Pokročilejší a speciální algoritmy řazení

prof. Ing. Pavel Tvrdík CSc.

Katedra počítačových systémů FIT
České vysoké učení technické v Praze

DSA, ZS 2009/10, Předn. 6

http://service.felk.cvut.cz/courses/X36DSA/

Rekurzivní algoritmus a metoda Rozděl-a-Panuj

- **Rekurzivní** algoritmus: volá sebe sama
  1. jednou na podmnožinu vstupních dat (Příklad HEAPIFY), nebo
  2. vícekrát na několik podmnožin vstupních dat. (Příklad QUICKSORT).
- Druhý případ se nazývá metoda **Rozděl-a-Panuj**.
  1. **Divide:** Rozděl (po příslušných přípravných výpočtech) řešení problému na dvě nebo více řešení stejných podproblémů na podmnožinách vstupních dat.
  2. **Conquer:** Řeš podproblémy rekurzivně. Pokud jsou podproblémy dostatečně malé, vyřeš je přímou metodou.
  3. **Combine:** Zkombinuj výsledky řešení podproblémů do celkového řešení.

Algoritmus MERGESORT

procedure $\textsc{MergeSort}(A, Aux, low, high)$
$\{$
(1) **if** $(low < high)$
(2) **then** $\{$ $half \leftarrow \lfloor (low + high)/2 \rfloor$; // Divide
(3) $\textsc{MergeSort}(A, Aux, low, half)$; // Conquer
(4) $\textsc{MergeSort}(A, Aux, half + 1, high)$; // Conquer
(5) // $A[low, .., half]$ a $A[half + 1, .., high]$ jsou nyní seřazeny
(6) $\textsc{Merge}(A, Aux, low, high)$; // Combine zpět do A
$\}$
$\}$

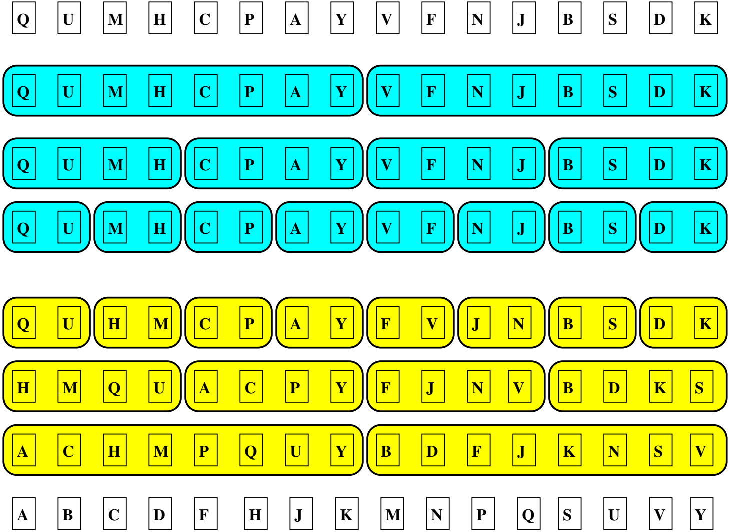
Kontrolní otázka: Kde se zastaví řetěz rekurzivních volání?

Algoritmus Merge

procedure $\textsc{Merge}(A, Aux, low, high)$

```
{
    half ← ⌊(low + high)/2⌋;
    i1 ← low; i2 ← half + 1; j ← low;
    while ((i1 ≤ half) & (i2 ≤ high))
       do { if (A[i1] ≤ A[i2])
                 then { Aux[j] ← A[i1]; i1 + +}
                 else { Aux[j] ← A[i2]; i2 + +};
              j + +;
          }
    while (i1 ≤ half) // připoj zbytek z levé poloviny
       do { Aux[j] ← A[i1]; i1 + +; j + +};
     while (i2 ≤ high) // připoj zbytek z pravé poloviny
       do { Aux[j] ← A[i2]; i2 + +; j + +};
     A ← Aux; // přepni opět na hlavní pole A
}
```

Příklad běhu MERGESORT



Věta

Časová složitost algoritmu MergeSort *pro* n *čísel je*

$$t_{\mathrm{MS}}(n) = \Theta(n \log n) \tag{1}$$

a nezávisí na vstupních hodnotách. Paměťová složitost je $2n + \Theta(1)$.

