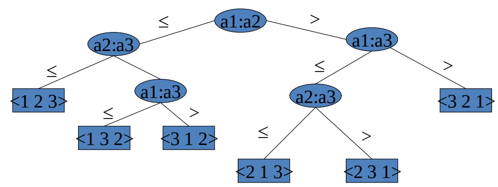
Lineárne usporadúvanie

dolné ohraničenie na usporadúvania porovnávaním

model rozhodovacieho stromu

predstavuje porovnania, ktoré vykoná usporadúvací algoritmus nad vstupom daného rozsahu



rozhodovacie stromy

- môžu modelovať usporadúvania porovávaním
- pre príslušný algoritmus:
  - jeden strom pre každé *n*
  - cesty v strome sú všetky možné stopy výpočtu
- *aká je asymptotická výška ľubovoľného rozhodovacieho stromu pre usporiadanie n prvkov?*
- $\Omega(n \lg n)$

Lineárne usporadúvanie

- porovnávacie algoritmy maximálne O(n log n)
- algoritmy s lineárnou časovou zložitosťou O(n) používajú iné operácie ako porovnávanie na usporiadanie postupností
- použitie distribuovaných algoritmov – algoritmy, kde údaje zo vstupu sú rozdelené do viacerých prechodných štruktúr, ktoré sa potom zhromaždia a umiestnia na výstupe

Lineárne usporadúvanie

- Výhody a nevýhody oproti porovnávacím algoritmom:
  - lepšia časová zložitosť
  - horšia pamäťová zložitosť – nedá sa pracovať na mieste (in-situ)
  - predpokladá, že vstupné údaje sú z nejakého intervalu

Usporadúvanie spočítavaním (counting sort)

- určuje počet prvkov menších ako prvok *x*, pomocou čoho zistí správnu pozíciu prvku *x* vo vstupnom poli
- predpokladá, že každý prvok z *n* vstupných prvkov je z nejakého intervalu 1..*k*.
- ak má byť efektívny, tak musí platiť, že *k* nie je oveľa väčšie ako *n*
- vhodný na použitie ak *k* je malé a kľúče sa často opakujú
- časová zložitosť O(n+k)
  - ak *k* patrí do O(n), tak beží v čase O(n). Ak *k* je oveľa väčšie ako *n*, napr. ak *k* patrí do O(n^2), tak sa berie O(*k*).

Usporadúvanie spočítavaním

Andy	Bubo	Cica	Dora	Edita	Fero	Gugo	Hugo	Ivo	Joe	Kika

n prvkov poľa, k rôznych kľúčov

spočítaj počet výskytov každého kľúča

Andy	Bubo	Cica	Dora	Edita	Fero	Gugo	Hugo	Ivo	Ján	Kika

jeden prechod cez pole, čas O(n)

	4
	1
	3
	3

transformuj na kumulatívne počty

Andy	Bubo	Cica	Dora	Edita	Fero	Gugo	Hugo	Ivo	Ján	Kika

jeden prechod cez tabuľku počtov, čas O(k)

	4
	5
	8
	11

kopíruj do druhého poľa

Andy	Bubo	Cica	Dora	Edita	Fero	Gugo	Hugo	Ivo	Ján	Kika

				Kika						

	4
	5->4
	8
	11

jeden prechod cez pole, začínajúc vpravo. v každom kroku sa dekrementuje počítadlo v tabuľke počtov. čas O(n).

