Alapvető algoritmusok

11. előadás

Dr. Pătcaș Csaba

BABEȘ-BOLYAI TUDOMÁNYEGYETEM
Matematika és Informatika Kar

Tartalom

1. Bonyolultsági osztályok
2. Oszd meg és uralkodj módszer
   - Szorzat
   - Minimumszámolás
   - Gyors hatványozás (másképp)
   - Bináris keresés
   - Hanoi tornyok
3. Nem-független részfeladatok (Ellenpéldák)
   - Fibonacci
   - Úszómedence
4. Rendezési algoritmusok
   - Összefésülésen alapuló rendezés (Mergesort)
   - Gyorsrendezés (Quicksort)

Bonyolultsági osztályok

- Az informatikában megjelenő feladatokat különböző bonyolultsági osztályokba sorolhatjuk, ezek közül számunkra a legfontosabbak a `P`, az `NP` és az `NP-teljes` osztályok.
- A `P` vagy `PTIME` bonyolultsági osztályba azok a feladatok tartoznak, melyek megoldhatóak polinomiális időben, vagyis a legrosszabb esetben vett időbonyolultságuk $n^{O(1)}$ formában írható. Például: osztja-e egyik szám a másikat; annak ellenőrzése, hogy egy szám prím-e
- A matematikai precizitást elhagyva, köznyelvileg mondhatjuk, hogy az `NP` bonyolultsági osztályba azok a feladatok tartoznak, melyek eredménye ellenőrízhető polinomiális időben. Például: a Hamilton-kör létezésének kérdésében a csúcsok egy sorozatáról könnyen ellenőrízhető, hogy egy Hamilton-kört írnak-e le; ha a kérdés az, hogy ki tudunk-e pontosan fizetni egy adott összeget bizonyos érmékkel, a kiválasztott érmék értékeit csak össze kell adnunk az ellenőrzéshez.

P és NP kapcsolata

$\mathtt{P} = \mathtt{NP}$ vagy $\mathtt{P} \neq \mathtt{NP}$?

Könnyen belátható, hogy $\mathtt{P} \subseteq \mathtt{NP}$, de az máig nyitott kérdés, hogy a két feladatosztály egyenlő-e, vagy $\mathtt{P} \subset \mathtt{NP}$. A szakértők túlnyomó többsége szerint a második eset áll fenn, de ezt még nem sikerült bizonyítani.

NP-teljes feladatok

- Az `NP-teljes` feladatok a legnehezebb (legáltalánosabb) feladatok az `NP` osztályból.
- Bármelyik `NP` feladat levezethető (redukálható) bármely `NP-teljes` feladatra.
- Ebből következik, hogy az `NP-teljes` feladatok egymásra redukálhatóak.

NP-teljes feladatok

- Királynők feladata
- Sudoku
- Kifizethető-e adott összeg adott érmékkel? (`Subset Sum`)
- Felbontható-e egy halmaz két egyforma összegű részhalmazra? (`Partition`)
- Létezik-e Hamilton-kör egy adott általános gráfban?
- Kiszínezhetőek-e egy gráf csúcsai kevesebb, mint k színnel, úgy, hogy két szomszédos csúcs ne legyen azonos színű? (`Chromatic number` vagy `Graph coloring`)
- Létezik-e egy gráfban k csúcsnál többet tartalmazó teljes részgráf? (`Clique problem`)
- Lefedhető-e az $\{1, \ldots, n\}$ halmaz összes eleme kevesebb mint k halmaz kiválasztásával a megadott m közül? (`Set cover`, ha egy számot csak egyszer szabad lefedni, akkor `Exact cover`)

Könnyű és nehéz feladatok

- Általában azokat a feladatokat tekintjük könnyen megoldhatónak (tractable), amelyek megoldására ismerünk polinomiális idejű algoritmust, vagyis a `P` feladatosztályba tartoznak.
- Ezekkel ellentétben vannak a nehezen megoldható (intractable) feladatok, melyeknek legfontosabb csoportját az `NP-teljes` feladatok képezik.
- Az `NP-teljes` feladatokat azért fontos ismerni, mert ha egy feladatról tudjuk, hogy nehezen megoldható, abból következik, hogy nagy eséllyel nincs értelme polinomiális megoldási algoritmust keresnünk rá.
- Ismeretlen feladatokról is gyakran úgy bizonyítjuk, hogy nehezen megoldhatóak, hogy levezetjük őket egy ismert `NP-teljes` feladatra.