Důkaz. Z rekurzivní definice algoritmu plyne, že

$$t_{\mathrm{MS}}(n) = \Theta(1) + 2t_{\mathrm{MS}}(n/2) + t_{\mathrm{M}}(n),$$

kde $t_{\mathrm{M}}(n) = \Theta(n)$ je časová složitost sloučení dvou posloupností o délce $n/2$ do jedné posloupnosti délky n pomocí Merge. Asymptoticky vždy lineární nezávisle na hodnotách vstupních čísel, počet operací srovnání je mezi $n/2$ a $n-1$.

Tato rekurentní rovnice má řešení $t_{\mathrm{MS}}(n) = \Theta(n \log n)$.
(podrobněji viz přednáška č. 8). ∎

QuickSort Revisited

- QuickSort je další typický představitel metody Rozděl-a-Panuj.
- Existují varianty dvou typů:
  1. jednoduché a stabilní, které ale mají větší skryté konstanty a vyžadují $O(n)$ pomocné paměti,
  2. komplikovanější, nestabilní, ale fakticky rychlejší a nevyžadující extra paměť, tzv. **in place** algoritmy.
- Algoritmy prvního typu:
  - Vyberou pivota.
  - Projdou pole, označí a oindexují čísla menší, rovna a větší než pivot.
  - Zahustí čísla menší než pivot doleva, za ně nahustí čísla rovna pivotu a za ně čísla větší než pivot.
  - Nad první a třetí částí spustí rekurzivně totéž.
- Algoritmus v Přednášce 4 je druhého typu a budeme se jím zabývat dále.
- V obou případech záleží hodně na výběru pivota: Obecně funkce $\textsc{SelectPivot}(A, low, high)$.

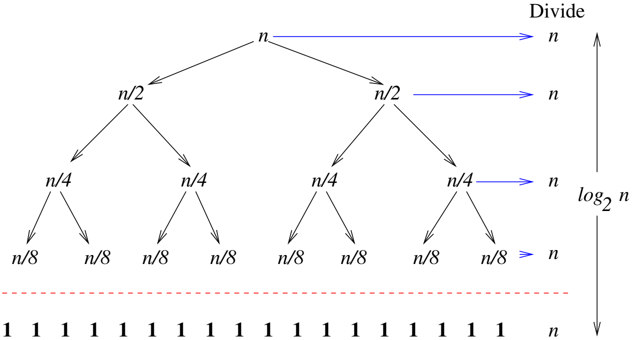
In-place algoritmus QuickSort

procedure $\textsc{QuickSort}(A, low, high)$
{
$il \leftarrow low$; $ir \leftarrow high$;
$pivot \leftarrow \textsc{SelectPivot}(A, low, high)$; // typicky $pivot \leftarrow A[low]$
do { // Rozděl
  while $(A[il] < pivot)$ **do** $il++$;
  while $(A[ir] > pivot)$ **do** $ir--$;
  if $(il < ir)$
    then { $A[il] \leftrightarrow A[ir]$; $il++; ir--$}
    else if $(il = ir)$
      then { $il++; ir--$};
      } **while** $(il \leq ir)$;
if $(low < ir)$ // a Panuj
  then $\textsc{QuickSort}(A, low, ir)$;
if $(il < high)$ // a Panuj
  then $\textsc{QuickSort}(A, il, high)$; }

