Základy řazení

Karel Richta a kol.

Přednášky byly připraveny s pomocí materiálů, které vyrobili Marko Berezovský, Petr Felkel, Josef Kolář, Michal Píše a Pavel Tvrdík

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

© Karel Richta a kol., 2020

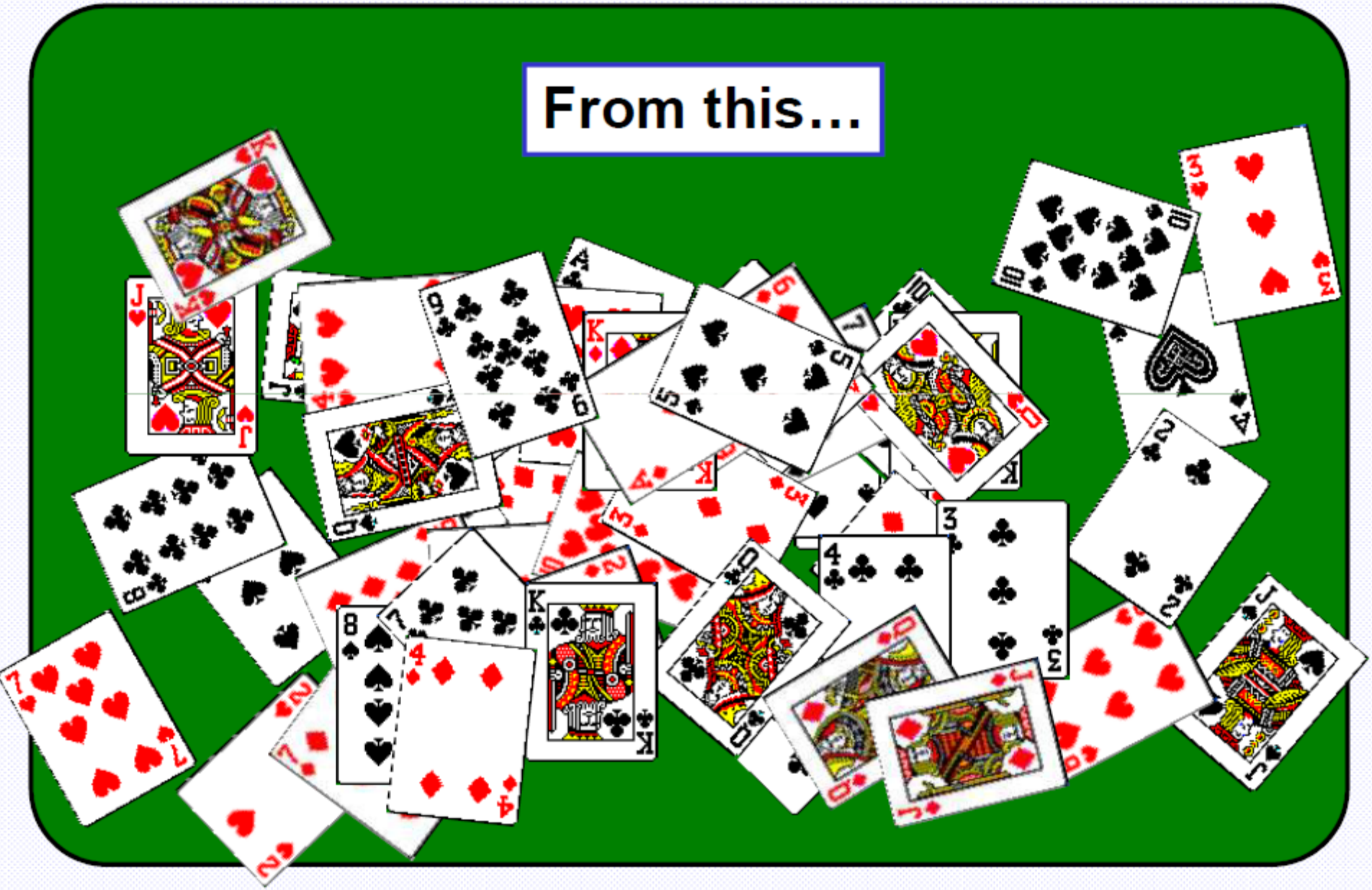
Datové struktury a algoritmy, B6B36DSA
02/2020, Lekce 4

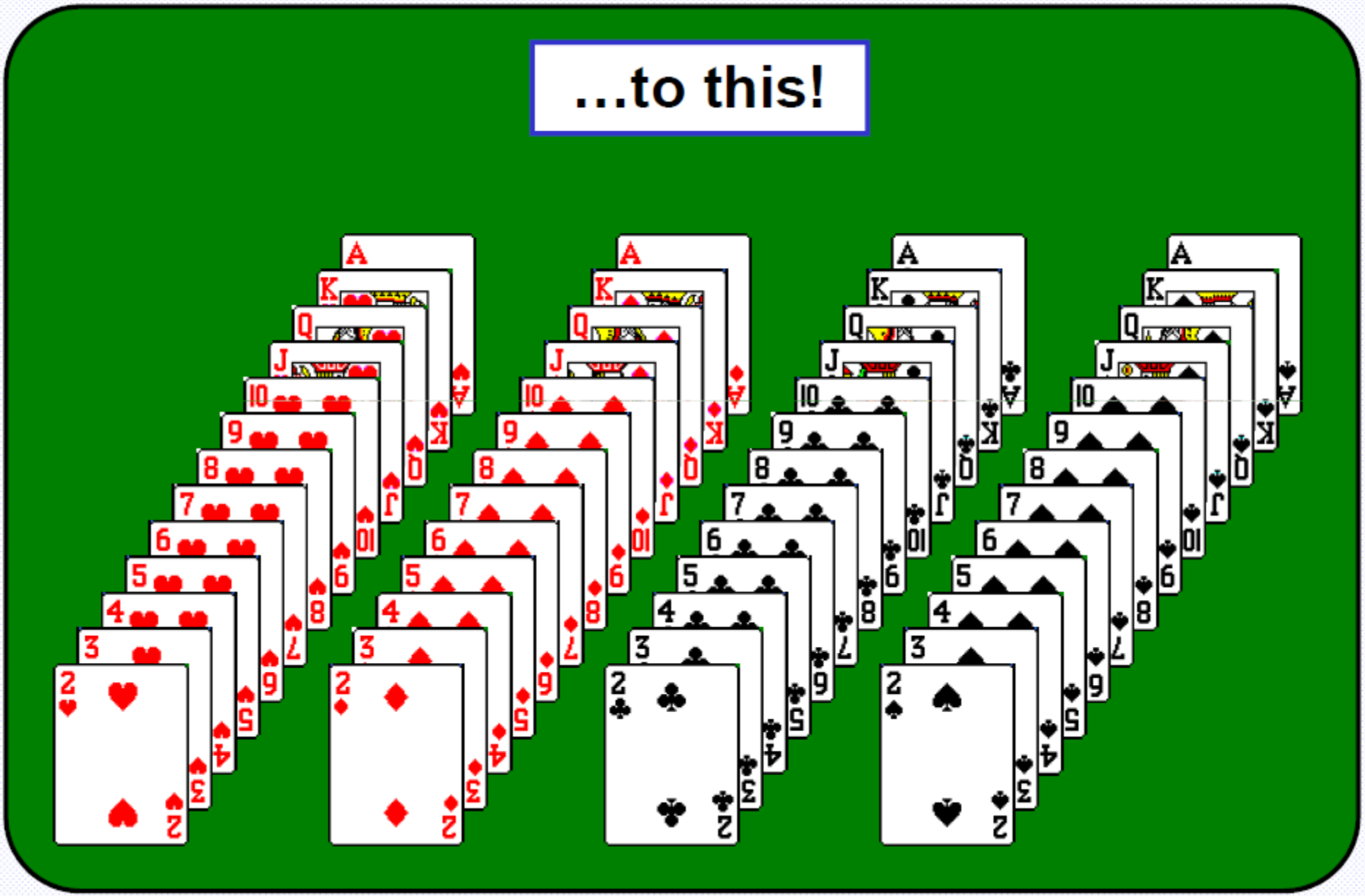
https://cw.fel.cvut.cz/wiki/courses/b6b36dsa/start

