10. Gyakorlat

Lineáris idejű rendezések

Témák:

- Radix rendezés
  - Listán
  - Tömbön (leszámláló rendezést használva számjegyek szerint)
  - Bináris radix rendezések (opcionális tananyag)
- Edény rendezés (bucket sort)

Tudjuk, hogy egy olyan rendezés, ami kizárólag az elemek összehasonlítása alapján dönt a sorrendjükről, a legkedvezőtlenebb esetben legalább n lg n összehasonlítást végez, ezért az összes lehetséges bemenetet tekintve: $Mt(n) \epsilon\, \Omega(n \lg n)$.

Ehhez kapcsolódó tételek az előadás jegyzetből:

Tétel: Tetszőleges rendező algoritmusra $mT(n) \epsilon\ \Omega(n)$.

Tétel: Bármely összehasonlító rendezés végrehajtásához a legrosszabb esetben $MC(n) \epsilon\ \Omega(n \lg n)$ kulcsösszehasonlítás szükséges.

Tétel: Tetszőleges összehasonlító rendezésre $MT(n) \epsilon\ \Omega(n \lg n)$.

Megjegyzés: ezek a tételek csak az utolsó előadások valamelyikén lesznek, így most még nem kell tudniuk, de fogják tanulni.

Lehet-e más elven működő algoritmussal ezen javítani? Ötlet (radix rendezésekhez): egész számok rendezéséhez használjuk ki a helyiértékeket!

Radix rendezés

Továbbiakban:

- d ~ helyiértékek száma
- r ~ számrendszer alapja

Helyiértékek sorszámai (d=5, r=4, kulcs:10320):

5	*4*	*3*	*2*	*1*
1	**0**	**3**	**2**	**0**

Kezdetleges algoritmus (előadáson szerepel):

radixSort(A : E2*, d : ℕ)

i := 1
i ≤ d
stabil rendezéssel rendezzük a listát az i. helyiérték alapján
i++

A legkevésbé szignifikánstól a legszignifikánsabbig minden helyiérték szerint rendezzük a listát egymás után egy stabil rendezéssel. Ez $\Theta\big(d * rendezés(n)\big)$ költségű, ahol a $rendezés(n)$ a belső rendezésünk költsége.

Ismerünk is egy $\Theta(n \lg n)$ műveletigényű stabil rendezést, az összefésülő rendezést. Azzal viszont $\Theta(d * n \lg n)$ műveletigényű lesz az algoritmus, szóval nem nyerünk semmit. Vagy mégis?

Ötlet: a belső stabil rendezés használja ki, hogy kevés lehetséges értéke lehet a helyiértékeknek!

A listán egyszer végig menve szétosztjuk a listaelemeket a helyiértékük szerint d darab listába. A listák első elemire mutató pointereket egy *d* hosszúságú tömbben helyezzük el: ha *i* áll az adott helyiértéken, akkor az i. lista **végére** rakjuk (így marad stabil). Ezután összefűzzük a listákat a tömbből, és kész is a rendezés egy helyiérték szerint. Fontos részlete az algoritmusnak, hogy a lista végére konstans időben lehessen az új elemeket ráfűzni, ekkor kaphatunk lineáris műveletigényt:

A belső rendezés műveletigénye $\Theta(n + r)$, így az egész rendezésé pedig $\Theta\big(d(n + r)\big)$. Mivel azonban mind az *r*, mind a *d* konstans, a költség átírható $\Theta(n)$-re.

Az algoritmus az L = *<101, 013, 310, 323, 003, 220, 211>* listán.

1. Menet:

   0 - <*31**0**, 22**0***>

   1 - <*10**1**, 21**1***>

   2 - <>

   3 - <*01**3**, 32**3**, 00**3***>

   Összefűzve: *<310, 220, 101, 211, 013, 323, 003>*
   -eredmény lista az utolsó számjegy szerint rendezett-

2. Menet:

   0 - <*1**0**1, 0**0**3*>

   1 - <*3**1**0, 2**1**1, 0**1**3*>

   2 - <*2**2**0, 3**2**3*>

   3 - <>

   Összefűzve: *<101, 003, 310, 211, 013, 220, 323>*
   -számjegy szerinti listákban a kulcsok megtartják a utolsó számjegy szerinti rendezettségüket (stabilitás), eredmény lista az utolsó két számjegy szerint rendezett lett-