Složitost algoritmu QuickSortu

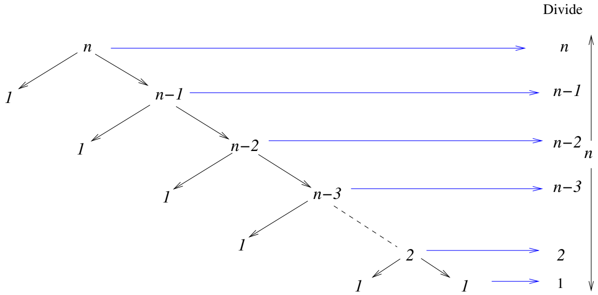
- Hlavní **while** cyklus (3)-(10) realizuje dělení pole na
  - část obsahující čísla menší nebo rovna pivotu $A[low, .., ir]$,
  - část obsahující čísla větší nebo rovna pivotu $A[il, .., high]$.
- Kontrolní otázka: Proč se pivot $A[low]$ vždy přehodí do pravé části?
- Časová složitost QuickSortu závisí na vstupních datech a způsobu výběru pivota, což má vliv na výsledný **dělící poměr**.
- **Dělící poměr** je po skončení hlavního **while** cyklu (3)-(10) roven $(ir - low) : (high - il)$.
- Rozlišují se 4 případy. Uvažujme pole o n prvcích.
  - **Nejlepší (vyvážený) dělící poměr**: dělení na stejné poloviny $n/2{:}n/2$.
  - **Nejhorší dělící poměr**: $1 : n - 1$.
  - **Pevný nerovný dělící poměr**: typicky $\alpha n : (1 - \alpha)n$ pro nějaké $0 < \alpha < 1/2$.
  - **Průměrný dělící poměr**: ?????

Složitost v nejlepším případě



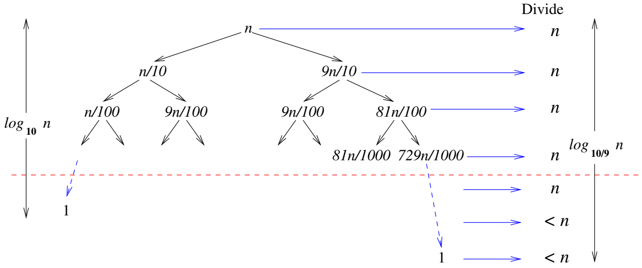
- Předpoklad: Dělící poměr je 1:1 na **všech úrovních** dělícího stromu.
- **Dělící strom** je UBS s hloubkou $\log n$.
- Celková složitost všech dělení (kód (3)-(10)) je v hloubce i dělícího stromu vždy $\Theta(n)$.
- $t_{\mathrm{QS}}(n) = 2 * t_{\mathrm{QS}}(n/2) + \Theta(n)$. Čili $t_{\mathrm{QS}}(n) = \Theta(n \log n)$.

Složitost v nejhorším případě



- Předpoklady:
  - Dělící poměr je 1:$n-1$ v **první úrovni** dělícího stromu.
  - Na dalších hladinách se zbývající úsek délky m, kde postupně $m = n-1, n-2, \ldots$, dělí v poměru 1:$m-1$ v čase $\Theta(m)$.
- Dělící strom je tudíž lineární o výšce n a složitost dělení lineárně klesá.
- $t_{\mathrm{QS}}(n) = t_{\mathrm{QS}}(n-1) + \Theta(n)$. Čili $t_{\mathrm{QS}}(n) = \Theta(n^2)$.
- Může snadno nastat: $pivot = A[low]$ & A je seřazena sestupně!!!!! (Jak takovou posloupnost zvládne MergeSort a InsertSort??)

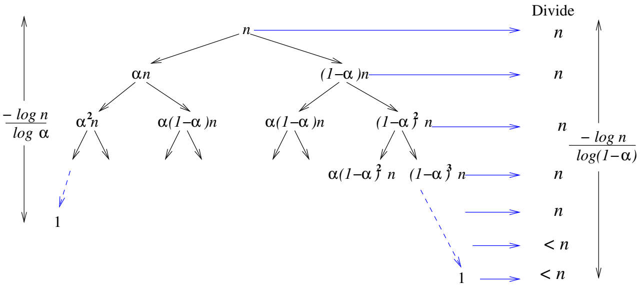
Složitost v případě pevného poměru 1:9



- Předpoklad: Dělící poměr je 1:9 na **všech úrovních** dělícího stromu (což lze obecně považovat za velmi **nevyvážený** poměr).
- Vznikne binární strom, ve kterém listy (úseky délky 1) se objeví
  - počínaje hloubkou $\lceil \log_{10} n \rceil$. Až potud se dělí celé pole o velikosti n.
  - konče hloubkou $\left\lceil \log_{\frac{10}{9}} n \right\rceil$. Velikost dělených úseků klesá a je $\leq n$.