Evropský sociální fond
Praha & EU: Investujeme do vaší budoucnosti

Definice problému řazení

- Mějme parciálně uspořádanou množinu (poset) M, kde uspořádání je binární relace $\leq \subseteq M \times M$, která je reflexivní, tranzitivní a antisymetrická.
- Definice problému řazení:
- Na vstupu je posloupnost prvků z M; cílem je najít takovou posloupnost prvků z M, pro kterou platí dvě základní kritéria:
  - výstupní posloupnost je <u>seřazená</u> a
  - výstupní posloupnost je <u>permutací</u> původní posloupnosti (obsahuje tedy stejná data, jen v jiném pořadí).
- Z hlediska řazení se vstupní data často chápou jako soubor dvojic klíč–hodnota, přičemž po seřazení je posloupnost <u>klíčů</u> monotónní, zatímco na připojené hodnoty se při řazení nebere zřetel. Při existenci několika položek se stejným klíčem se však podle pořadí odpovídajících hodnot rozlišují <u>stabilní</u> a <u>nestabilní</u> algoritmy. V definici relace uspořádání $\leq$ se přitom bere ohled pouze na klíče příslušných hodnot.


From this…


...to this!

Základní taxonomie řazení

- Řazení bez využití porovnání (adresní řazení: radix-sort, bucket-sort)
- Řazení využívající porovnání (asociativní řazení):
  - Nevyvážené dělení řazené posloupnosti:
    - Důraz na analýzu (řazení přímým výběrem: selection–sort)
    - Důraz na syntézu (řazení přímým zatřiďováním: insertion-sort)
  - Vyvážené dělení řazené posloupnosti:
    - Důraz na analýzu (řazení dělením: quick-sort)
    - Důraz na syntézu (řazení slučováním: merge-sort)

Cvičení

Navrhněte a popište algoritmus pro seřazení balíčku karet.

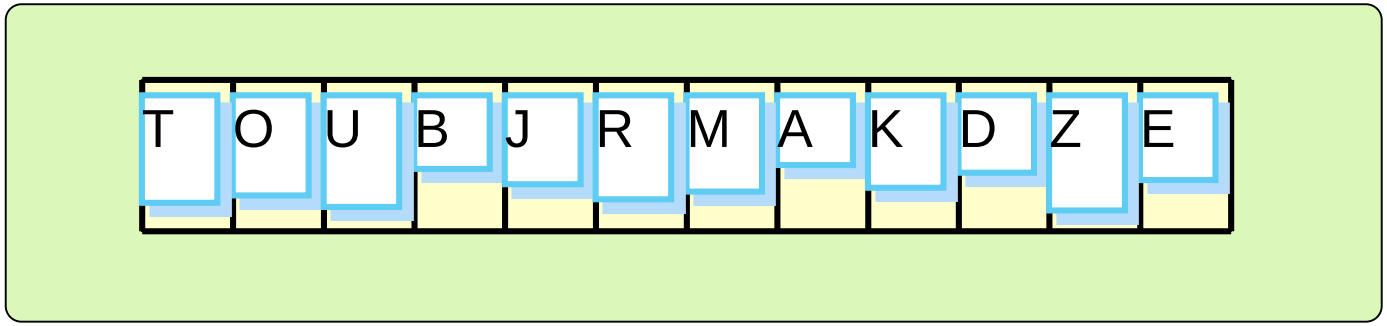
- Smíte používat jen jednu ruku a v ní smíte držet vždy nanejvýš jednu kartu.
- Během řazení můžete (a musíte) odkládat karty do několika pomocných balíčků.
- Z každého balíčku je vidět a je známa pouze hodnota vrchní karty.

Úkolem je sestavit algoritmus tak, aby využíval co nejmenšího počtu pomocných balíčků.

