ADATSZERKEZETEK ÉS ALGORITMUSOK

Négyzetes rendezők, Gyorsrendező, Kupacrendező

A gyorsrendezés lényege

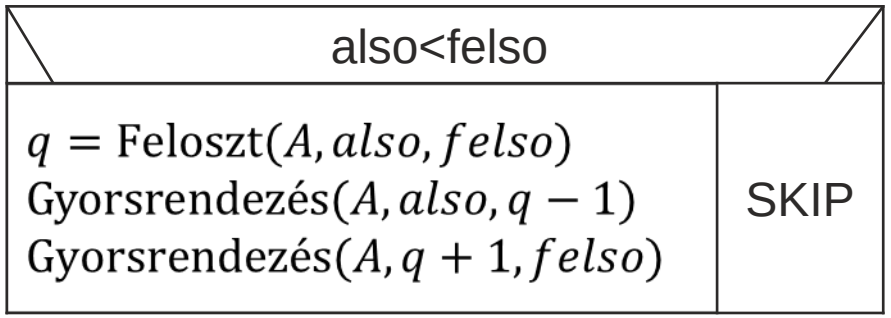
- Az algoritmus két alaplépése:
  - Elsőként egy előre kiválasztott elemhez (pivot) képest rendezzük a tömb elemeit (nagyobbak a jobb oldalon, kisebbek a bal oldalon),
  - Majd újrakezdjük a rendezést az így kapott két oldalon.
- Rendezés általában $O(n \cdot \log_2(n))$ lépés alatt.

Miért jó?

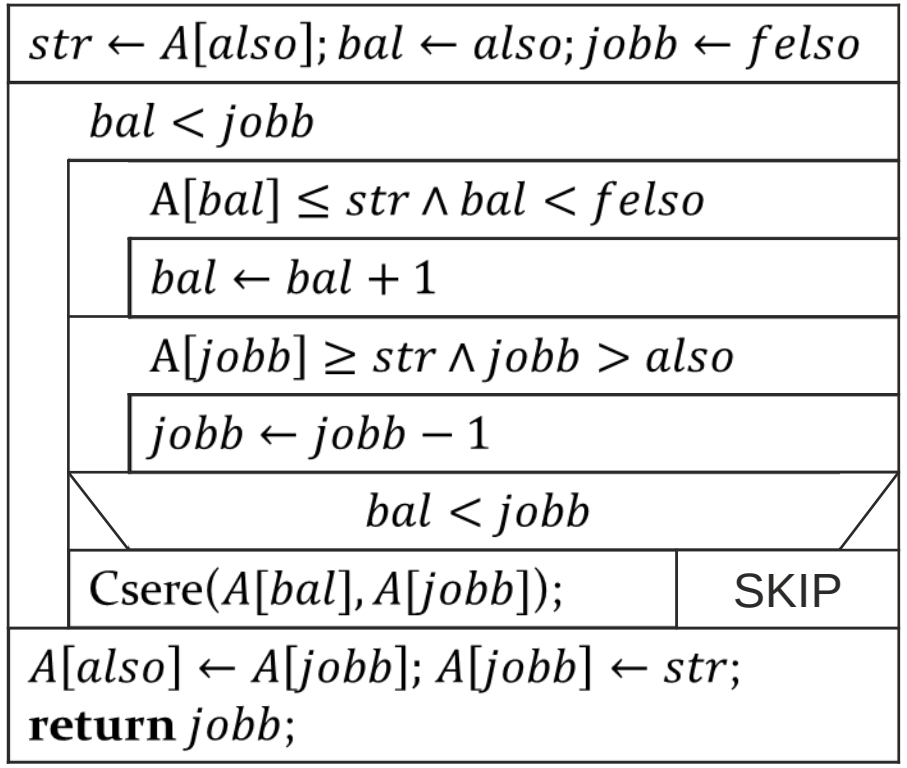
- Nem ”túl” bonyolult?
- Igaz, hogy legrosszabb esetben a futásidő $O(n^2)$, de átlagos esetben $O(n \cdot \log_2(n))$.
- A legrosszabb eset nagyban függ a pivot választástól, így például random pivot választással a legrosszabb eset előfordulási valószínűsége csökkenthető.
- Könnyen párhuzamosítható, mert a kezdeti szétválasztás után a jobbra illetve balra került elemek rendezését végezhetjük egyszerre.

Gyorsrendezés algoritmusa

`Gyorsrendezés (A, also, felso)`



`Feloszt(A, also, felso)`



Órai feladat

- Futtassuk le a kiadott kódot!
- Értékeljük az eredményeket. Összhangban van az előadáson hallottakkal?
- A „quicksort.hpp”-ban implementált gyorsrendező rendezett elemekre $O(n^2)$ idő alatt fut le. Mi lehet ennek az oka? Mit lehet tenni, hogy gyorsabb legyen?
- Módosítsuk a quicksort implementációt, hogy rendezett elemek esetén is $O(n \cdot \log_2(n))$ komplexitású legyen (legalábbis nagyságrendekkel gyorsabb, mint jelenleg).

Órai kódolás

- Használjuk fel a kupacot rendezésre!
- Hasonlítsuk össze a kupacrendezés futásidejét a gyorsrendezőével!
  - Vizsgáljuk meg, hogy a gyakorlatban mi történik a legrosszabb esetben

std::sort

- A C++ standard könyvtárának részét képező rendezőfüggvény.
- Garantáltan legfeljebb $O(n \cdot \log_2(n))$ bonyolultságú, de nem gyorsrendezés.
- Használat:
- `#include <algorithm>`
- `sort(begin, end, comparator);`
- Ahol `begin` a sorozat első elemére, `end` az utolsó eleme mögé mutató iterátor, `comparator` pedig az elemek összehasonlítására használt függvény. A *`comparator`* paraméter elhagyható, ekkor az algoritmus a `<` operátort fogja használni.
- További megkötés, hogy `begin` és `end` „random access” iterátorok legyenek. Ehhez tudnunk kell az iterátort egyetlen lépésben tetszőleges távolságra léptetni. (A láncolt lista vagy keresőfa iterátorok például ezt nem teljesítik).