Andy	Bubo	Cica	Dora	Edita	Fero	Gugo	Hugo	Ivo	Ján	Kika

			Ján	Kika						

	4→3
	4
	8
	11

Andy	Bubo	Cica	Dora	Edita	Fero	Gugo	Hugo	Ivo	Ján	Kika

			Ján	Kika			Ivo			

	3
	4
	8→7
	11

Andy	Bubo	Cica	Dora	Edita	Fero	Gugo	Hugo	Ivo	Ján	Kika

		Hugo	Ján	Kika			Ivo			

	3→2
	4
	7
	11

13

Andy	Bubo	Cica	Dora	Edita	Fero	Gugo	Hugo	Ivo	Ján	Kika

		Herb	Ján	Kika		Gugo	Ivo			

	2
	4
	7→6
	11

14

Andy	Bubo	Cica	Dora	Edita	Fero	Gugo	Hugo	Ivo	Ján	Kika

		Hugo	Ján	Kika		Gugo	Ivo			Fero

	2
	4
	6
	11→10

15

Andy	Bubo	Cica	Dora	Edita	Fero	Gugo	Hugo	Ivo	Ján	Kika

		Hugo	Ján	Kika		Gugo	Ivo		Edita	Fero

	2
	4
	6
	10→9

16

Andy	Bubo	Cica	Dora	Edita	Fero	Gugo	Hugo	Ivo	Ján	Kika

	Dora	Hugo	Ján	Kika		Gugo	Ivo		Edita	Fero

	2→1
	4
	6
	9

17

Andy	Bubo	Cica	Dora	Edita	Fero	Gugo	Hugo	Ivo	Ján	Kika

	Dora	Hugo	Ján	Kika	Cica	Gugo	Ivo		Edita	Fero

	1
	4
	6→5
	9

18

Andy	Bubo	Cica	Dora	Edita	Fero	Gugo	Hugo	Ivo	Ján	Kika

	Dora	Hugo	Ján	Kika	Cica	Gugo	Ivo	Bubo	Edita	Fero

	1
	4
	5
	9→8

algoritmus je stabilný. celkový čas je O(n+k). ak k je konštantné a malé oproti n, O(n).

Andy	Bubo	Cica	Dora	Edita	Fero	Gugo	Hugo	Ivo	Ján	Kika

Andy	Dora	Hugo	Ján	Kika	Cica	Gugo	Ivo	Bubo	Edita	Fero

	1→0
	4
	5
	8

Usporadúvanie spočítavaním

- Pracuje s tromi poliami:
  - Pole *A[]* obsahuje údaje, ktoré sa majú usporiadať – veľkosť poľa je *n*
  - Pole *B[]* obsahuje konečný usporiadaný zoznam údajov – veľkosť poľa je *n*
  - Pole *C[]* je použité na počítanie počtu prvkov – veľkosť poľa je *k*

Usporadúvanie spočítavaním

- Fázy algoritmu:
  - prvý cyklus inicializuje pole *C[]* na nulové hodnoty
  - druhý cyklus inkrementuje hodnoty v *C[]* podľa početnosti ich výskytu v poli *A[]* – početnosť sa zapíše do indexu, ktorý zodpovedá hodnote prvku.
  - tretí cyklus pripočíta ku každej hodnote poľa *C[]* kumulatívny súčet predchádzajúcich hodnôt tohto poľa
  - štvrtý cyklus vypíše usporiadané hodnoty do poľa *B[]* – hodnota na indexe i poľa *C[]* určuje index v poli *B[]*, do ktorého sa zapíše prvok, ktorý sa rovná indexu *i*.

Usporadúvanie spočítavaním

vstup: *A* [1 .. *n*],
A[*J*] ∈ {1,2, . . . , *k* }

výstup: *B* [1 .. *n*],
usporiadaný

používa *C* [1 .. *k*],
pomocná pamäť

Counting-Sort(*A*, *B*, *k*)
1. **for** *i* ← 1 **to** *k*
2. **do** *C*[*i*] ← 0
3. **for** *j* ← 1 **to** *length*[*A*]
4. **do** *C*[*A*[*j*]] ← *C*[*A*[*j*]] + 1
5. **for** *i* ← 2 **to** *k*
6. **do** *C*[*i*] ← *C*[*i*] +*C*[*i* -1]
7. **for** *j* ← *length*[*A*] **down** 1
8. **do** *B* [*C*[*A*[*j*]]] ← *A*[*j*]
9. *C*[*A*[*j*]]] ← *C* [*A*[*j*]] -1

Analysis:

Adapted from Cormen,Leiserson,Rivest

	1	2	3	4	5	6
A	4	1	3	4	3	4

k = 4, dĺžka/*length* = 6

C	0	0	0	0

po riadkoch 1-2

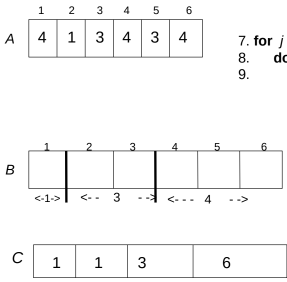
C	1	0	2	3

po riadkoch 3-4

C	1	1	3	6

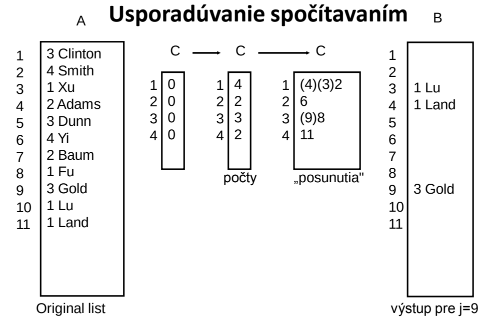
po riadkoch 5-6

Counting-Sort(*A*, *B*, *k*)
1. **for** *i* ← 1 **to** *k*
2. **do** *C*[*i*] ← 0
3. **for** *j* ← 1 **to** *length*[*A*]
4. **do** *C*[*A*[*j*]] ← *C*[*A*[*j*]] + 1
5. **for** *i* ← 2 **to** *k*
6. **do** *C*[*i*] ← *C*[*i*] +*C*[*i* -1]



7. **for** *j* ← *length*[*A*] **down** 1
8. **do** *B* [*C*[*A*[*j*]]] ← *A*[*j*]
9. *C*[*A*[*j*]]] ← *C* [*A*[*j*]] -1

Usporadúvanie spočítavaním



Usporadúvanie spočítavaním

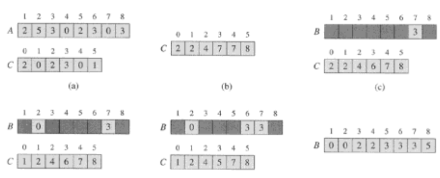
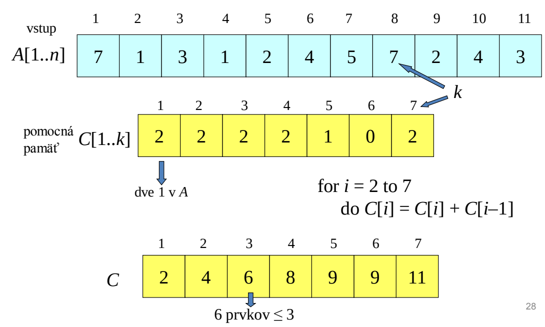
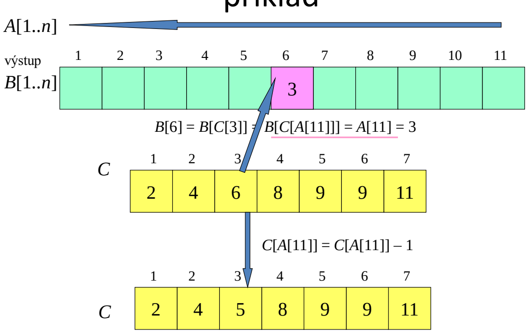


Figure 8.2 The operation of COUNTING-SORT on an input array $A[1..8]$, where each element of A is a nonnegative integer no larger than $k = 5$. **(a)** The array A and the auxiliary array C after line 4. **(b)** The array C after line 7. **(c)–(e)** The output array B and the auxiliary array C after one, two, and three iterations of the loop in lines 9–11, respectively. Only the lightly shaded elements of array B have been filled in. **(f)** The final sorted output array B.

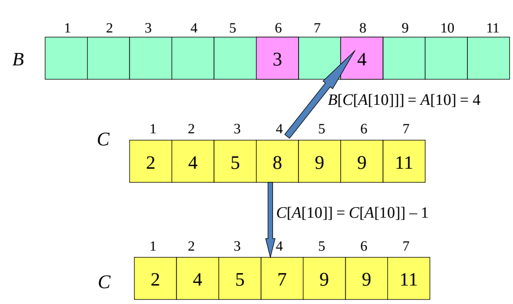
príklad



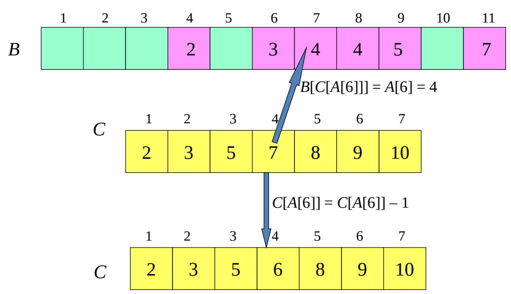
príklad



príklad



príklad



usporadúvanie spočítavaním

```
CountingSort(A, B, k)
      for i=1 to k
            C[i]= 0;
      for j=1 to n
            C[A[j]] += 1;
      for i=2 to k
            C[i] = C[i] + C[i-1];
      for j=n downto 1
            B[C[A[j]]] = A[j];
            C[A[j]] -= 1;
```