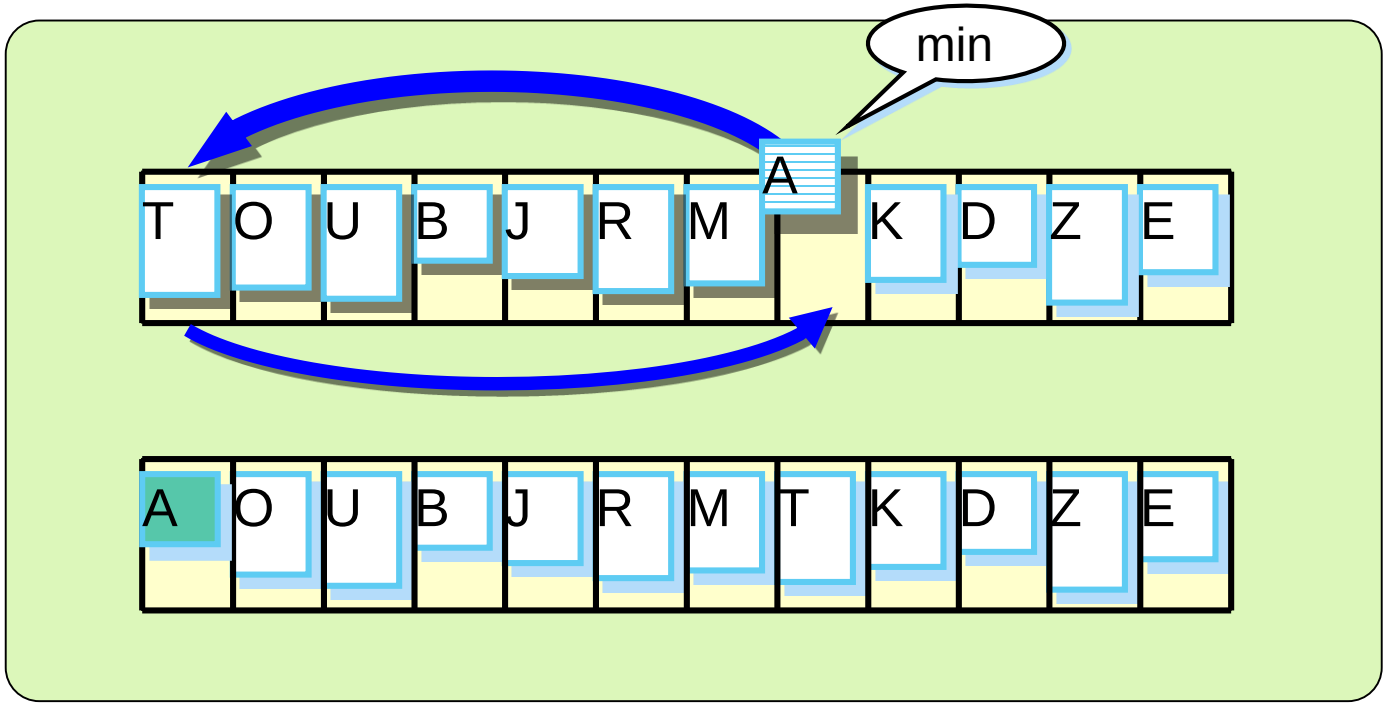
SELECTION-SORT
řazení přímým výběrem

SELECTION-SORT

Start

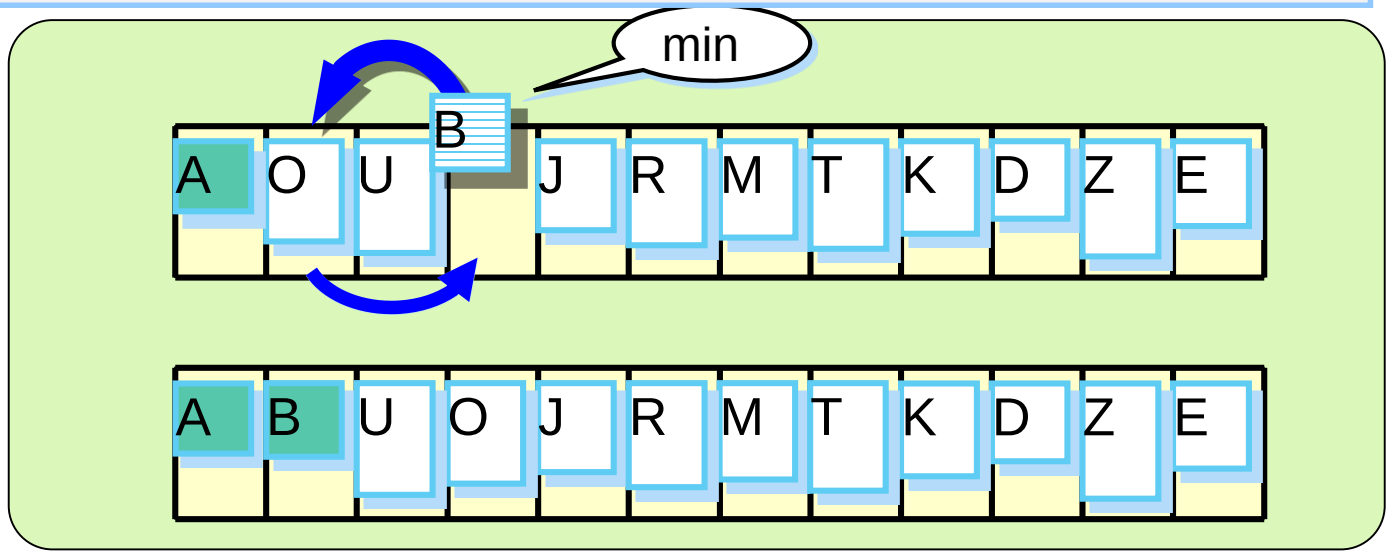


Krok 1

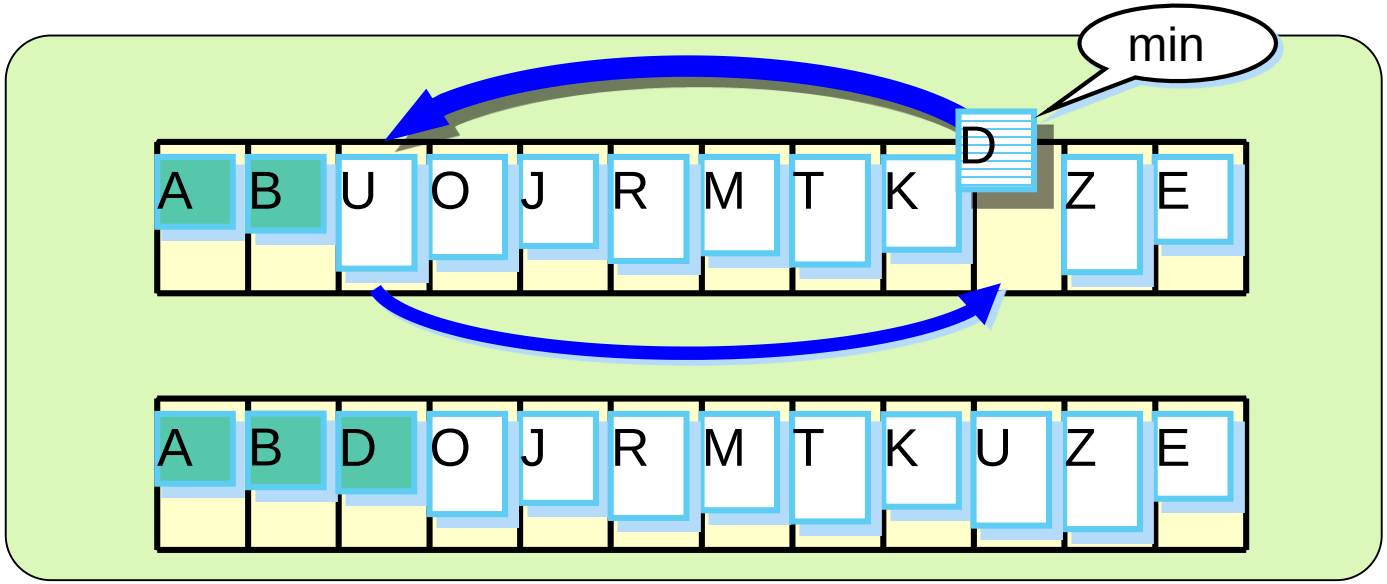


SELECTION-SORT

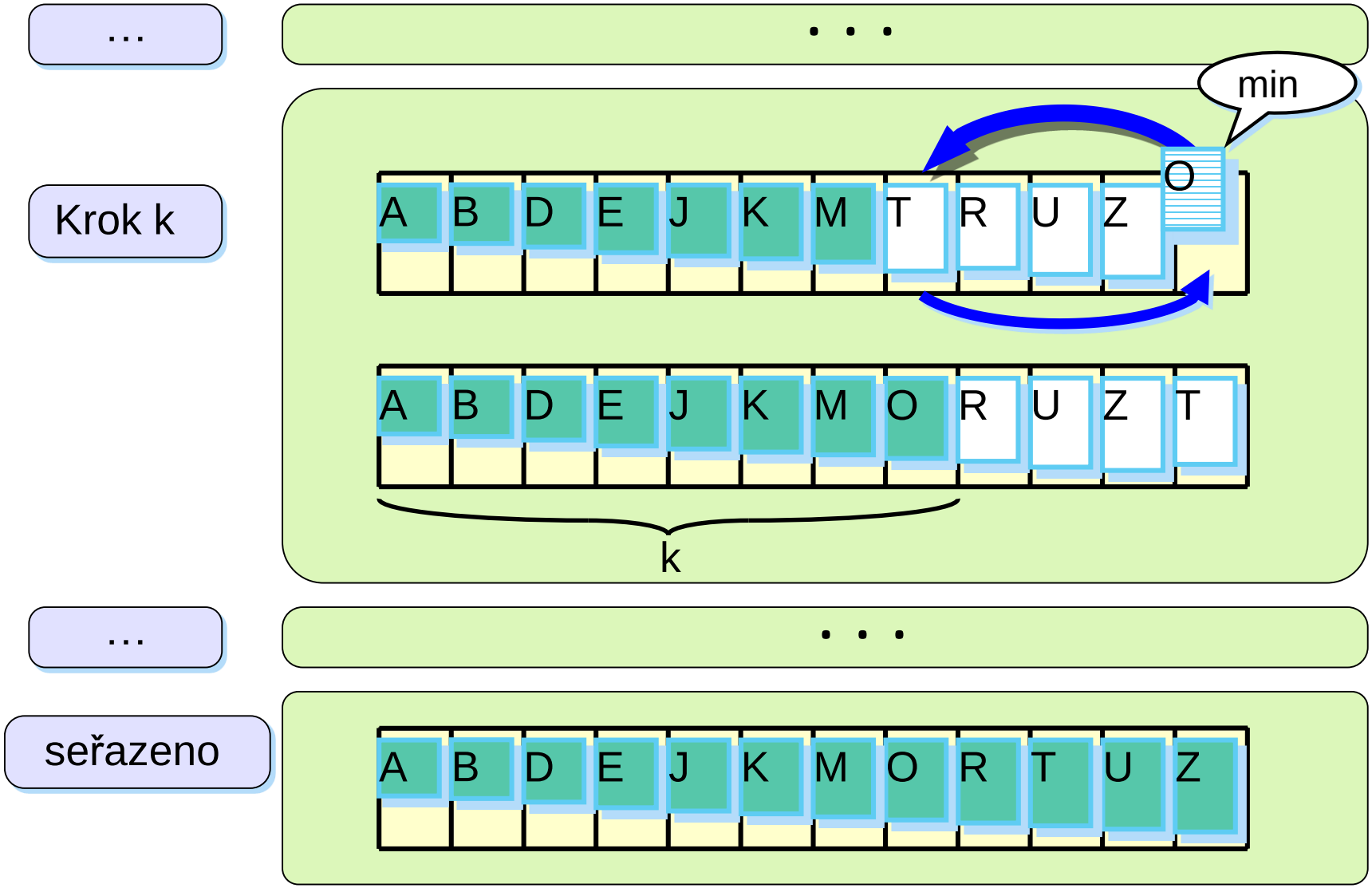
Krok 2



Krok 3



SELECTION-SORT



Pseudokód pro SELECTION-SORT

```
SELECTION-SORT(A)
  n = A.length
  for j = 1 to n − 1
      smallest = j
      for i = j + 1 to n
          if A[i] < A[smallest]
              smallest = i
      exchange A[j] with A[smallest]
```