3. Menet:

   0 - <***0**03, **0**13*>

   1 - <***1**01*>

   2 - <***2**11, **2**20*>

   3 - <***3**10, **3**23*>

   Összefűzve: *<003, 013, 101, 211, 220, 310, 323>*
   -eredmény lista rendezett-

Algoritmus (jegyzetben megtalálhatjuk):

L egy C2L típusú lista, a listaelemek kulcsai *d* számjegyű, *r* alapú számrendszerben felírt számok. Szükségünk van még egy függvényre, amely megadja a kulcs i-dik helyiértéken álló számjegyét, ha a számrendszerünk alapja *r*, ez lesz a digit(i, r: $\mathbb{N}$, k:T) függvény.

radixSort(L : E2*, d, r : ℕ)

```
HeadZ : E2[r]   //nullától indexelt, fejelemekből álló tömb
Bz : E2*[r]     //fejelemekre mutató pointerekből álló tömb
i = 0  to  r
    Bz[i] := &HeadZ[i]
i = 1  to  d
    distribute(L, i, Bz)
    gather(Bz, L); i++
```

distribute(L : E2*, i : ℕ, Bz : E2*[])

```
L->next ≠ L
    p := L->next; out(p)
    precede(p, Bz[digit(i, r, p->key)])
```

gather(Bz : E2*[], L : E2*)

```
i := 0
i < Bz.M
    append(L, Bz[i])
    i++
```

append(L : E2*, B : E2*)

```
B->next ≠ B
    p, q, r := L->prev, B->next, B-prev
    p->next, q->prev := q, p
    r->next, L->prev := L, r
    B->next := B->prev := B
```

Ha a bemenetünk egy tömb, hogyan valósíthatjuk meg a radix rendezést?

Leszámláló rendezés

A módszerhez két tömbre lesz szükségünk, legyenek ezek A és B, kezdetben A-ban vannak a rendezendő kulcsok, B lesz az eredmény tömb. A módszer akkor használható, ha a kulcsok egészek, r féle értéket vehetnek fel. Tegyük fel, hogy a [0..r-1] intervallumba esnek, továbbá r<<n.

Két menetből áll: első menetben egy számláló tömbben, legyen ez a Z[0..r-1] leszámoljuk, hogy melyik kulcsból mennyi van. A leszámlálás után a számláló tömböt összesítjük: Z[0] nem változik, Z[i] (ha, i>0) akkor a Z[0]+..+Z[i] összeget fogja tartalmazni, azaz Z[i] azt adja meg, hogy hány kulcs esett a [0..i] tartományba összesen (kumulált gyakoriság).

Második menetben az A tömbből helyükre másoljuk a rendezendő rekordokat. Ha A[i] = j, akkor Z[j] adja meg, hogy hova kerül az A[i] a rendezett sorozatban. Hogy stabil rendezést kapjunk, az A tömböt visszafelé: A.M downto 1 irányú ciklussal járjuk be. Miután A[i]-t helyre tettük, a megfelelő Z-beli számlálót eggyel csökkenteni kell.

```
counting_sort(A, B : 𝔗[] ; r : ℕ ; φ : 𝔗 → 0..(r−1))
Z : ℕ[r] // counter array
k = 0 to r−1
    Z[k] = 0 // init the counter array
i = 1 to A.M
    Z[φ(A[i])]++ // count the items with the given key
k = 1 to r−1
    Z[k] += Z[k − 1] // Z[k] = the number of items with key ≤ k
i = A.M downto 1
    k = φ(A[i]) // k = the key of A[i]
    B[Z[k]] = A[i] //Let A[i] be the last of the {unprocessed items with key k}
    Z[k] − − // The next one with key k must be put before A[i]
```

Megjegyzés: a fenti, jegyzetből kimásolt algoritmus egy φ : T→[0..r-1] függvényt használ, mely leképezi A[i] kulcsot a számláló tömb indexeire.

A radix rendezéshez ezt használjuk rendre a kulcsok k=1...d számjegyeire. Egy számjegy szerinti rendezés két menetből áll:

Első menetben megszámoljuk, hogy a kulcs *k*-adik helyiértékén melyik számjegyből mennyi van.