usporadúvanie spočítavaním

```
CountingSort(A, B, k)
      for i=1 to k                     čas O(k)
            C[i]= 0;
      for j=1 to n
            C[A[j]] += 1;
      for i=2 to k
            C[i] = C[i] + C[i-1];           čas O(n)
      for j=n downto 1
            B[C[A[j]]] = A[j];
            C[A[j]] -= 1;
```


usporadúvanie spočítavaním

- celkový čas: O(n + k)
  - zvyčajne, k = O(n)
  - teda celkovo O(n)
- čiže významne lepšie než $\Omega(n \lg n)$!
  - lebo toto nie je porovnávací algoritmus
  - je stabilný

usporadúvanie spočítavaním

- prečo vlastne nepoužívame vždy usporadúvanie spočítavaním?
- lebo závisí od rozsahu prvkov k
- *mohli by sme použiť usporadúvanie spočítavaním na usporiadanie celých čísiel zapísateľných do 32 bitov? Prečo áno/nie?*
- nie, k je príliš veľké (2^{32} = 4,294,967,296)

Herman Hollerith

- (1860-1929)
- 1880 spracovanie sčítania ľudu USA trvalo skoro 10 rokov (robí sa každých 10 rokov).
- ako prednášateľ na MIT, Hollerith navrhol prototyp strojov na spracovanie diernych štítkov.
- pomocou jeho strojov sa doba spracovania ďalšieho sčítania ľudu v 1890 skrátila na 6 týždňov.
- založil firmu Tabulating Machine Company v 1911, ktorá sa spojila sa ďalšími firmami v 1924 - vznikla International Business Machines.

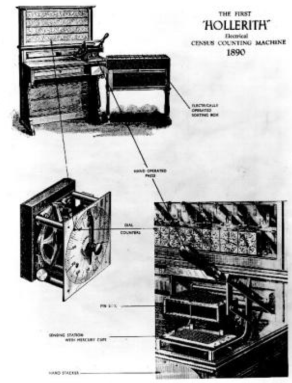
dierny štítok

- dierny štítok = obsahuje záznam údajov.
- dierka = hodnota.

kópia dierneho štítku použitého na spracovanie sčítania ľudu 1900 U.S.A. [Howells 2000]

Hollerithov tabulačný systém

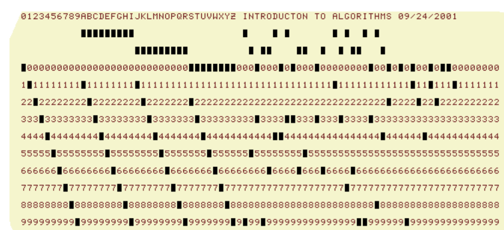
obrázok z [Howells 2000].



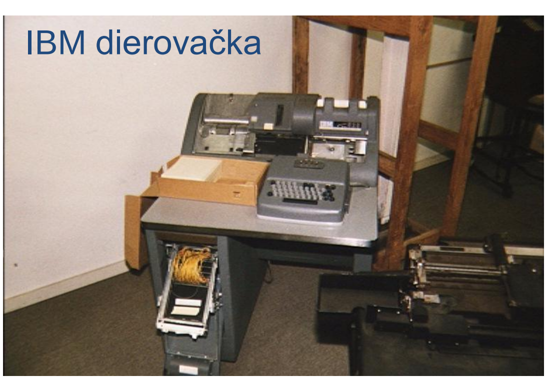
Ako pôvodne zbohatla IBM?