- Algoritmus udržuje invariant, že na začátku každé iterace ve vnější smyčce pro *j* obsahuje podpole *A*[1..*j*-1] *j*-1 nejmenších elementů v poli *A*[1..*n*] a tyto elementy jsou seřazeny podle velikosti.
- Po prvých *n*-1 krocích podpole A[1..*n*-1] obsahuje *n*-1 seřazených elementů.
- Proto element A[*n*] musí být největší element.

SELECTION-SORT

```
for (j = 0; j < n-1; j++) {
                                  // select min
  smalest = j;
  for (i = j+1; i < n; i++) {
    if (a[i] < a[smalest]) {
      smalest = i;
    }
  }
                                  // exchange min
  min = a[smalest];
  a[smalest] = a[j];
  a[j] = min;
}
```

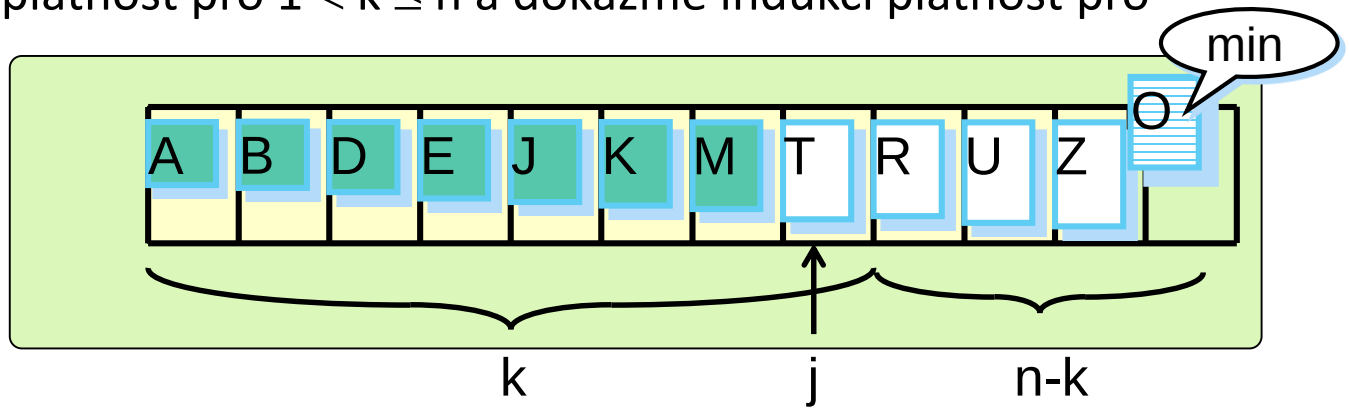
Platí požadovaný invariant?

```
for (j = 0; j < n-1; j++) {
  // tady musí platit, že začátek do j-1 je seřazen
  smalest = j;
  for (i = j+1; i < n; i++) {
    if (a[i] < a[smalest]) { smalest = i; }
  }
  min = a[smalest]; a[smalest] = a[j]; a[j] = min;
}
```

- Pro n=1 to zřejmě platí (triviálně).
- Předpokládejme platnost pro $1 < k \leq n$ a dokažme indukcí platnost pro k+1.

Pomůcka:

- j se nemění
- a[j] >= a[j-1]



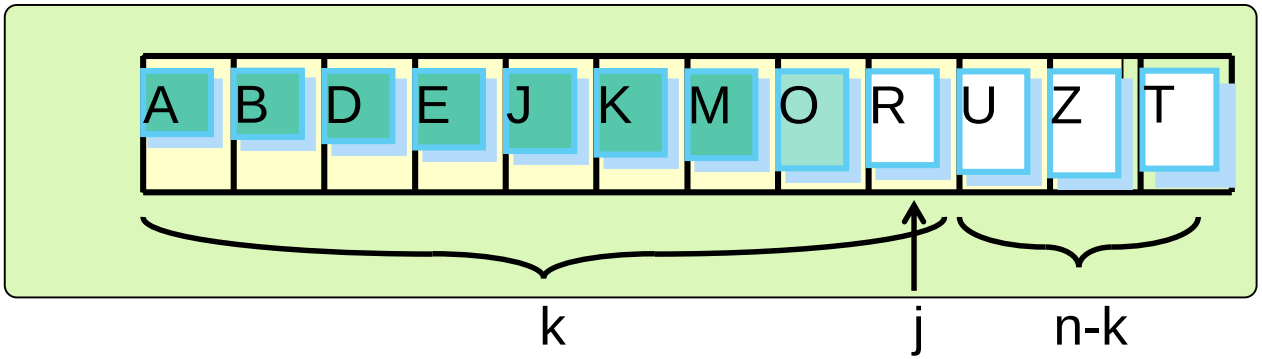
Platí požadovaný invariant?

```
for (j = 0; j < n-1; j++) {
  // tady musí platit, že začátek do j-1 je seřazen
  smalest = j;
  for (i = j+1; i < n; i++) {
    if (a[i] < a[smalest]) { smalest = i; }
  }
  min = a[smalest]; a[smalest] = a[j]; a[j] = min;
}
```

- Pro n=1 to zřejmě platí (triviálně).
- Předpokládejme platnost pro $1 < k \leq n$ a dokažme indukcí platnost pro k+1.

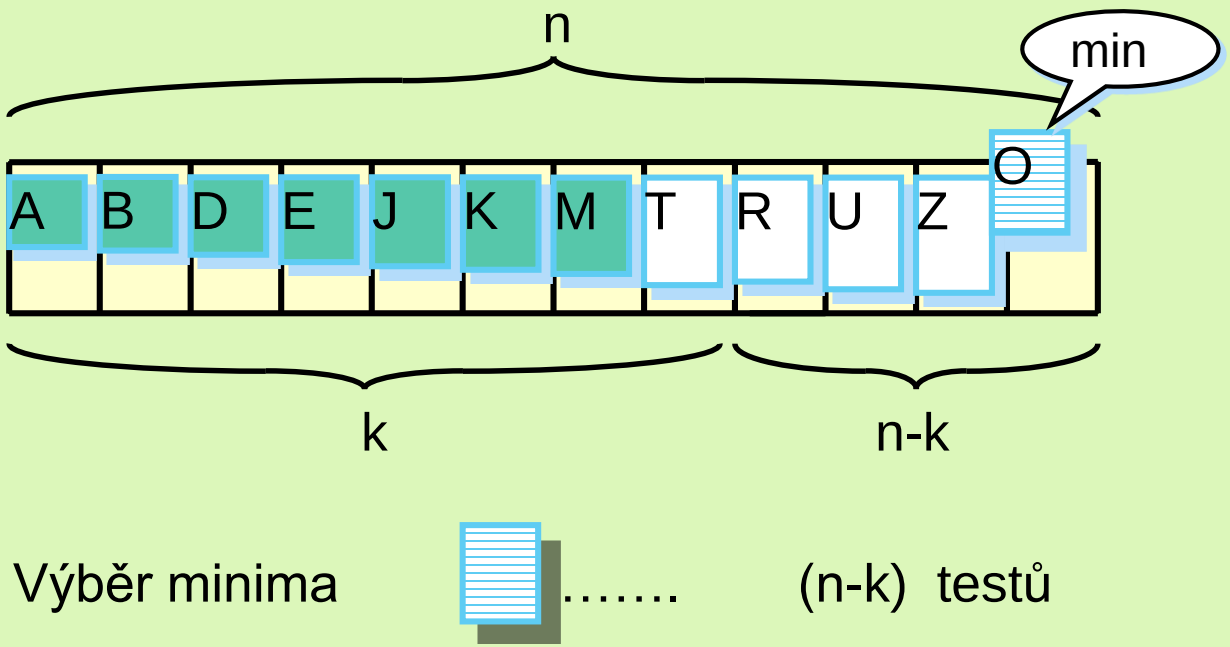
Pomůcka:

- j se nemění
- a[j] >= a[j-1]



SELECTION-SORT

Krok k

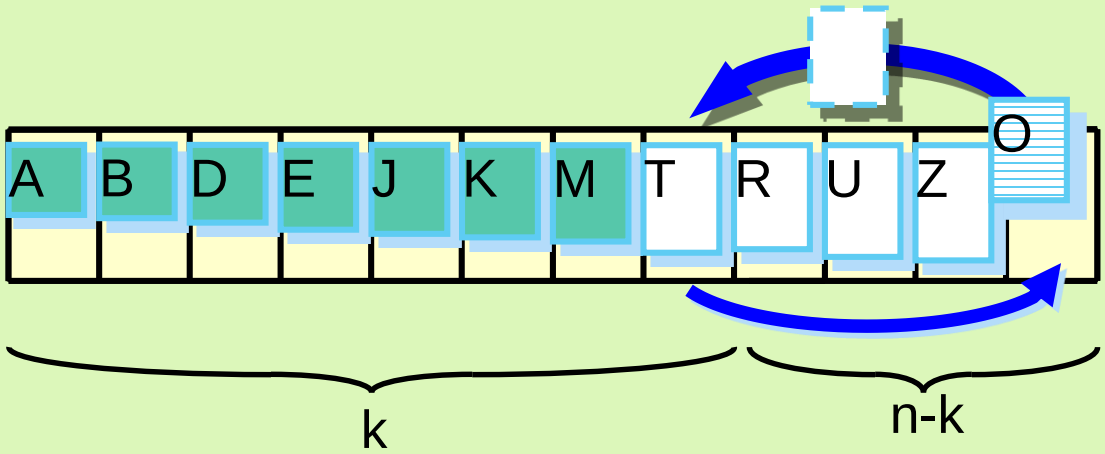


Celkem testů

$$\sum_{k=1}^{n-1}(n-k) = \sum_{k=1}^{n-1} n - \sum_{k=1}^{n-1} k = n(n-1) - \frac{n(n-1)}{2} = \boxed{\frac{1}{2}(n^2 - n)}$$

SELECTION-SORT

Krok k



přesuny ……. 3

Celkem přesunů

$$\sum_{k=1}^{n-1} 3 = \boxed{3(n-1)}$$

SELECTION-SORT

Shrnutí

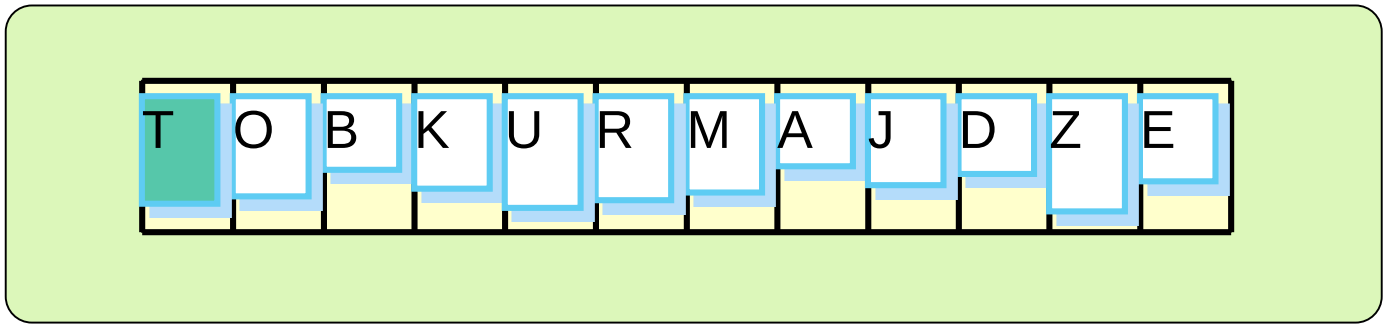
Celkem testů	$\frac{1}{2}(n^2 - n) = \Theta(n^2)$
Celkem přesunů	$3(n-1) = \Theta(n)$
Celkem operací	$\frac{1}{2}(n^2 - n) + 3(n-1) = \Theta(n^2)$