Mihez kezdhetünk a nehéz feladatokkal?

- Kis bemenetekre alkalmazhatunk nyers erőre, vagy visszalépéses keresésre alapuló megközelítéseket.
- Ha egy gyorsan előállítható megoldásra van szükségünk, valamilyen greedy heurisztikán alapuló stratégia jó választás lehet.
- Bizonyos feladatokra léteznek közelítő algoritmusok, amelyek garantált hibaszázalékon belül maradnak az optimumhoz képest.
- Valamivel nagyobb futási időt igényelnek, de sokszor lényegesen jobb megoldásokkal szolgálnak, a mesterséges intelligencia területéről ismert különböző metaheurisztikák. Ezek egy része nemdeterminisztikus algoritmus, vagyis véletelenszámokra alapszik, így két egymás utáni futás nem mindig ad azonos eredményt.

Tartalom

Oszd meg és uralkodj (Divide et impera) módszer

Mikor használjuk?

- Amikor az eredeti feladat felbontható hozzá hasonló kisebb részfeladatokra.
- Olyankor hatékony, ha ezek a részfeladatok egymástól függetlenek, az előadás első részében erre látunk majd példákat.
- Amikor a részfeladatok nem függetlenek egymástól, a futási idő exponenciálissá válhat, erre nézünk meg két példát a második részben.

Oszd meg és uralkodj (Divide et impera) módszer

Hogyan alkalmazzuk?

1. Felbontjuk az eredeti feladatot olyan részfeladatokra, amelyek az eredetihez hasonlóak, de kisebb adathalmazra definiáltak.
2. A részfeladatokkal hasonlóan járunk el, ameddig nagyon egyszerű részfeladatokhoz nem jutunk.
3. Ezeket a legegyszerűbb részfeladatokat megoldjuk.
4. A részfeladatok eredményeiből fokozatosan felépítjük mindig a következő méretű feladat eredményeit, ezek összerakása által. Az utolsó összerakás az eredeti feladat eredményét adja.

Hogyan implementáljuk?

- Megfigyelhetjük, hogy a részfeladatok bejárása top-down sorrendben történik, viszont a megoldásuk bottom-up sorrendben.
- Ezért a legtöbb esetben rekurzívan természetes implementálni a *Divide et impera* módszerrel megoldott feladatokat.
- A felbontást a rekurzív hívások által valósítjuk meg, az eredmények összerakását a rekurzióból való visszalépés után (mert ekkor lesznek megoldva az adott részfeladatok).

Szorzat

Megjegyzés: a `Szorzat` és `Minimumszámolás` feladatok csak didaktikai célt szolgálnak a módszer megértéséhez, a programozási tételeket alkalmazó megoldások hatékonyabbak és közérthetőbbek, ezért a gyakorlatban ezeket használjuk.

Feladat

Számítsuk ki n valós szám szorzatát!

Szorzat

Megoldás

- A szorzatot részszorzatokra bontjuk.
- A szorzótényezőket két csoportja osztjuk.
- Kiszámítjuk egy-egy csoport szorzatát.
- A két csoport kiszámított szorzatát összeszorozzuk.
- A felbontást addig végezzük, ameddig egy csoport egy szorzótényezőből nem fog állni.