    (Srovnej např. $\log_{10} 10^{12} = 12$ a $\log_{\frac{10}{9}} 10^{12} \doteq 262$).
- $t_{\mathrm{QS}}(n) = t_{\mathrm{QS}}(9n/10) + t_{\mathrm{QS}}(n/9) + O(n)$. Čili $t_{\mathrm{QS}}(n) = O(n \log_{\frac{10}{9}} n)$.

Složitost v případě obecného pevného poměru $\alpha : (1-\alpha)$



- Předpoklad: Dělící poměr je $\alpha : (1-\alpha)$ na **všech úrovních** dělícího stromu, $0 < \alpha < 0.5$.
- Naprosto stejná úvaha i závěry:
  - Binární dělící strom s listy v hloubce od $\log_{\frac{1}{\alpha}} n$ do $\log_{\frac{1}{1-\alpha}} n$.
  - $t_{\mathrm{QS}}(n) = t_{\mathrm{QS}}((1-\alpha)n) + t_{\mathrm{QS}}(\alpha n) + O(n)$. Čili $t_{\mathrm{QS}}(n) = O(n \log_{\frac{1}{1-\alpha}} n) = O(-n \log n / \log(1-\alpha))$.
- Závěr: Složitost řazení mnoha "špatných dat" je spíše blízko k nejlepšímu případu než k nejhoršímu.

Složitost v průměrném případě

- Předpoklad rovnoměrného rozložení: každá z $n!$ možných vstupních posloupností se objevuje s pravděpodobností $\frac{1}{n!}$.
- Je velmi nepravděpodobné, že dělící poměr na všech úrovních bude stejný.
- Pravděpodobně některá dělení budou vyvážená a některá nevyvážená.
- Model průměrného případu:
  - např. **střídání** nejlepšího $(1{:}1)$ a nejhoršího dělícího průměru $(1{:}n-1)$.
  - vyjde jenom o něco hůře než nejlepší.
- Otočíme pohled: místo modelování a spekulování o dělících poměrech **vnutíme** vstupním datům charakter náhodné posloupnosti.
- Typicky: vstupní posloupnost podrobíme **randomizaci** (rozházení), čímž složitost QS není závislá na uspořádanosti vstupu.
- Potřebujeme **náhodný generátor**.

Příklad randomizovaného QuickSortu

Původní QuickSort v Přednášce 4 používal:

function $\textsc{SelectFirstPivot}(A, low, high)$
{
  return $A[low]$ }

Mějme funkci $\textsc{Random}(a, b)$, která vrátí celé číslo c, $a \le c \le b$, s pravděpodobností $\frac{1}{b-a+1}$.

Pak randomizovaný QuickSort používá:

function $\textsc{SelectRandomPivot}(A, low, high)$
{
  $i \leftarrow \textsc{Random}(low, high)$;
  $A[i] \leftrightarrow A[low]$;
  return $A[low]$ }

Optimalita a spodní meze na složitost řazení

- Všechny dosud uvedené algoritmy řazení jsou založeny na **binární** operaci **Porovnej-a-Vyměň** (CE):
  - výsledné pořadí je určeno pouze porovnáváním dvou hodnot,
  - jednotlivé algoritmy se liší pouze strategií výběru porovnávaných dvojic,
  - složitost je měřena celkovým počtem operací CE.
- MergeSort dosahuje **vždy** složitosti $\Theta(n \log n)$.
- HeapSort dosahuje v **nejhorším případě** složitosti $O(n \log n)$.
- QuickSort dosahuje v **průměrném případě** složitosti $O(n \log n)$.
- Pro každý z těchto algoritmů existují data, která si vynutí $\Omega(n \log n)$ CE operací.