Asymptotická složitost SELECTION-SORT je $\Theta(n^2)$

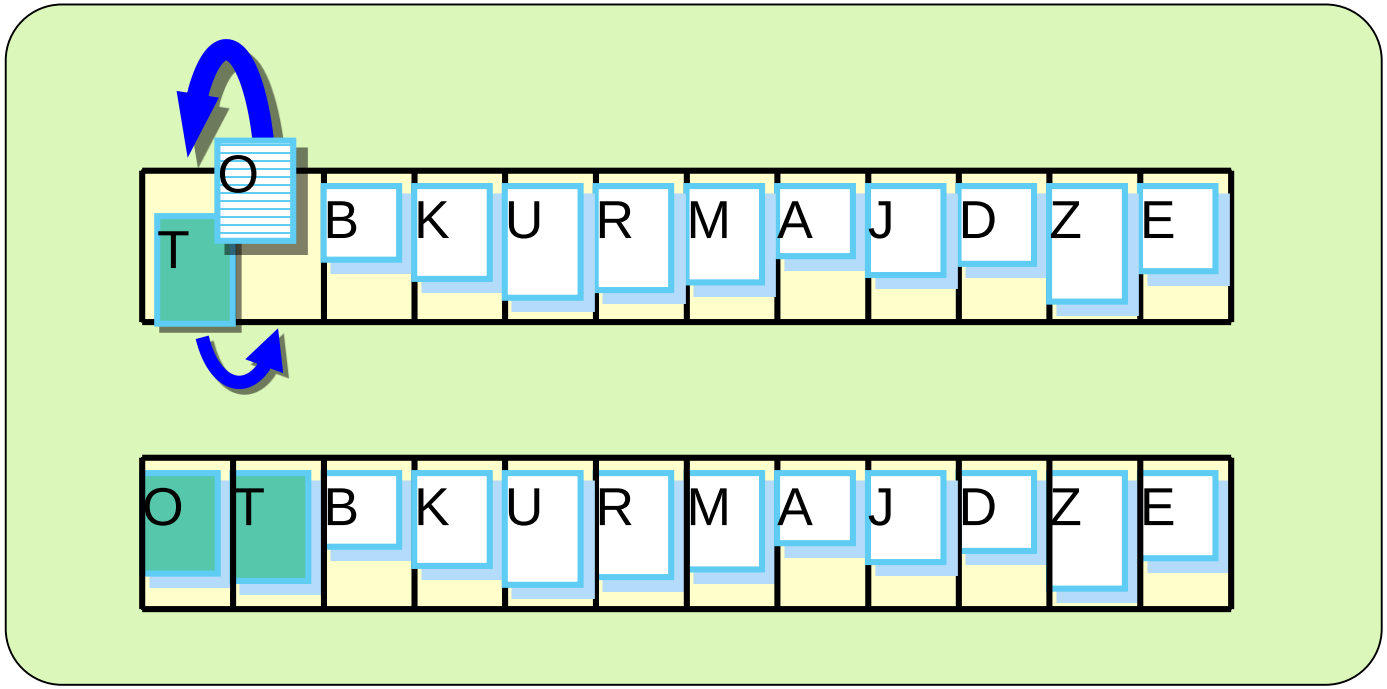
INSERTION-SORT
řazení přímým vkládáním

INSERTION-SORT

Start

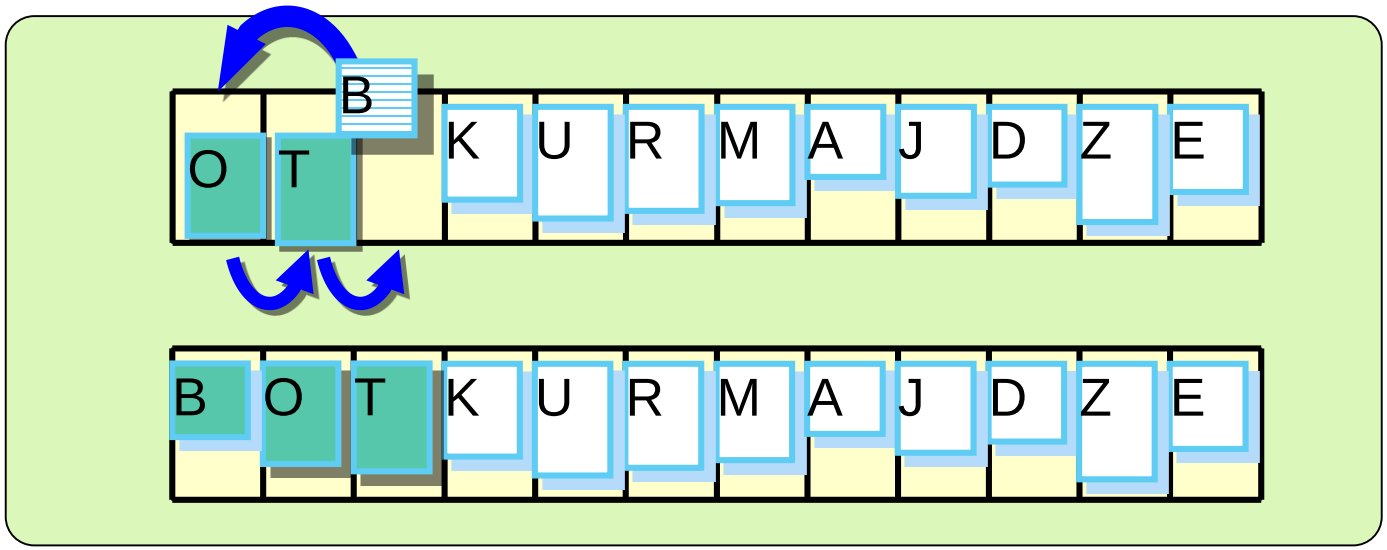


Krok 1

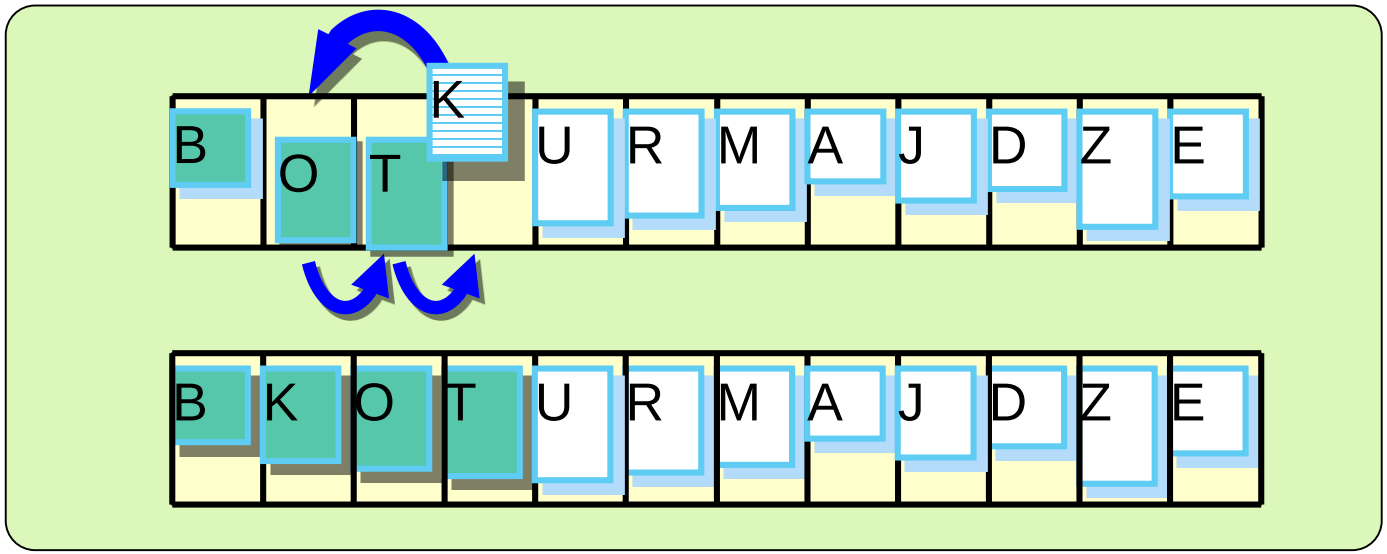


INSERTION-SORT

Krok 2



Krok 3

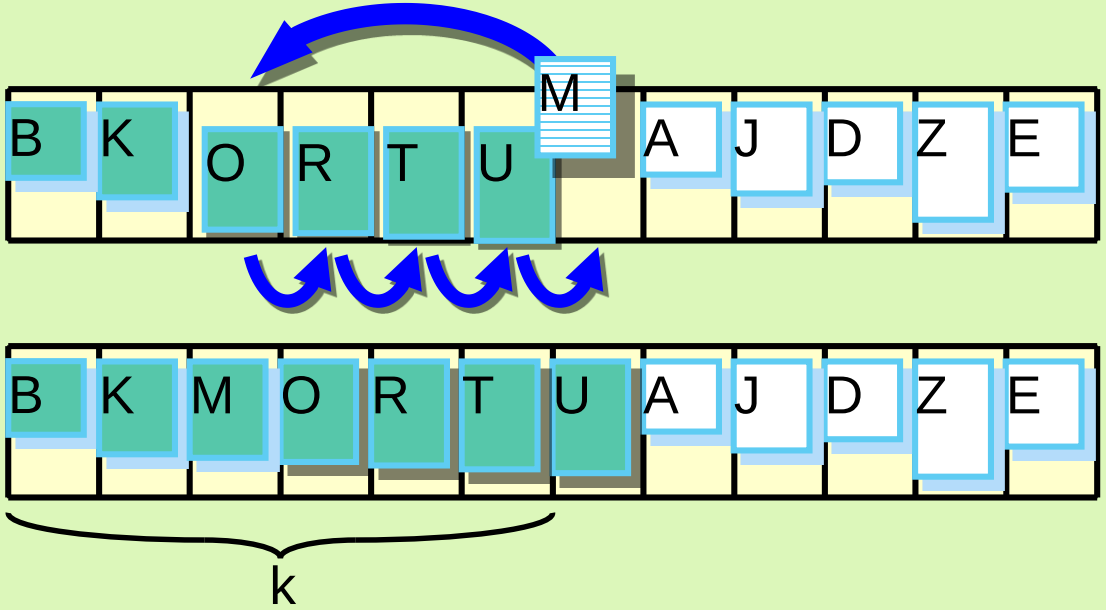


INSERTION-SORT

…

. . .