Szorzat

Megoldás

- Minden részfeladat más-más szorzatot számol ki, tehát ezek függetlenek egymástól.
- Ezt fás ábrázolásban úgy is elképzelhetjük, hogy egy csúcs leszármazottaihoz olyan részfeladatok tartoznak, amelyek között nincs „átfedés” (a sorozat valamely eleme csak az egyikben szerepel).
- Előnyösebb, ha a megoldásunk egyből visszatériti az eredményt, ezért ezt egy rekurzív függvénnyel implementáljuk.

Szorzat

Pszeudokód

```
ALGORITMUS SzorzatDivImp(a, bal, jobb)
  HA (bal = jobb) akkor
    VISSZATÉRÍT: a[bal]
  KÜLÖNBEN
    közép = (bal + jobb) / 2
    p1 = SzorzatDivImp(a, bal, közép)
    p2 = SzorzatDivImp(a, közép + 1, jobb)
    VISSZATÉRÍT: p1 * p2
  VÉGE(Ha)
VÉGE(Algoritmus)
```

Minimumszámolás

Feladat

Állapítsuk meg n szám közül a legkisebbet!

A megoldáshoz hasonlóan járunk el, mint az előző feladatnál (feltételezzük, hogy a `min` függvény visszatéríti a két paraméter közül a kisebbiket).

Minimumszámolás
Pszeudokód

```
ALGORITMUS MinDivImp(a, bal, jobb)
  HA (bal = jobb) akkor
    VISSZATÉRÍT: a[bal]
  KÜLÖNBEN
    közép = (bal + jobb) / 2
    min1 = MinDivImp(a, bal, közép)
    min2 = MinDivImp(a, közép + 1, jobb)
    VISSZATÉRÍT: min(min1, min2)
  VÉGE(Ha)
VÉGE(Algoritmus)
```

Gyors hatványozás (másképp)

Feladat

Határozzuk meg egy x valós szám k. hatványát, ahol k egy egész szám.

- Korábbi előadásokon láttuk, hogy a gyors hatványozás algoritmusa $\log k$ nagyságrendű szorzást hajt végre, a triviális $k - 1$ helyett.
- Implementáltuk a módszert iteratívan és rekurzívan is, az *Oszd meg és uralkodj* módszert használó megoldásban a rekurzív megoldáshoz hasonlóan, a k. hatványra emelést a $k/2$. hatványra emelés segítségével oldjuk meg (ez lesz a részfeladat).
- Negatív hatványkitevő esetén visszavezetjük a megoldást a pozitív esetre a `GyorsHatványDivImp(1 / x, -k)` hívással.

Gyors hatványozás (másképp)
Pszeudokód

```
ALGORITMUS GyorsHatványDivImp(x, k)
  HA (k = 0) akkor
    VISSZATÉRÍT: 1
  KÜLÖNBEN
    segéd = GyorsHatványDivImp(x, k / 2)
    HA ((k % 2) = 1) akkor
      VISSZATÉRÍT: segéd * segéd * x
    KÜLÖNBEN
      VISSZATÉRÍT: segéd * segéd
    VÉGE(Ha)
  VÉGE(Ha)
VÉGE(Algoritmus)
```

Bináris keresés

Feladat

Adott egy n egész számból álló szigorúan növekvő sorozat. Állapítsuk meg egy adott szám helyét a sorozatban! Ha az illető szám nem található meg a sorozatban, írjunk ki megfelelő üzenetet!

Példa: [1 4 5 11 12 13 25]

Ha a keresett szám a 13-as, visszatérítjük, hogy ez a 6. pozíción található.

Ha a keresett szám a 14-es, visszatérítjük, hogy ez nem található meg a sorozatban.

Megjegyzés: A mohó módszernél a `Növekvő részsorozatokra bontás` feladatánál láthattuk, hogy a bináris keresés módszere ennél általánosabb, használható a legkisebb elem helyének meghatározására, amely nagyobb mint a keresett elem, vagy a legnagyobb elem helyének a meghatározására, amely kisebb, mint a keresett elem. A sorozat lehet implicit is, például egy folytonos valós függvény.