Rozhodovací stromy a optimalita

Definice

Algoritmus je **optimální***, pokud má v nejhorším případě složitost rovnou asymptoticky spodní mezi složitosti řešení daného problému.*

Věta

Spodní mez složitosti řešení problému řazení n čísel pomocí CE operace je $\Omega(n \log n)$.

Důkaz. Uvažujme pouze vstupy s n různými čísly $A = [a_1, .., a_n]$.
Libovolný CE algoritmus S lze **abstraktně** popsat pomocí **rozhodovacího stromu** $RS(S, n)$:

- Každý vnitřní uzel $RS(S, n)$ odpovídá $CE(a_i, a_j)$ pro nějaká $1 \leq i \neq j \leq n$.
- Levý podstrom diktuje následné CE operace, pokud $a_i \leq a_j$ nebo to je list.
- Pravý podstrom diktuje následné CE operace, pokud $a_i > a_j$ nebo to je list.

- Každý list odpovídá právě jedné permutaci $\pi(A)$, kde $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$.
- Každému provedení S odpovídá právě jeden průchod v $RS(S,n)$ z kořene do nějakého listu.

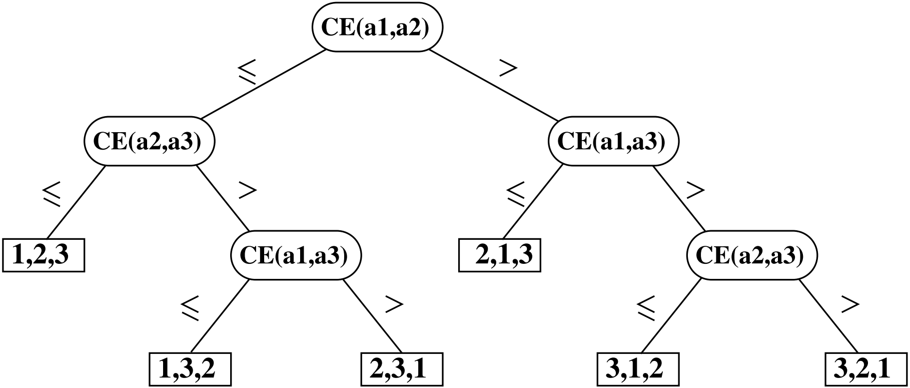
Pro libovolný S je $RS(S,n)$ **plný binární** strom s $n!$ listy
$\Rightarrow$ jeho hloubka je **nejméně** $\log(n!) = \Theta(n \log n)$ (viz 1. cvičení). ∎

Důsledek

Algoritmy MergeSort *v každém,* HeapSort *v nejhorším a* QuickSort *v průměrném případě jsou optimální.*

Kontrolní otázka 1: Jaká je nejmenší možná hloubka listu v $RS(S,n)$?
Kontrolní otázka 2: Může existovat CE řadící algoritmus, kterému pro aspoň jednu polovinu ze všech $n!$ vstupů stačí $O(n)$ CE operací?

Příklad RS pro $n = 3$



Speciální algoritmy řazení v $O(n)$ čase

- Vzhledem k výše popsané nepřekročitelné mezi $\Omega(n \log n)$ lze seřadit n čísel v čase menším, pouze pokud:
  1. Na vstupní hodnoty je uvalena dodatečná omezující podmínka a díky tomu
  2. Řazení není založeno na operaci CE (tudíž složitost algoritmu se počítá pomocí základních operací).
- Uvedeme si 3 takové algoritmy:
  1. CountingSort
  2. RadixSort
  3. BucketSort

CountingSort

Předpoklady: $\exists$ konstanta k; $\forall i, 1 \leq a_i \leq k$

Hlavní myšlenka: $\forall a_i$ spočítáme, kolik v A existuje čísel menších $\leq a_i =$ výsledná pozice a_i po seřazení.