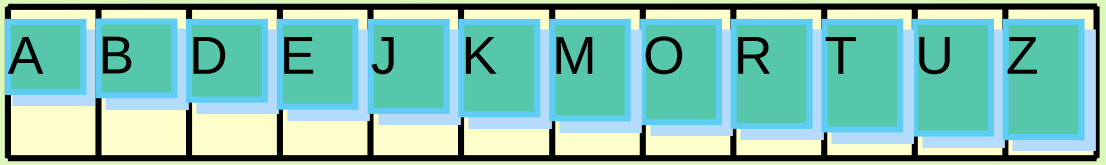
Krok k



…

. . .

seřazeno



INSERTION-SORT

```
for (j = 1; j < n; j++) {
                                // find & make
                                // place for a[j]
  insVal = a[j];
  i = j-1;
  while ((i >= 0) && (a[i] > insVal)) {
    a[i+1] = a[i];
    i--;
  }
                                // insert a[j]
  a[i+1] = insVal;
}
```

Korektnost algoritmu INSERTION-SORT

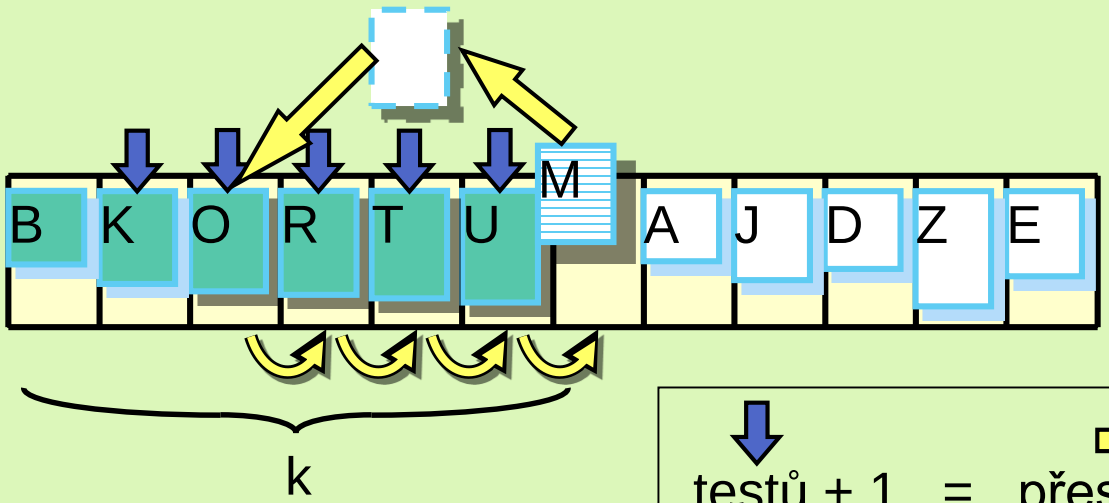
- Abychom zjistili, že algoritmus vždy dává korektní výsledky, zpravidla dokazujeme ***invariant algoritmu***.
- **Invariant pro INSERTION-SORT**: Na začátku každé vnější iterace pro j obsahuje podpole $A[1..j\text{-}1]$ j-1 nejmenších elementů v poli $A[1..n]$ a tyto elementy jsou seřazeny podle velikosti.
- Musíme dokázat 3 věci:
  - **Počáteční krok (initialization):** Invariant platí před první iterací.
  - **Indukční krok (maintenance):** Pokud invariant platí před iterací, zůstane v platnosti i po iteraci (před další iterací).
  - **Terminace:** Algoritmus musí po konečném počtu iterací skončit.
- Tyto vlastnosti zajistí, že algoritmus je **korektní**.

Korektnost INSERTION-SORT

- **Počáteční krok (initialization):** před první iterací je j = 1. Proto podpole o jednom prvku a[0] je triviálně seřazeno.
- **Indukční krok (maintenance):** V podstatě se jedná o invariant vnitřního cyklu (**while**). Předpoklad je, že a[0..j-1] je seřazeno. Vybraný prvek a[j] je zatříděn do tohoto úseku před nejblíže větší prvek. Zřejmě tedy bude po jeho vložení seřazen i úsek a[0..j].
- **Terminace:** Vnější cyklus skončí, když $j \geq n$, tj. j = n. Navíc je úsek a[0..n] seřazen vzestupně.

INSERTION-SORT

Krok k



testů + 1 = přesunů

testů …….	1	nejlepší případ
	k	nejhorší případ
	(k+1)/2	průměrný případ

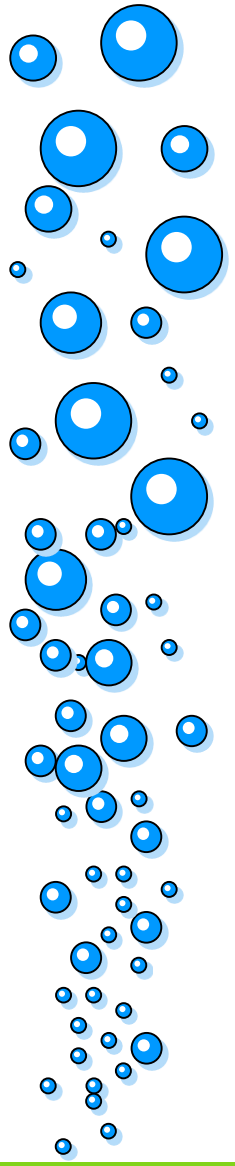
přesunů …….	2	nejlepší případ
	k+1	nejhorší případ
	(k+3)/2	průměrný případ

INSERTION-SORT

Shrnutí

Celkem testů	$n - 1$	$= \Theta(n)$	nejlepší případ
	$(n^2 - n)/2$	$= \Theta(n^2)$	nejhorší případ
	$(n^2 + n - 2)/4$	$= \Theta(n^2)$	průměrný případ
Celkem přesunů	$2n - 2$	$= \Theta(n)$	nejlepší případ
	$(n^2 + n - 2)/2$	$= \Theta(n^2)$	nejhorší případ
	$(n^2 + 5n - 6)/4$	$= \Theta(n^2)$	průměrný případ

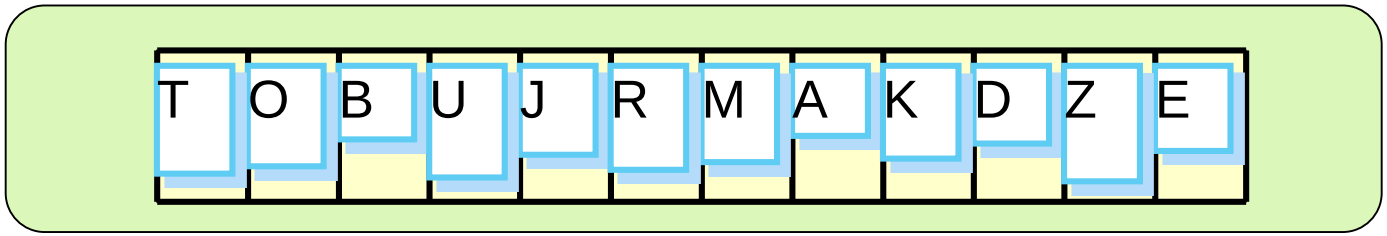
Asymptotická složitost INSERTION-SORT je $O(n^2)$ (!!)