Az elemet a sorozat közepén fogjuk először keresni, egyre csökkentjük az aktuális sorozatot, amelyben a keresést végezzük ennek a végeit a `bal` és `jobb` változók jelzik, kezdetben `bal = 1, jobb = n`. Három eset lehetséges:

1. Ha `keresett = a[közép]`, megtaláltuk az elemet a `közép` indexen.
2. Ha `keresett < a[közép]`, mivel a sorozat rendezett, az elemet az aktuális sorozat első felében keressük tovább a `bal..közép - 1` intervallumban.
3. Ha `keresett > a[közép]`, a keresett számot az aktuális sorozat második felében keressük tovább a `közép + 1..jobb` intervallumban.

- A feladat átalakul ugyan két részfeladattá, de ezek közül csak az egyiket kell megoldani.
- Így nem lesz szükség a *Divide et impera* utolsó lépésére, az eredmények összerakására.
- Ennek köszönhetően a bináris keresés iteratívan is könnyedén implementálható.
- Ha az aktuális sorozat üressé vált, azt jelenti, hogy a keresett elem nem található meg a bemeneti sorozatban.

Bináris keresés (rekurzívan)
Pszeudokód

```
ALGORITMUS BinKeresRek(bal, jobb, keresett, a)
  HA (bal > jobb) akkor
    VISSZATÉRÍT: -1 //a keresett elem nincs a sorozatban
  KÜLÖNBEN
```

Bináris keresés (rekurzívan)

Pszeudokód

```
    közép = (bal + jobb) / 2
    HA (keresett > a[közép]) akkor
      VISSZATÉRÍT: BinKeresRek(közép + 1, jobb, keresett, a)
    KÜLÖNBEN
      HA (keresett < a[közép]) akkor
        VISSZATÉRÍT: BinKeresRek(bal, közép - 1, keresett, a)
      KÜLÖNBEN
        VISSZATÉRÍT: közép
      VÉGE(Ha)
    VÉGE(Ha)
  VÉGE(Ha)
VÉGE(Algoritmus)
```

```
ALGORITMUS BinKeresIt(bal, jobb, keresett, a)
  bal = 1
  jobb = n
  megvan = HAMIS
```

Bináris keresés (iteratívan)
Pszeudokód

```
AMÍG ((NEM megvan) ÉS (bal <= jobb)) végezd el:
  közép = (bal + jobb) / 2
  HA (keresett > a[közép])
    bal = közép + 1
  KÜLÖNBEN
    HA (keresett < a[közép])
      jobb = közép - 1
    KÜLÖNBEN
      megvan = IGAZ
    VÉGE(Ha)
  VÉGE(Ha)
VÉGE(Amíg)
```

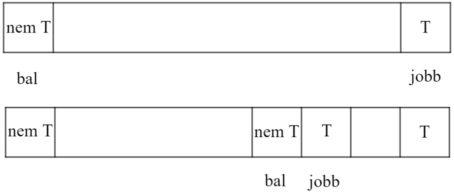
Bináris keresés (iteratívan)

Pszeudokód

```
  HA (NEM megvan) akkor
    VISSZATÉRÍT: -1
  KÜLÖNBEN
    VISSZATÉRÍT: közép
  VÉGE(Ha)
VÉGE(Algoritmus)
```

Általánosított (diszkrét) bináris keresés

- Legyen `T` egy tulajdonság melyet a tömb elemein értelmezünk.
- Feltételezzük, hogy kezdetben a `bal` indexen található elem nem rendelkezik a `T` tulajdonsággal, míg a `jobb` indexen található rendelkezik a `T` tulajdonsággal ($\texttt{bal} < \texttt{jobb}$).
- Ekkor a bináris keresés ezen változata talál két szomszédos elemet ($\texttt{jobb} - \texttt{bal} = 1$), úgy, hogy `T(bal) = HAMIS` és `T(jobb) = IGAZ`
- Vegyük észre, hogy az algoritmus akkor is helyesen működik, ha a sorozatban több mint egy „váltási pont” van.