procedure $\textsc{CountingSort}(A, B, Count, k)$
{ // A vstupní pole, B výstupní pole, $Count$ pomocné pole
  for $i \leftarrow 1$ **to** k
    do $Count[i] \leftarrow 0;$
  for $j \leftarrow 1$ **to** $length(A)$
    do $Count[A[j]] \leftarrow Count[A[j]] + 1;$
  // $Count[i] =$ čítač počtu vstupních čísel rovných i
  for $i \leftarrow 2$ **to** k // prefixový součet
    do $Count[i] \leftarrow Count[i] + Count[i-1];$
  // $Count[i] =$ čítač počtu vstupních čísel $\leq i$
  for $j \leftarrow length(A)$ **downto** 1
    do { $B[Count[A[j]]] \leftarrow A[j];$
        $Count[A[j]] \leftarrow Count[A[j]] - 1;$ } }

Příklad běhu CountingSortu

	1	2	3	4	5	6	7	8
A	3	6	4	1	3	4	1	4

	1	2	3	4	5	6
Count	2	0	2	3	0	1

Příklad běhu CountingSortu

	1	2	3	4	5	6
Count	2	2	4	7	7	8

Příklad běhu CountingSortu

	1	2	3	4	5	6	7	8
B							4	

	1	2	3	4	5	6
Count	2	2	4	6	7	8

Příklad běhu CountingSortu

	1	2	3	4	5	6	7	8
B		1					4	

	1	2	3	4	5	6
Count	1	2	4	6	7	8

Příklad běhu CountingSortu

	1	2	3	4	5	6	7	8
B		1				4	4	

	1	2	3	4	5	6
Count	1	2	4	5	7	8

Příklad běhu CountingSortu

	1	2	3	4	5	6	7	8
B	1	1	3	3	4	4	4	6

Časová a paměťová složitost a stabilita algoritmu CountingSort

- **for** cykly na řádcích (1)-(2) a (6)-(7) trvají $O(k)$.
- **for** cykly na řádcích (3)-(4) a (9)-(11) trvají $O(n)$.
- Celkově $t_{\mathrm{CS}}(n) = O(k+n)$.
- Pokud $k = O(n)$ (typická podmínka nasazení CountingSort), pak $t_{\mathrm{CS}}(n) = O(n)$.
- Paměťová složitost je evidentně $2n + k + \Theta(1)$.
- CountingSort je stabilní.
  - Před vstupem do cyklu na řádce (9) je v $Count[i]$ **celkový** počet vstupních čísel $\leq i$.
  - Tento počet byl získán průchodem **zleva doprava**.
  - Protože **for** cyklus na řádce (9) postupuje **odzadu zprava doleva**, je $Count[i]$ současně i výsledná pozice posledního výskytu hodnoty i.
  - Pokud se hodnota i vyskytuje vícekrát, pak na řádku (11) odečtení 1 z $Count[i]$ při každém zapracování jednoho výskytu hodnoty i způsobí, že všechna čísla s touto hodnotou se postupně ukládají vedle doleva.

RadixSort

Předpoklady:

1. Každé vstupní číslo a_i se dá vyjádřit pomocí k číslic v d-ární poziční soustavě.
2. Pozice se číslují LSD$= 1$ až MSD$= k$.

procedure $\textsc{RadixSort}(A, k)$
$\{$// A vstupní pole, případně i výstupní (záleží na implem. (2))
(1) **for** $i \leftarrow 1$ **to** k
(2) **do** seřaď čísla v A podle číslic na pozici i
pomocí libovolného **stabilního řazení** $\textsc{StableSort}$
$\}$

Možná aplikace: řazení postupně podle několika klíčů, např. den, měsíc, rok.

Příklad běhu algoritmu RadixSort

$$n = 9,\ d = 3,\ k = 4$$

Vstup	$i = 1$	$i = 2$	$i = 3$	$i = 4$
2101	012**0**	21**0**1	1**0**01	**0**021
1022	210**1**	10**0**1	0**0**21	**0**120
2211	221**1**	12**0**2	1**0**22	**1**001
0120	002**1**	22**1**1	2**1**01	**1**022
0021	111**1**	11**1**1	1**1**11	**1**111
1202	212**1**	01**2**0	0**1**20	**1**202
1111	100**1**	00**2**1	2**1**21	**2**101
2121	102**2**	21**2**1	1**2**02	**2**121
1001	120**2**	10**2**2	2**2**11	**2**211

Korektnost alg. RadixSort (aneb Jak to, že funguje?)