Gyakorló feladat G08F01

- Adott egy bemeneti fájl, ami mérésadatok listáját tartalmazza:
  - helység;eszközNeve;időkód;maximumFogyasztás;dimmerArány;áramkörSzáma
  - (string;string;long;double;double;integer)

  formátumban, soronként egy mérésadat szerepel
- Olvasd be a fájl tartalmát egy `vector`-ba.
- Készítsd el a szükséges összehasonlító operátorokat, hogy a mérésadatokat az időkód alapján tudjad rendezni.
- Rendezd a ma tanult rendezőkkel úgy, hogy közben méred a rendezések szükséges idejét, majd jelenítsd meg az eredményeket a konzolon.
- Két-két rendezés között ne felejtsd el megkeverni az elemeket a `vector`-ban.
- Majd kiíratással ellenőrizd, hogy az adatok rendezettek-e.

Gyakorló feladat G08F02

- Implementál egy olyan függvényt, ami megkeresi egy *n* elemű tömb *k*-adik legkisebb elemét úgy, hogy közben átrendezi az elemeket.
- A feladatra **két** megoldást kell készíteni. Az elsőnek $O(k \cdot n)$ időn belül kell megoldást adnia (ezt érdemes a kiválasztásos vagy a buborékrendezésre alapozva elkészíteni), a másodiknak átlagosan $O(n)$ idő alatt (ehhez a gyorsrendezést kell átírni).
- Mérd le és vesd össze a két algoritmus átlagos futásidejét különböző méretű tömbök esetén. **Egy tömbméretre nem elég egyetlen mérést végezni.**

Gyakorló feladat G08F03

- Az alap gyorsrendező tovább fejlesztése különböző pivot választási stratégiákkal a tömbrészlet
  - első eleme.
  - középső eleme.
  - alsó, középső, felső eleme közül az átlagos értékű.
  - egy véletlen eleme.
  - három különböző véletlen eleme közül az átlagos értékű.
- A gyorsrendező meghívásakor legyen lehetőség egy paraméter segítségével kiválasztani, hogy melyik stratégia szerint válasszunk pivotot.
- Hajts végre különböző rendezettségű (növekvő, csökkenő, random) tömbökön, különböző pivot választási stratégiát használó gyorsrendezéseket és figyeld meg az időigényt.

Rendezés beépített függvénnyel

```
template <class RandomAccessIterator>
void sort (RandomAccessIterator first, RandomAccessIterator last);

template <class RandomAccessIterator>
void stable_sort (RandomAccessIterator first, RandomAccessIterator last);
```

- Mindkét függvény növekvő sorrendbe rendezi az iterátorok által megadott tartományon [first, last) szereplő elemeket.
- Használatukhoz szükség van a `#include <algorithm>` könyvtárra, valamint a szokásos `std` névtérben találhatóak.
- A `sort` nem stabil rendező, nem őrzi meg az azonos elemek eredeti sorrendjét.
- A `stable_sort` megőrzi az azonos elemek eredeti sorrendjét.

- További információk: https://en.cppreference.com/w/cpp/algorithm/sort

Gyakorló feladat G09F01

- Írjunk egy interaktív konzolos programot, amellyel fontossági sorrendbe tudjuk állítani teendőinket.
- A programnak a következő menüpontjai legyenek:
  - Feladat hozzáadása prioritással
    - A prioritást egy 1-től 100-ig növekvő skálán lehessen megadni
  - Legfontosabb feladat elvégzése
    - Ekkor írassuk ki az elvégzendő feladatot
  - Feladatok hozzáadása fájlból
    - Itt egy fájl nevét kell megadni, melyben sorokban tároljuk a feladatok nevét és prioritását
    - A fájl sorainak formátuma legyen: **string;int**
    - Ezeket a már meglévő feladatokhoz kell hozzáadni
  - (Kilépés a programból)

Gyakorló feladat G09F02

- 1) Módosítsd a megírt kupacot úgy, hogy egy template paraméterként átadott `bool` paraméter segítségével lehessen meghatározni, hogy maximum-kupaccal vagy minimum-kupaccal szeretnénk dolgozni (a kupac tetején a legkisebb elem foglal helyet, a csomópontok gyerekei nagyobb értékűek a szülőjüknél.)

  •

  Törekedj effektív kódolásra, lehetőség szerint kerüld a kódismétlést!

- 2) Alakítsd át a bináris kupacot ternáris kupaccá, azaz minden csúcsnak maximum 3 gyereke lehet az eddigi kettő helyett! (pl: bal, középső, jobb gyerek) Őrizd meg a balra tömörített, majdnem teljes tulajdonságot!

Gyakorló feladat G09F03

- Készíts alkalmazást, ami összehasonlítja a gyors- vagy kupac- és a beszúró rendezést úgy, hogy generál különböző méretű véletlen számokból álló tömböt, és számolja az összehasonlításokat és cseréket mindkét esetben.
  - Vizsgáld meg a szélsőséges helyzetek esetén, hogy hogyan viselkednek az algoritmusok
  - Vizsgáld meg 1000, 10000, 100000 méretű tömbökre