDEN` ciklust alkalmazva.
- Azokat a pozíciókat ahol megfelelő elemet találunk vagy kiíratjuk, vagy megőrizzük egy sorozatban.

Megszámlálás

Feladat

Adott egy n elemű a sorozat és az elemein értelmezett T tulajdonság. Határozzuk meg a T tulajdonsággal rendelkező elemek számát!

- Mivel minden elemet meg kell vizsgálnunk, `MINDEN` típusú struktúrával dolgozunk.
- Az eredményt a `db` változóban számoljuk.

Megszámlálás

```
ALGORITMUS Megszámlálás(n, a, db)
  db = 0
  MINDEN i = 1, n végezd el:
    HA (T(a[i])) akkor
      db = db + 1
    VÉGE(Ha)
  VÉGE(Minden)
VÉGE(Algoritmus)
```

Maximumkiválasztás (minimumkiválasztás)

Feladat

Adott egy n elemű a sorozat. Határozzuk meg a sorozat legnagyobb (vagy legkisebb) értékét (vagy annak sorszámát)!

- Előfordulhat több legnagyobb elem létezik, de nekünk csak az értékét vagy az első előfordulási helyét kell meghatároznunk.
- Mivel minden elemet meg kell vizsgálnunk, ezért `MINDEN` ciklust használunk.
- A `max` segédváltozóban kapjuk meg a maximum értékét.
- A segédváltozónak az adatok közül választunk kezdőértéket, mivel így nem áll fenn a veszély, hogy az algoritmus eredménye egy, az adataink között nem szereplő érték legyen.
- Ha ez valamiért nem lehetséges vagy praktikus, használhatunk egy olyan értéket ($-\infty$), amelynél biztosan lesz nagyobb (kisebb) az értékek között.

Maximumkiválasztás

```
ALGORITMUS Maximumkiválasztás1(n, a, max)
  max = a[1]
  MINDEN i = 2, n végezd el:
    HA (max < a[i])
      max = a[i]
    VÉGE(Ha)
  VÉGE(Minden)
VÉGE(Algoritmus)
```

Maximumkiválasztás

Második változat, maximum helye

```
ALGORITMUS Maximumkiválasztás2(n, a, hely)
  hely = 1
  MINDEN i = 2, n végezd el:
    HA (a[hely] < a[i])
      hely = i
    VÉGE(Ha)
  VÉGE(Minden)
VÉGE(Algoritmus)
```

Maximumkiválasztás

Második változat, maximum helye

- Ha nem az első, hanem az utolsó maximumot tároló pozíciót keressük, bejárhatjuk a sorozatot fordított sorrendben az előző algoritmust módosítva.
- Ekkor egy másik lehetőség $<$ helyett $<=$-t használni.

Maximumkiválasztás

Harmadik változat, minden maximum helye

- A feladat egy másik változatában az összes olyan pozíciót keressük, ahol a legnagyobb érték található.
- Ezt megoldhatjuk a sorozat kétszeri bejárásával.
- Először megállapítjuk a maximum értékét.
- Majd bejárjuk újra a sorozatot, abból a célből, hogy kiírhassunk minden indexet, amelyhez ez a legnagyobb érték tartozik.

Maximumkiválasztás

Harmadik változat, minden maximum helye

```
ALGORITMUS Maximumkiválasztás3(n, a, db, indexek)
  max = a[1]
  MINDEN i = 2, n végezd el:
    HA (max < a[i])
      max = a[i]
    VÉGE(Ha)
  VÉGE(Minden)
```

Maximumkiválasztás

Harmadik változat, minden maximum helye

```
  db = 0
  MINDEN i = 1, n végezd el:
    HA (a[i] = max) akkor
      db = db + 1
      indexek[db] = i
    VÉGE(Ha)
  VÉGE(Minden)
VÉGE(Algoritmus)
```

Maximumkiválasztás

Negyedik változat, minden maximum helye, egyszeri bejárással

```
ALGORITMUS Maximumkiválasztás4(n, a, db, indexek)
  max = a[1]
  db = 1
  indexek[1] = 1
```

Maximumkiválasztás

Negyedik változat, minden maximum helye, egyszeri bejárással

```
  MINDEN i = 2, n végezd el:
    HA (max < a[i]) akkor
      max = a[i]
      db = 1
      indexek[db] = i
    KÜLÖNBEN
      HA (max = a[i])
        db = db + 1
        indexek[db] = i
      VÉGE(Ha)
    VÉGE(Ha)
  VÉGE(Ha)
VÉGE(Algoritmus)
```