Általánosított (diszkrét) bináris keresés

Pszeudokód

```
ALGORITMUS AltalanosBinKereses(bal, jobb, T)
  AMÍG (jobb - bal > 1)
    közép = (bal + jobb) / 2
    HA (T(közép))
      jobb = közép
    KÜLÖNBEN
      bal = közép
    VÉGE(Ha)
  VÉGE(Amíg)
VÉGE(Algoritmus)
```

Általánosított (diszkrét) bináris keresés
Példák

- A klasszikus bináris keresést úgy kapjuk az általánosból, hogy a tömb két végére strázsát állítunk (`a[0]` $= -\infty$, `a[n + 1]` $= \infty$), kezdetben `bal` $= 0$ és `jobb` $= n + 1$ és a tulajdonságot a következő módon vesszük fel: `T(i)` $=$ `a[i]` $\geq$ `keresett`
- A növekvő részsorozatokra bontásnál a `végződések` tömb mindkét végére strázsát állítunk, kezdetben `bal` $= 0$ és `jobb` $=$ `hányvégződés` $+ 1$ és `T(i) = végződések[i] < keresett`
- Mindkét esetben az algoritmus futása után a keresett index a `jobb` változóban lesz. Az első esetben ha `a[jobb]` $\neq$ `keresett`, akkor a keresett elem nem szerepel a tömbben és visszatéríthetünk -1-et. A második esetben ha `jobb = hányvéződés + 1`, új sorozatot kell kezdenünk.
- A DOMJudge `demo` versenyében a `boolfind` feladat megoldása is erre az elvre épül.
- A háziban több feladatban is „visszaköszönnek” majd a bináris keresés különböző alkalmazásai.

Hanoi tornyok

Feladat

Adva van három rúd, melyeket az A, B és C betűkkel jelölünk. Az elsőre fel van fűzve n darab különböző átmérőjű korong, úgy, hogy a korongok az átmérőjük csökkenő sorrendjében helyezkednek el egymás fölött. A másik két rúd üres. Költöztessük át a korongokat az első rúdról a másodikra a harmadik segítségével, úgy, hogy egyszerre csak egy korongot mozgassunk és csak kisebb átmérőjű korongot tehetünk egy nagyobb átmérőjű korongra. Írjuk ki a lépések sorozatát!

Példa: $n = 3$ esetén 7 lépésre van szükség

Az n korong átköltöztetése az A rúdról a B-re felbontható három, ehhez hasonló részfeladatra:

1. $n - 1$ korong áthelyezése az A rúdról a C-re
2. A megmaradt korong áthelyezése a B-re
3. $n - 1$ korong áthelyezése C-ről B-re

Hanoi tornyok

Pszeudokód

```
ALGORITMUS Hanoi(korongok, honnan, hova, segéd)
  Ha (korongok = 1) akkor
    KI: honnan -> hova
  KÜLÖNBEN
    Hanoi(korongok - 1, honnan, segéd, hova)
    Hanoi(1, honnan, hova, segéd)
    Hanoi(korongok - 1, segéd, hova, honnan)
  VÉGE(Ha)
VÉGE(Algoritmus)
```

Az algoritmus hívása `Hanoi(n, A, B, C)` alakú.

Tartalom

Fibonacci

- A *Rekurzió* fejezetnél láttuk, hogy az *n*. Fibonacci-szám kiszámítása rekurzívan exponenciális futási időhöz vezet, mivel ugyanazt az elemet többször is kiszámoljuk.
- Ennek tulajdonképpen az oka, hogy a részfeladatok nem függetlenek egymástól.
- Például $f(5)$ kiszámításához szükség van $f(4)$-re és $f(3)$-ra, de az $f(4)$ kiszámításához is szükség van $f(3)$-ra, vagyis az $f(5)$-ből leszármazó $f(4)$ és $f(3)$ között van „átfedés" (a teljes $f(3)$).
- Ez a probléma kiküszöbölhető különböző iteratív megoldásokkal (pl. a dinamikus programozás módszerével), vagy a *memoizálás* módszerének beépítésével rekurzióba (erről a későbbiekben lesz szó).