- začiatkom 20. storočia vyrobila čítačky diernych štítkov pre spracovanie údajov zo sčítania ľudu.
- dierny štítok má 80 stĺpcov, stĺpec má 12 miest pre dierky. pomocou dierovačiek diernych štítkov sa na štítky zapisovali údaje.
- triedičky mali 12 priečinkov.
- základná myšlienka: začni triediť podľa najnižšieho rádu.
- voči dovtedajšiemu spôsobu dokázala proces rádovo zrýchliť (1880=7 rokov)

dierny štítok



IBM dierovačka



IBM triedička

pôvod radixového usporadúvania

- Hollerithov pôvodný patent z r. 1889 naznačuje radixové usporadúvanie od najvyššieho rádu:
  - *„Najzložitejšie kombinácie [záznamy, pozostávajúce z položiek údajov] sa dajú ľahko spočítať [usporiadať] s relatívne malým počtom počítadiel tak, že sa najprv štítky zoradia podľa prvých položiek, vstupujúcich do kombinácií, potom sa preusporiadajú podľa druhej položky, vstupujúcej do kombinácie [záznamu] a tak ďalej a nakoniec spočítaním na tých málo počítadlách poslednú položku kombinácie pre každú skupinu [súbor] štítkov. „*
- Úprava algoritmu na usporadúvanie od najnižšieho rádu sa zdá byť výsledok ľudovej tvorivosti operátorov sčítačiek.
- poznámka: a potom, že algoritmus nie je patentovateľný! ak sa patentuje zariadenie, môže opis patentu obsahovať aj algoritmus.

radixové usporadúvanie

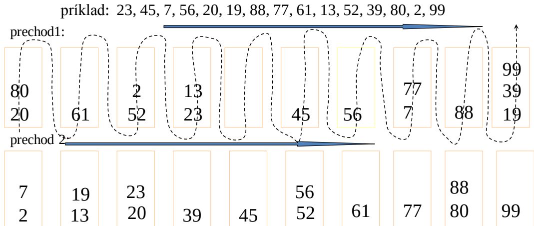
RADIX-SORT(A, d)
 for i ← 1 **to** d
 do stabilné usporadúvanie(A) podľa číslice i

44

radixové usporadúvanie - príklad

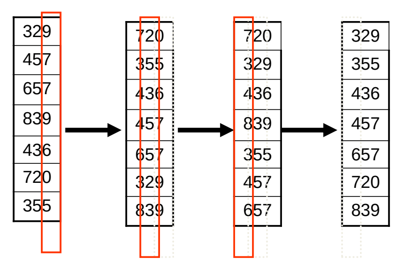
usporiada množinu čísiel vo viacerých prechodoch, začínajúc od číslic najnižšieho (jednotkového) rádu, potom usporiada podľa číslic najbližšieho vyššieho (desiatkového) rádu atď.

príklad: 23, 45, 7, 56, 20, 19, 88, 77, 61, 13, 52, 39, 80, 2, 99



45

radixové usporadúvanie – ďalší príklad



46

radixové usporadúvanie

- číslo zapísané v pozičnej sústave so základom k
  - hodnota = $x_{d-1}k^{d-1} + x_{d-2}k^{d-2} + \ldots + x_2k^2 + x_1k^1 + x_0k^0$
- Radixové usporadúvanie neporovnáva dva kľúče, ale spracúva a porovnáva časti kľúčov
- Kľúče považuje za čísla zapísané v číselnej sústave so základom *k* (radix, koreň), pracuje s jednotlivými číslicami
- Dokáže usporadúvať čísla, znakové reťazce, dáta, ... (počítače reprezentujú všetky údaje ako postupnosti 1 a 0 – binárna sústava => 2 je základ)
  - problém: usporiadať 1 millión 64-bitových čísiel
  - 64 prechodov cez milión čísiel?
    - čo tak interpretovať ich ako čísla v sústave so základom (radixom) 2^{16}, budú to najviac 4-miestne čísla
    - vtedy radixové usporadúvanie usporiada len v 4 prechodoch!

47

radixové usporadúvanie

- LSD Radix sort (least significant digit) – usporadúvanie podľa číslic postupuje od poslednej číslice (s najmenšou váhou) k prvej číslici (s najväčšou váhou)– stabilný.
- MSD Radix sort – od prvej číslice k poslednej – lexikografické usporiadanie – nestabilný
- Je dôležité na samotné usporadúvanie podľa jednotlivých číslic použiť nejaký stabilný algoritmus, aby sa nemenilo poradie prvkov s rovnakými číslicami jednej váhy pri usporadúvaní podľa inej váhy.
- Keďže počet možných číslic (ak *k*=10) je len 10, tak na usporiadanie podľa nich je výhodné použiť usporadúvanie spočítavaním.