Majd a Z számláló tömbön végrehajtjuk az összesítést: Z[i]=Z[i-1]+Z[i] i=1..r-1., ekkor Z[i] azt mutatja, hogy 0..*i* számjegyekből összesen mennyi volt a kulcsok *k*-adik helyiértékén.

A második menetben kerülnek helyre a kulcsok. Az A tömbből a B tömbbe másoljuk át őket. Ha A[i] kulcs *k*-dik számjegye *j*, akkor Z[j] adja meg a kulcs helyét a B tömbben. Ha több kulcsban is *j* áll a *k*-adik helyiértéken, akkor az utolsónak a helyét mutatja Z[j]. Ezért helyre rakásnál az A tömböt fordítva, A.M .. 1 irányban járjuk be. Ha A[i] átmozgatását elvégeztük, Z[j]-t eggyel csökkenteni kell:

counting_sort(A, B : ℕ[], r, k : ℕ)

```
Z : ℕ[r]
i = 0 to r-1
    Z[i] := 0
i = 1 to A.M
    Z[digit(k, r, A[i])]++
i = 1 to r-1
    Z[i] = Z[i] + Z[i - 1]
i = A.M downto 1
    j := digit(k, r, A[i])
    B[Z[j]] := A[i]
    Z[j]--
```

A műveletigény $\Theta(n + r)$, de mivel az r konstans, $\Theta(n)$, így az ezt a stabil rendezést használó radix is lineáris lesz.

Szemléltessük a *<11, 20, 10, 23, 21, 30>* tömbön a radix rendezést, belül leszámláló rendezést használva. Megjegyzések a szemléltetéshez: fontos, hogy két tömbös az algoritmus, jelöljük mindig, hogy melyik tömbből, melyikbe másoljuk át a kulcsokat! A táblázat a *Z* tömb értékeit mutatja a megnövelés, illetve a csökkentés után. A stabilitás miatt fontos, hogy amikor helyükre másoljuk az elemeket, fordított sorrendben haladjunk:

A=*<11, 20, 10, 23, 21, 30>*

Leszámlálás a második számjegy szerint:

		1**1**	2**0**	1**0**	2**3**	2**1**	3**0**			30	21	23	10	20	11
0	0		1	2			3	3	3	2			1	0	
1	0	1				2		2	5		4				3
2	0							0	5						
3	0				1			1	6			5			
Helyre rakáskor a Z tömbből olvassuk ki, hova kell tenni az elemet, majd csökkentjük eggyel Z megfelelő értékét.															

B=	1	2	3	4	5	6
	20	10	30	11	21	23

B= *<20, 10, 30, 11, 21, 23>*

Leszámlás az első számjegy szerint:

		20	**1**0	**3**0	**1**1	**2**1	**2**3			23	21	11	30	10	20
0	0							0	0						
1	0		1		2			2	2			1		0	
2	0	1				2	3	3	5	4	3				2
3	0			1				1	6				5		

A=	1	2	3	4	5	6
	10	11	20	21	23	30

A=*<10,11,20,21,23,30>*

Páros darabszámú számjegy esetén a rendezendő számok épp abban a tömbben vannak, amelyikben eredetileg voltak.

Radix rendezések bináris, nem negatív számokon.

Ez azanyagrész nincs benne a jegyzetben, vizsgára nem kell tudni, csak ha van időnk, akkor érdemes foglalkozni vele. Kétféle változata van. (1) bináris radix „visszafelé”, két tömbbel (2) bináris radix „előre”, egy tömbös, cserélgetős.

Bináris radix két tömbbel, a helyiértékeket „visszafelé” a legkisebbtől a legnagyobb felé dolgozzuk fel. *(Listás radixhoz hasonló, de tömbben dolgozik)*

Amennyiben bináris számokat rendezünk, mivel csak kétfelé kell osztani a számokat, két „edény” kell a szétosztályozáshoz. Megtehetjük azt, hogy a nullások a tömb elejétől: az 1-es indextől növekvőleg, az egyesek pedig a tömb végétől csökkenőleg lesznek elhelyezve. Így haladunk a legkisebb helyiértéktől visszafelé, ügyelve arra, hogy elsőként mindig a nullás edényt dolgozzuk fel 1-től növekvőleg, majd az egyes edényt n-től csökkenőleg. (Ez a feldolgozási irány biztosítja a stabilitást.)