Úszómedence

Feladat

Egy tulajdonos szeretne egy úszómedencét építeni a kertjében. A kert téglalap alakú és bizonyos pontjain dísznövények találhatóak. A tulajdonos szeretné tudni, hogy mekkora lehetne a legnagyobb területű úszómedence, amelyet úgy építhetne, hogy egy növényt sem kell elköltöztetni az eredeti helyéről. A medence szintén téglalap alakú lesz és oldalai párhuzamosak a kertet körülvevő kerítéssel. Egy dísznövény maradhat a medence szélén is.

Példa: egy 10×10-es kertben található három dísznövény a $(3, 8)$, $(7, 7)$ és $(4, 4)$ koordinátákon.
A legnagyobb medence területe 42, bal alsó sarka a $(4, 0)$, jobb felső sarka a $(10, 7)$ pontban lesz.

- Ha a kertben nem lenne egyetlen dísznövény sem, a medence lefedhetné a teljes kertet.
- Ha a kertben egyetlen növény lenne, négy téglalapra oszthatnánk a területet az ezen áthaladó két merőleges egyenes mentén.
- Az így létrejött négy téglalap közül a legnagyobb területű lenne a megoldás.
- Innen jön az ötlet, hogy minden téglalap esetében döntsük el, hogy található-e benne dísznövény.
- Ha nem, egy lehetséges megoldásunk van, hiszen építhetünk úszómedencét az aktuális területre.
- Ha igen, felbontjuk a feladatot négy részfeladatra, amelyeket hasonlóan oldunk meg.

- Figyeljük meg a fenti megoldás gondolatmenetét: először a legegyszerűbb eset megoldását kerestük meg (nincs dísznövény), majd a következő legegyszerűbbét (egyetlen dísznövény van) és végül ezek alapján fogalmaztuk meg az általános feladat részfeladatokra bontásának módját.
- A fenti megoldással a probléma, hogy a négy terület nem független, vannak közöttük átfedések.
- Pontosabban a négy „sarokban” lévő kisebb téglalapok mindegyikére kétszer oldjuk meg a feladatot.

- Ezért ennek a feladatnak a megoldása exponenciális időt igényel a dísznövények számában.
- A feladat megoldható hatékonyan, az ehhez szükséges módszerek nem tartoznak az előadás anyagához.
- Ha a medence oldalai nem kell párhuzamosak legyenek a kert oldalaival, a feladat tovább bonyolódik, de ez a változat is megoldható polinomiális időben.

Tartalom

Összefésülésen alapuló rendezés (Mergesort)

- Emlékezzünk az `Összefésülés` programozási tételre!
- Ez két rendezett sorozatból állított elő lineáris időben egy olyan rendezett sorozatot, amely az összes elemét tartalmazta az eredeti két sorozatnak.
- Az előző feladat gondolatmenetén elindulva, megállapíthatjuk, hogy egyetlen elem mindig rendezett sorozatot alkot.
- Két elem vagy jó sorrendben van, vagy fel kell cserélnünk őket. Ez tulajdonképpen tekinthető a két, egyetlen elemből álló sorozat összefésülésének.
- Innen jön az általános ötlet, hogy osszuk fel a sorozatot két egyenlő hosszúságú részsorozatra, rendezzük ezeket, majd fésüljük őket össze.
- Az algoritmus futási ideje minden esetben $\Theta(n \log n)$, de mivel az összefésüléshez szükség van segédtömbre, nem dolgozik helyben, hanem lineáris memóriabonyolultságú. Könnyen implementálható úgy, hogy stabilan rendezzen.