48

radixové usporadúvanie

- *ako preukázať, že naozaj usporadúva?*
  - predpokladajme, že postupnosť je podľa číslic nižších rádov {j: j<i} usporiadaná
  - treba ukázať, že usporiadanie podľa nasledujúcej číslice rádu i zanechá postupnosť usporiadanú (podľa nižších rádov ale už vrátane i)
    - ak sú dve číslice na i-tom ráde (mieste odspodu) rôzne, usporiadanie dvoch čísiel podľa tohto rádu je správne (je vyšší rád než všetky, podľa ktorých sa usporadúvalo doteraz, keďže sa ide od najnižšieho, nižšie rády sú irelevantné)
    - ak sú dve číslice na i-tom ráde (mieste odspodu) rovnaké, čísla sú už usporiadané podľa nižších rádov. ak používame stabilný algoritmus, čísla zostanú v správnom poradí

radixové usporadúvanie

- *aký algoritmus použiť na usporiadanie podľa číslic?*
- ponúka sa usporadúvanie spočítavaním:
  - usporiada n čísiel podľa číslic v sústave so základom k, tj rozsah číslic je 0..k-1
  - čas: O(n + k)
- každý prechod cez n d-miestnych čísiel (s d *číslicami)* si vyžiada čas O(n+k), takže celkový čas je O(dn+dk)
  - ak d je konštantné a k=O(n), vyžiada si čas O(n)

radixové usporadúvanie

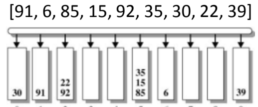
- r.u. založené na usporadúvaním spočítavaním je
  - rýchle
  - asymptoticky rýchle (t.j. O(n))
  - ľahko sa programuje

Iný príklad

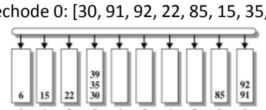
now	sob	tag	ace
for	nob	ace	bet
tip	ace	bet	dim
ilk	tag	dim	for
dim	ilk	tip	hut
tag	dim	sky	ilk
jot	tip	ilk	jot
sob	for	sob	nob
nob	jot	nob	now
sky	hut	for	sky
hut	bet	jot	sob
ace	now	now	tag
bet	sky	hut	tip

príklad

- vstup: [91, 6, 85, 15, 92, 35, 30, 22, 39]



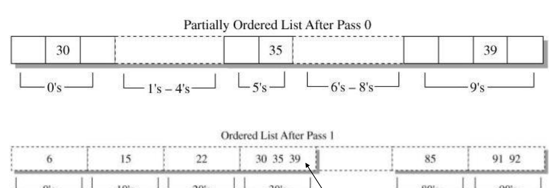
po prechode 0: [30, 91, 92, 22, 85, 15, 35, 6, 39]



po prechode 1: [6, 15, 22, 30, 35, 39, 85, 91, 92]

príklad pokr.

pôvodná postupnosť: {91, 6, 85, 15, 92, 35, 30, 22, 39}



čísla s rovnakým počtom desiatok. v poradí podľa jednotiek

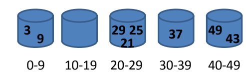
Vedierkové usporadúvanie (Bucket sort)

- Predpokladá, že vstup je akoby generovaný náhodným procesom, ktorý prvky distribuuje rovnomerne na celom intervale.
- Rozdelí interval na *n* rovnako veľkých disjunktných podintervalov (vedierok - bucketov) a potom do nich rozmiestni vstupné čísla
- osobitne v každom vedierku sa potom tieto čísla usporiadajú.