Példa rendezése a *<101, 001, 011, 110, 010, 111, 000>* tömbnek:

A	B	B	B	B	B	B	B
10**1**				110	110	110	11**0**
00**1**					010	010	01**0**
01**1**							00**0**
11**0**						111	11**1**
01**0**			011	011	011	011	01**1**
11**1**		001	001	001	001	001	00**1**
00**0**	101	101	101	101	101	101	10**1**

	B	A	A	A	A	A	A	A
1.	**1**10			000	000	000	000	**0**00
	010				101	101	101	**1**01
	000					001	001	**0**01
2	**1**11							**1**11
	011						011	**0**11
	001		010	010	010	010	010	**0**10
	101	110	110	110	110	110	110	**1**10

	A	B	B	B	B	B	B	B
1	**0**00	000	000	000	000	000	000	000
	101			001	001	001	001	001
	001					010	010	010
2	**1**11						011	011
	011							111
	010				110	110	110	110
	110		101	101	101	101	101	101

Utolsó menet eltérő, a 0-s edényt átírjuk, az 1-es edényt megfordítjuk:

	B	A	A	A	A	A	A	A
1	000	000	000	000	000	000	000	000
	001		001	001	001	001	001	001
	010			010	010	010	010	010
	011				011	011	011	011
2	111					101	101	101
	110						110	110
	101							111

Bináris radix egy tömbbel, a helyiértékeket a legnagyobbtól a legkisebb felé dolgozza fel, egy meneten belül cserélgeti az elemeket.

Nem negatív bináris számokat úgy is rendezhetünk radix módszerrel, hogy a legnagyobb helyiértékkel kezdünk, cserélgetésekkel átalakítjuk úgy, hogy a 0 kezdő bitesek elől legyenek, az 1 kezdő bitesek pedig a tömb második felében. Ehhez a tömb elejétől indulva keresünk egy 1-es bittel kezdődő számot, majd a tömb végéről indulva keresünk egy 0-s bittel kezdődőt, ha a két kapott index különbözik, felcseréljük őket. Folytatjuk, amíg össze nem ér a két index. Ekkor a tömb elején már csupa 0 kezdőbites szám lesz, a tömb végén pedig az 1-es bittel kezdődők. Rekurzívan folytatjuk ugyanezt a nullás és egyes edényre.

Példa rendezése a *<101, 001, 011, 110, 010, 111, 000>* tömbnek (vizsgált bit félkövér, E0-nullával kezdődők, E1-egyessel kezdődők…):

101	000	000	E0	000	E00	000	000	000
001	001	001		001		001	001	001
011	011	011		011	E01	011	011	010
110	110	010		010		010	010	011
010	010	110	E1	110	E10	101	101	101
111	111	111		111	E11	111	111	110
000	101	101		101		110	110	111
csere	*csere*			*csere*			*csere*	*kész*

A rendezés algoritmusát legegyszerűbben rekurzívan készíthetjük el. Amikor az aktuális bit szerint átrendezzük a tömböt, rekurzívan meghívjuk a keletkezett 0-s és 1-es bites részekre.

(Esetleg hf-nek is feladható, ha előtte gyakorlaton lejátsszuk):

binary_counting (A: $\mathbb{N}_2$[], ah, fh, b : $\mathbb{N}$)

```
ah ≥ fh v b < 1
    (igaz ág: SKIP)
    (hamis ág:)
    i0, i1:= ah, fh
    i0 ≤ i1
        i0 ≤ i1 ∧ bit(A[i0], b) = 0
            i0++
        i0 ≤ i1 ∧ bit(A[i0], b) = 1
            i1++
        i0 < i1
            csere(A[i0], A[i1])
            i0, i1 := i0 + 1, i1 – 1
            (különben: SKIP)
    binary_counting(A, ah, i0 - 1, b - 1)
    binary_counting (A, i0, fh, b - 1)
```

- Hívása: binary_counting(A,1,A.M,d)
- bit(e,i) függvény megadja egy „e" egész szám i-edik bitjét (egy „és" művelettel és a megfelelő kettő hatvánnyal könnyen kinyerhető)

Idáig jó, de mi van, ha mondjuk valós kulcsokkal kell dolgozni? Ki tudjuk ezt a számlálós módszert terjeszteni?