Példa: [9 10 8 7 4]

Összefésülésen alapuló rendezés (Mergesort)

```
ALGORITMUS MergeSort(a, bal, jobb)
  HA (bal < jobb)
    közép = (bal + jobb) / 2
    MergeSort(a, bal, közép)
    MergeSort(a, közép + 1, jobb)
    Összefésül(a, bal, közép, jobb)
    //összefésüli az a[bal..közép] és a[közép + 1..jobb] sorozatokat
    //az eredményt az a[bal..jobb] pozíciókon tárolja
  VÉGE(Ha)
VÉGE(Algoritmus)
```

Hívás: `MergeSort(a, 1, n)`

Gyorsrendezés (Quicksort)

- Az eredeti sorozatot úgy rendezi, hogy két rendezendő részsorozatra bontja.
- A bal oldali részsorozat elemei kisebbek mint a jobb oldali részorozat elemei. Ezek között lehet egy harmadik sorozat, amely tartalmazhat egyetlen elemet, vagy több egyenlő elemet (a felosztási algoritmustól függően).
- Az elemet amely alapján a felosztás történik nevezzük őrszemnek vagy strázsának (angolul pivot). Ez határozza meg a helyet, ahol az adott tömb két részre oszlik, amely kulcskérdés az algoritmus végrehajtása során.
- A részsorozatok rendezése egymástól függetlenül történik.
- A részeredmények összerakása hiányzik.
- Az algoritmus helyben rendez, de nem stabil.
- Legrosszabb esetben futási ideje négyzetes, átlagesetben $\Theta(n \log n)$, a gyakorlatban gyorsabb mint a `Mergesort` véletlen bemenetekre.

Gyorsrendezés (Quicksort)

Pszeudokód

```
ALGORITMUS QuickSort(a, bal, jobb)
  HA (bal < jobb)
    m = Feloszt(a, bal, jobb)
    QuickSort(a, bal, m)
    QuickSort(a, m + 1, jobb)
  VÉGE(Ha)
VÉGE(Algoritmus)
```

- Gyakran a legbaloldalabbi elemet (`a[bal]`-t) vagy a legjobboldalabbit (`a[jobb]`-t) választjuk strázsának. Ekkor az algoritmus négyzetes időben fog futni, ha a kezdeti sorozat rendezett (növekvően vagy csökkenően).
- Választhatjuk a strázsát a középső elemnek is, ekkor az [1 2 3 4 5 4 3 2 1] vagy [5 4 3 2 1 2 3 4 5] alakú sorozatokra kapunk kedvezőtlen futási időt.
- Ha azt szeretnénk, hogy ne lehessen könnyen ellenpéldát fabrikálni az algoritmusunkra, megválaszthatjuk a strázsát véletlenszerűen. Még jobban működik a gyakorlatban, ha három véletlenszerű strázsa közül a középső érték mellett döntünk.
- Véletlenített strázsaválasztás esetén nem szükséges újraírni a `Feloszt` alprogramot, elég a véletlen strázsát felcserélni a megfelelő elemmel a `Feloszt` hívása előtt.

Gyorsrendezés (Quicksort)

A strázsa helye

- Látjuk, hogy a legjobb az lenne, ha a strázsa pont a sorozat mediánja lenne, mert ekkor két egyenlő részre hívhatnánk meg az algoritmust rekurzívan.
- Ahhoz, hogy ne váljon négyzetessé az algoritmus, tulajdonképpen elegendő annyi, hogy valamilyen konstans arányban osszuk fel a tömböt.
- Ezt használja ki a Blum–Floyd–Pratt–Rivest–Tarjan-algoritmus (röviden BFPRT), amely legrosszabb esetben kb. 30%-70% osztja fel a sorozatot.
- Az algoritmus determinisztikus és alapötlete, hogy ötös csoportokra osztja a tömb elemeit, minden csoportban valamilyen naív módszerrel (pl. beszúró rendezéssel) megkeresi a mediánt, majd az így kapott $\lfloor n/5 \rfloor$ medián mediánját keresi tovább rekurzívan.
- Ezért a módszert nevezik mediánok mediánjának is (lásd könyvészet, pl. CLRS vagy Jeff Erickson könyve).