Věta

Algoritmus RadixSort *řadí vstupní čísla správně a je stabilní.*

Důkaz. Indukcí před číslo pozice. Dokážeme **indukční hypotézu**: Po provedení i iterací **for** cyklu platí, že vstupní pole hodnot ořezaných na nižších i pozic je správně seřazené.

- **Základ indukce:** Platí evidentně pro první iteraci, kdy se nahoru přesunou čísla končící číslicí 0, pak následují čísla končící číslicí 1, atd.
- **Indukční krok:** Předpokládejme, že uvedené tvrzení platí pro j, $1 \le j < k$.
  - Řazení v $(i = j + 1)$-té iteraci **posouvá směrem nahoru** nejprve čísla mající na pozici i číslici 0. Protože tato čísla oříznutá na posledních $i - 1$ pozic byla v předchozích iteracích seřazena správně, bude tato skupina čísel oříznutých na i pozic také seřazena správně.
  - Pak bude podobným způsobem následovat skupina čísel, majících na pozici i číslici 1, po té číslicí 2, atd.
  - Po skončení i-té iterace cyklu budou tedy čísla oříznutá na i pozic seřazena správně.

Složitost algoritmu RadixSort

- Záleží primárně na použitém pomocném řadícím algoritmu $t_{\mathrm{RS}}(n) = k * t_{\mathrm{StableSort}}(n)$
- Pokud je d malé, pak je vhodným kandidátem CountingSort.
  - Každá iterace pak trvá $O(n + d)$.
  - Celkově pak $t_{\mathrm{RS}}(n) = O(k(n + d))$.
  - Paměťová složitost je stejná jako u CountingSort.
  - Pokud k je konstantní a $d = O(n)$, pak $t_{\mathrm{RS}}(n) = O(n)$.

Srovnání a praktická použitelnost:

- Uvažujme $n = 10^9$ 64-bitových čísel, která interpretujeme jako $(k = 4)$-ciferná čísla v $(d = 2^{16} = 65536)$-ární soustavě.
- Pak optimální CE algoritmus bude nad každým číslem provádět přibližně $\log 10^9 \doteq 30$ CE operací. Celkově pak $n \log n = 10^9 * 30$.
- RadixSort zavolá 4krát CountingSort a ten na každé číslo sáhne právě 2krát plus několik průchodů polem o velikosti $d = 2^{16}$.
- Časová složitost je tedy výrazně nižší.
- Paměťová složitost RadixSort je ale $2n + d$, kdežto některé $O(n \log n)$ algoritmy mají složitost n, ale jsou zase nestabilní.

Domácí úkol

- Navrhněte algoritmus stabilního řazení n čísel z intervalu hodnot $[1, n^2]$ s časovou složitostí $O(n)$.