Edényrendezés

Az edényrendezés [0,1) közötti valós kulcsokkal dolgozik, annyi edénybe szétosztva őket, amennyi listaelem van. Ha szerencsénk van, akkor egyenletesen oszlanak el, és minden edényben csak egy elem lesz. Ha nem, akkor amiben több van, azt majd le kell rendezni. A műveletigénye így (összefésülő rendezést használva) $O(n\,logn)$, $\Omega(n)$.

Jegyzetben található algoritmus az alábbi (ezt nem kell felírni):

bucket_sort(L : list)

n = the length of L	
Z : list[n] // Create the buckets $Z[0..(n-1)]$	
$j = 0$ to $(n-1)$	
	Let $Z[j]$ be empty list
$L \neq \emptyset$	
	Remove the first element of list L
	Insert this element according to its key k into list $Z[\lfloor n * k \rfloor]$
$j = 0$ to $(n-1)$	
	Sort list $Z[j]$ nondecreasingly
Append lists $Z[0], Z[1], \ldots, Z[n-1]$ in order into list L	

Készítsük el az algoritmust egyirányú egyszerű listára (erre a lista típusra a jegyzet tartalmazza a merge-sort algoritmust):

bucket_sort(&L : E1*)

n := length(L)	
Z : E1*[n] //edények listáinak első elemére mutató pointerek tömbje	
j = 0 to n-1	
	Z[j] := 0;
L ≠ 0	
	p, L := L, L->next
	j:= ⌊n * p->key⌋
	p->next:=Z[j] Z[j]:=p //lista elejére fűzünk
j = 0 to n-1	
	sort(Z[j])
appendLists(L,Z)	

Az algoritmus feltételez listaműveleteket, ezek:

- length(L : E1*): ℕ **-** a lista hossza (szerepel a jegyzetben, nem érdemes felírni)
- sort(L : E1*) - lerendezi a listát egy stabil rendezéssel, mergeSort(&L:E1*) használható, jegyzetben benne van.
- appendLists(&L:E1*, Z:E1*[ ]) – ez feladható hf-nek. Össze kell rakni a listát *L*- egyszerű listába: egymás után kell fűzni a *Z*-ben található lista darabokat. Vigyázzunk, hogy a műveletigény lineáris maradjon!

Az algoritmus lejátszása a *<0.79, 0.13, 0.16, 0.64, 0.39, 0.20, 0.89, 0.53, 0.71, 0.42>* listán:

	Beszúrás után		Rendezés után	
0	<>		<>	
1	<0.13>	<0.16, 0.13>	<0.13, 0.16>	*Amelyik listába több kulcs is kerül, ott mindig a lista elejére szúrjuk be az új elemet, majd lerendezzük a listákat.*
2	<0.20>		<0.20>	
3	<0.39>		<0.39>	
4	<0.42>		<0.42>	
5	<0.53>		<0.53>	
6	<0.64>		<0.64>	
7	<0.79>	<0.71, 0.79>	<0.71, 0.79>	
8	<0.89>		<0.89>	
9				

Végeredmény: *<0.13, 0.16, 0.20, 0.39, 0.42, 0.53, 0.64, 0.71, 0.79, 0.89>*

Hogyan módosíthatnánk az algoritmust, ha a kulcsok valós számok, de nem a [0,1) intervallumból valók?

Házi feladatok:

(1) Elkészíthetik a bináris radix rendező algoritmusokat, ha vettük az órán.
(2) Feladhatjuk a bucket_sort-nál tárgyalt appendLists algoritmust.
(3) érdekes, gondolkodtató a következő feladat:

Kis n-ekre kell készíteni olyan speciális algoritmusokat, melyek legrosszabb esetben is legfeljebb $log_2 n!$ összehasonlítást használnak. (Tudjuk, hogy ez az alsó korlátja *MC(n)*-nek)

Az első pár n-re mit ad a képlet:

n	n!	$\lceil \log_2(n!) \rceil$
2	2	1
3	6	3
4	24	5
5	120	7
6	720	10
7	5040	13
8	40320	16
9	362880	19
10	3628800	22

Három elem esetén legfeljebb 3 összehasonlítás végezhető, ilyen algoritmus könnyen adható:

- A számok legyenek *(a), (b), (c)*. összehasonlítjuk az (a)-t és a (b)-t (így tudjuk mi a min((a), (b), és a max((a),(b))). Azután összehasonlítjuk a max((a),(b))-t és a (c)-t. Ha a (c) nagyobb, akkor a sorrend <min((a), (b)), max((a),(b)), (c)>. Különben össze kell hasonlítani a min((a), (b))-t és a (c)-t. Ha a (c) kisebb, akkor a sorrend <(c), min((a), (b)), max((a), (b))>, ha pedig nagyobb, akkor <min((a),(b)), (c), max((a),(b))>.