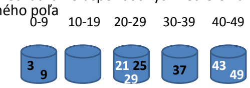
Vedierkové usporadúvanie

- Vytvoria sa prázdne vedierka veľkosti M/n (M – maximálna hodnota vstupného poľa, n – počet prvkov vstupného poľa)
- Rozptýlenie – prechádzanie vstupným poľom a rozmniestnenie každého prvku do prislúchajúceho vedierka (pozor: v príklade na obrázku je počet vedierok stanovený inak)

29 25 3 49 9 37 21 43



- Usporiadanie naplnených vedierok
- Zreťazenie vedierok – postupné prechádzanie usporiadaných vedierok a presúvanie prvkov späť do vstupného poľa

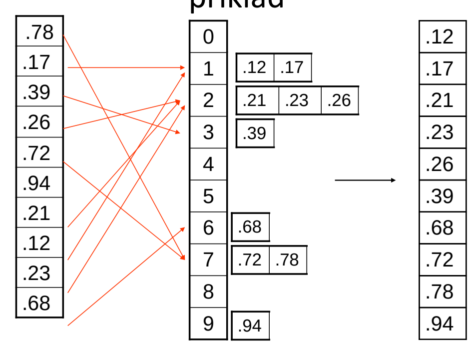


3 9 21 25 29 37 43 49

Vedierkové usporadúvanie

```
BUCKET-SORT(A)
    n ← length(A)
    for i ← 0 to n
        do vlož A[i] do zoznamu B[floor(n*A[i])]
    for i ← 0 to n –1
        do Insertion-Sort(B[i])
zreťaz zoznamy B[0], B[1], … B[n –1] v tomto poradí
```


príklad



vedierko i obsahuje hodnoty z polouzavretého intervalu [i/10, (i + 1)/10).

Vedierkové usporadúvanie - zložitosť

- jednotlivé vedierka väčšinou predstavujú spájaný zoznam, do ktorého sa na správne miesto presúvajú prvky zo vstupného poľa (insert sort)
- činnosti ako vytvorenie vedierok, určenie prislúchajúceho vedierka, presunutie prvku do vedierka a zreťazenie vedierok do výslednej postupnosti trvajú **O(n)**
- časové usporiadanie prvkov vo vedierkach insert sortom trvá **O(n^2)**

Vedierkové usporadúvanie - zložitosť

- výsledná časová zložitosť závisí od rozloženia prvkov vo vedierkach. Ak sú prvky rozmiestnené nerovnomerne a v niektorých vedierkach ich je veľmi veľa, tak časová zložitosť insert sortu O(n^2) prevažuje nad lineárnou zložitosťou a predstavuje výslednú zložitosť celého usporadúvania
- takýto stav sa môže vyskytnúť ak rozsah prvkov *m* je oveľa väčší ako ich počet
- preto sa niekedy celková zložitosť značí podobne ako pri counting sorte O(n+m). Ak m=O(n), tak výsledná časová zložitosť je O(n)
- ak sa počet vedierok rovná počtu vstupných prvkov, tak v priemere to vychádza na jeden prvok v každom vedierku, a preto sa za priemernú zložitosť berie O(n)

Porovnanie jednotlivých metód

Podľa časovej zložitosti

	priem.	*najhoršia*
Vkladaním	0(N^2)	0(N^2)
Výmenou	0(N^2)	0(N^2)
Výberom	0(N^2)	0(N^2)
Shellovo	0(N $\log^2$N)	0(N^2)
QuickSort	0(N log N)	0(N^2)
MergeSort	0(N log N)	0(N log N)
HeapSort	0(N log N)	0(N log N)
CountingSort	0(N + M)	0(N + M)
RadixSort	0(k.N)	0(k.N)
BucketSort	0(N)	0(N^2)

Podľa časovej zložitosti a stability

	pam. zložitosť	*stabilný*
Vkladaním	0(1)	áno
Výmenou	0(1)	áno
Výberom	0(1)	áno
Shellovo	0(1)	nie
QuickSort	0(log N)	nie
MergeSort	0(N)	áno
HeapSort	0(1)	nie
CountingSort	0(N + M)	áno
RadixSort	0(N)	áno(LSD)
BucketSort	0(N)	áno

Porovnanie jednotlivých metód

- aj napriek tomu, že distribuované algoritmy usporadúvania majú v priemere lineárnu zložitosť, tak kvôli vyššej réžii niektorých krokov (extrahovanie cifier, kopírovanie polí) sú v praxi väčšinou pomalšie ako porovnávacie algoritmy usporadúvania s priemernou časovou zložitosťou O(n log n)