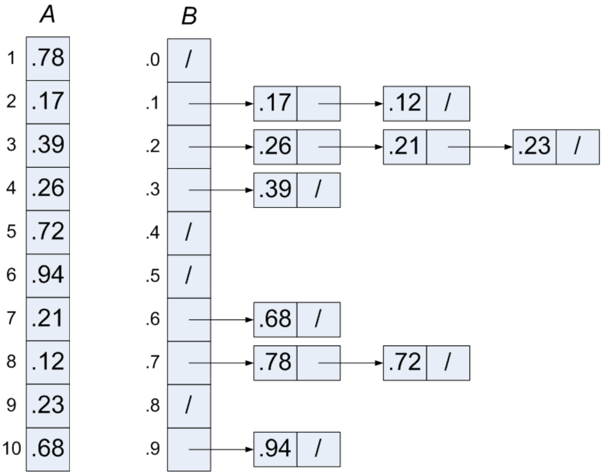
BucketSort

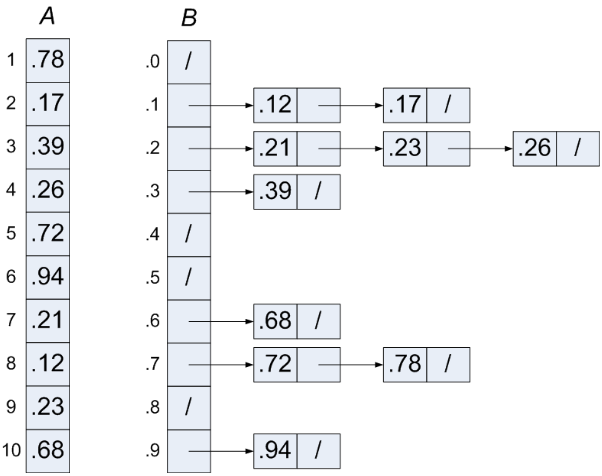
- Vstupní čísla jsou z intervalu $[0, 1)$.
- Algoritmus je vhodný, pokud jsou vstupní čísla **rovnoměrně rozložená** v tomto intervalu.
- Hlavní myšlenky:
  1. Interval $[0, 1)$ rozdělíme na n přihrádek, do kterých vstupní pole distribuujeme.
  2. Díky předpokladu rovnoměrnosti rozdělení předpokládáme, že do jedné přihrádky spadne málo čísel.
  3. Nicméně, protože velikost přihrádky nelze dopředu přesně určit, je implementována jako dynamický zřetězený seznam.
  4. Každou přihrádku rychle seřadíme a výsledek dostaneme zřetězením seřazených přihrádek.

procedure $\textsc{BucketSort}(A, B)$
$\{//\ A$ vstupní pole, B pole přihrádek
(1) $n \leftarrow length(A)$;
(2) **for** $i \leftarrow 1$ **to** n
(3) **do** připoj $A[i]$ k seznamu v přihrádce $B[\lfloor n * A[i] \rfloor]$;
(4) **for** $i \leftarrow 0$ **to** $n - 1$
(5) **do** seřaď seznam $B[i]$ stabilním alg. (např. $\textsc{InsertionSort}$em)
(6) zřetěz seznamy $B[0], B[1], B[2], \dots, B[n-1]$ do A.
$\}$

Poznámka: Počet přihrádek závisí na mapovací funkci na řádku (3), může být $< n$.

Příklad běhu BucketSortu




A
B
1
.78
2
.17
3
.39
4
.26
5
.72
6
.94
7
.21
8
.12
9
.23
10
.68
.0
/
.1
.12
.17
/
.2
.21
.23
.26
/
.3
.39
/
.4
/
.5
/
.6
.68
/
.7
.72
.78
/
.8
/
.9
.94
/

Korektnost a stabilita BUCKETSORT

Věta

Algoritmus BUCKETSORT *řadí vstupní čísla správně a je stabilní.*

Důkaz.

- Uvažujme 2 vstupní hodnoty $A[i]$ a $A[j]$.
- Pokud padnou do stejné přihrádky, pak jsou v ní seřazeny a objeví se na výstupu ve správném pořadí.
- Dvě stejné vstupní hodnoty díky stabilnosti řazení v přihrádkách nezmění pořadí.
- Nechť padnou do různých přihrádek $B[i']$ a $B[j']$. Nechť $i' < j'$.
- Funkce $f(x) = \lfloor n * x \rfloor$ je ale neklesající funkce: $f(x) < f(y)$ $\Rightarrow \quad x < y$. Čili

$$i' = \lfloor n * A[i] \rfloor < j' = \lfloor n * A[j] \rfloor \Rightarrow A[i] < A[j].$$

Překladový slovníček

Rozděl-a-Panuj	Divide-and-Conquer
rozhodovací strom	decision tree
Porovnej-a-Vyměň	Compare-and-Exchange
LSD	least significant digit
MSD	most significant digit