Gyorsrendezés (Quicksort)

Felosztás

- A `Feloszt` alprogramot számos módon megvalósíthatjuk, ennek a megválasztása is befolyásolja az algoritmus hatékonyságát.
- A legismertebb a Hoare által eredetileg leírt módszer, melynek megfelelően elindulunk a tömb két szélső elemétől és felcseréljük egymás között azokat az elemeket, amelyek nagyobbak a strázsánál (és a tömb első részében vannak), azokkal, amelyek kisebbek, mint a strázsa (és a tömb második részében vannak).
- Ahol ez a bejárás véget ér, ott osztjuk két részre a tömböt.

Felosztás (Hoare)
Pszeudokód

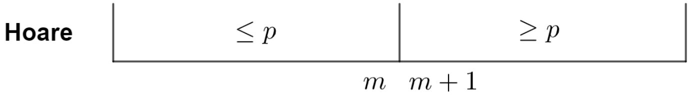
```
ALGORITMUS Feloszt(a, bal, jobb)
  strázsa = a[bal]
  i = bal - 1
  j = jobb + 1
```

```
  ISMÉTELD
    ISMÉTELD
      j = j - 1
    AMEDDIG (a[j] <= strázsa)
    ISMÉTELD
      i = i + 1
    AMEDDIG (a[i] >= strázsa)
    HA (i < j) akkor
      Csere(a[i], a[j])
    VÉGE(Ha)
  AMEDDIG (i >= j)
  VISSZATÉRÍT: j
VÉGE(Algoritmus)
```

Példa: [8 7 6 10 4 11 2 5 9]

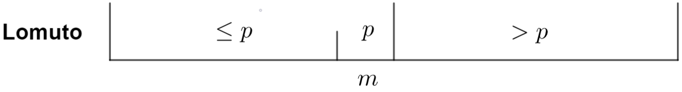
Felosztás (Hoare)

Potenciális buktatók implementáláskor



- Mivel a strázsa végső helyéről nem tudunk semmit, fontos, hogy a két rekurzív hívással lefedjük a teljes sorozatot, ne hagyjuk ki a „középső” *m*. elemet.
- Ellenkező esetben bizonyos esetekre a sorozatot nem rendezné helyesen az algoritmus (pl. [4 2 1 1 5]).
- Ahhoz, hogy a végtelen rekurziót elkerüljük, a felosztás mindkét fele kell tartalmazzon legalább egy elemet.
- Részben emiatt vettük be az egyenlőséget mindkét belső ciklus megállási feltételébe.
- Ugyancsak emiatt nem választhatjuk strázsának az utolsó elemet.

Felosztás (egy kompaktabb, de kevésbé hatékony változat)



- A következő Lomuto-féle változat rövidebben implementálható, viszont véletlen bemenetekre kb. háromszor annyi felcserélést végez.
- A strázsa ezúttal az utolsó elem lesz.
- Ez a módszer az őrszemet nem teszi be egyik keletkező résztömbbe se (vagyis ez mindig „a helyén lesz”), így az első rekurzív hívás `QuickSort(a, bal, m)` helyett `QuickSort(a, bal, m - 1)` kell legyen.
- Ellenkező esetben végtelen rekurzió állhat elő, például ha a sorozat elemei egyenlőek.

Felosztás (Lomuto)
Pszeudokód

```
ALGORITMUS Feloszt(a, bal, jobb)
  strázsa = a[jobb]
  i = bal - 1
  MINDEN j = bal, jobb - 1 végezd el:
    HA (a[j] <= strázsa) akkor
      i = i + 1
      Csere(a[i], a[j])
    VÉGE(Ha)
  VÉGE(Minden)
  Csere(a[i + 1], a[jobb])
  VISSZATÉRÍT: i + 1
VÉGE(Algoritmus)
```

Példa: [8 7 6 10 4 11 2 5 9]