Négy elem estén legfeljebb 5 összehasonlítást használhatunk. Többféle ötlet is működik. Egyik legegyszerűbb megoldás:

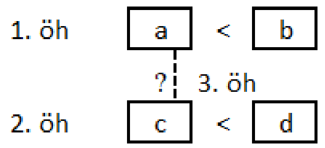
- Rendezzünk le 3 elemet, erre legfeljebb 3 összehasonlítást kell használnunk, még maradt 2 db. A negyedik számot helyezzük el az előbb kapott rendezett számsorba, úgy, hogy elsőként a középsővel hasonlítjuk össze, majd az eredménytől függően a legkisebbel, vagy a legnagyobbal, erre épp elég a megmaradt 2 összehasonlítás.

Ami már nehezebb: öt elemünk van, és legfeljebb 7 összehasonlítást használhatunk. Ezt lehet feladni házinak.

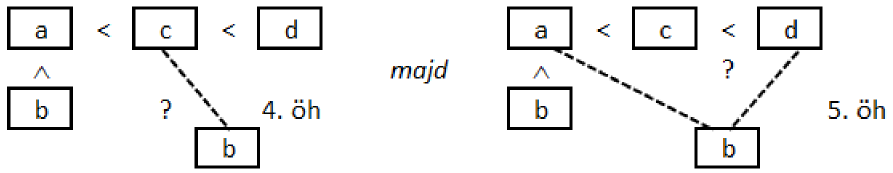
Megoldás:

Tegyük fel, hogy az öt elem mind különböző, a módszer a következő lesz:

- Összehasonlítunk, két-két elemet, ehhez két összehasonlítást használunk.
  Tegyük fel, hogy az alábbi a<b és c<d relációkat kapjuk.
- Majd összehasonlítjuk a két kisebb elemet:

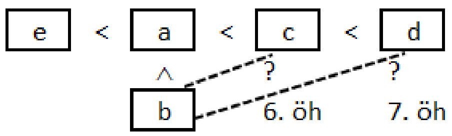


- Tegyük fel, hogy a < c, (hasonló elrendezést kapunk c<a esetén)
  A 4. és 5. összehasonlítással beillesztjük az ötödik elemet az (a,c,d) rendezett sorrendbe. Ehhez két összehasonlítás elegendő, ha elsőként a középső elemmel, majd a bal-, vagy jobb szélen levővel hasonlítunk össze.

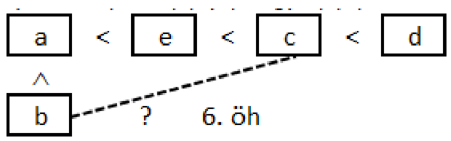


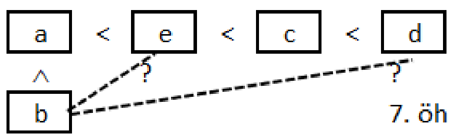
- Két, a megoldás folytatása szempontjából különböző elrendeződés jöhet létre:

Első elrendeződés, ha az ötödik elem a legkisebb, ekkor a rendezés befejező lépései:



Második elrendeződés akkor jön létre, ha az ötödik elem nagyobb „a”-nál. Ekkor három a-nál nagyobb elemünk van, amik már rendezettek. Hogy az előbb beillesztett „e” pontosan hova került, az lényegtelen, így tegyük fel, hogy az alábbi relációk teljesülnek. A rendezés befejező lépései az alábbiak:





Hasonlóan két összehasonlítás elegendő, ha a<c<e<d, vagy a<c<d<e a sorrend.

Készült az „Integrált kutatói utánpótlás-képzési program az informatika és számítás-tudomány diszciplináris területein” című EFOP 3.6.3-VEKOP-16-2017-00002 azonosítójú projekt támogatásával.