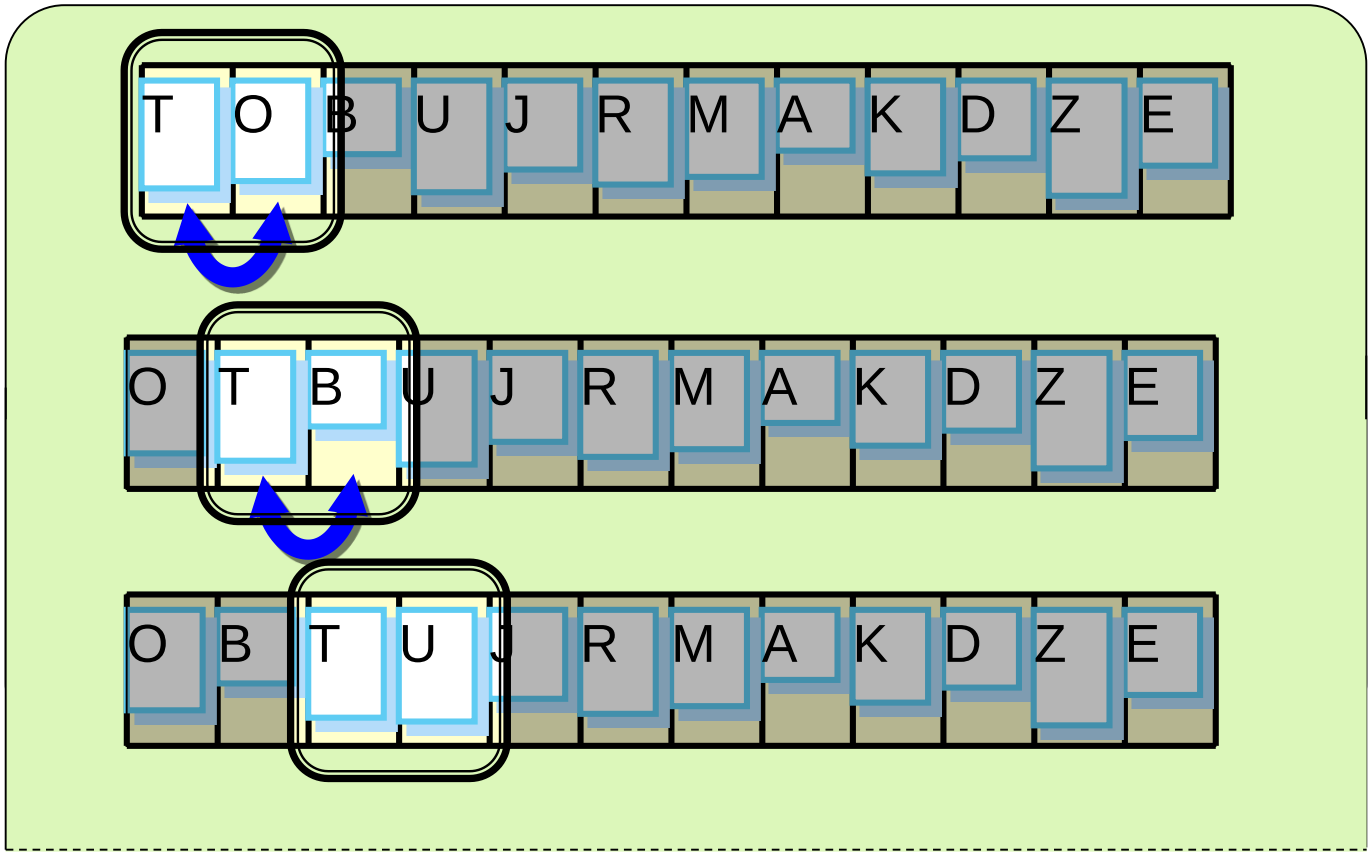
Bubble-Sort
bublinkové řazení

BUBBLE-SORT

Start

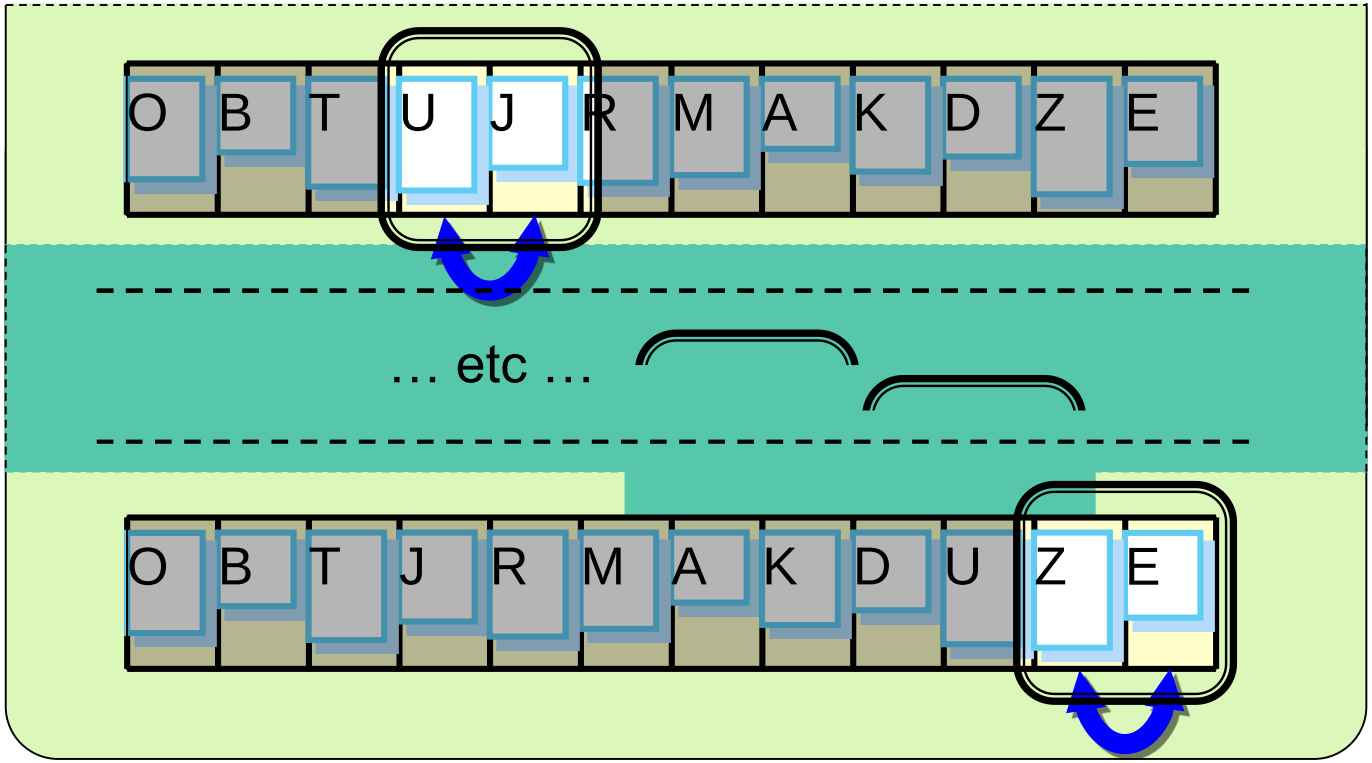


Fáze 1

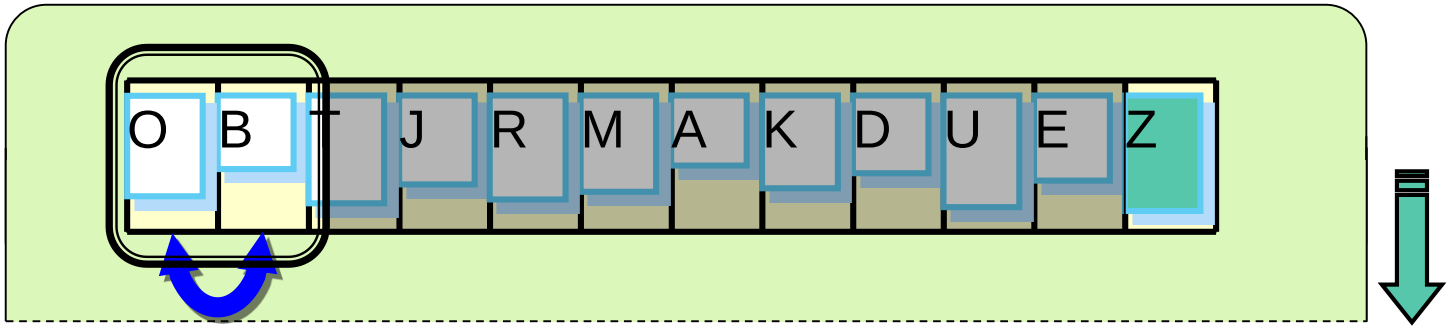


Bubble-Sort

Fáze 1

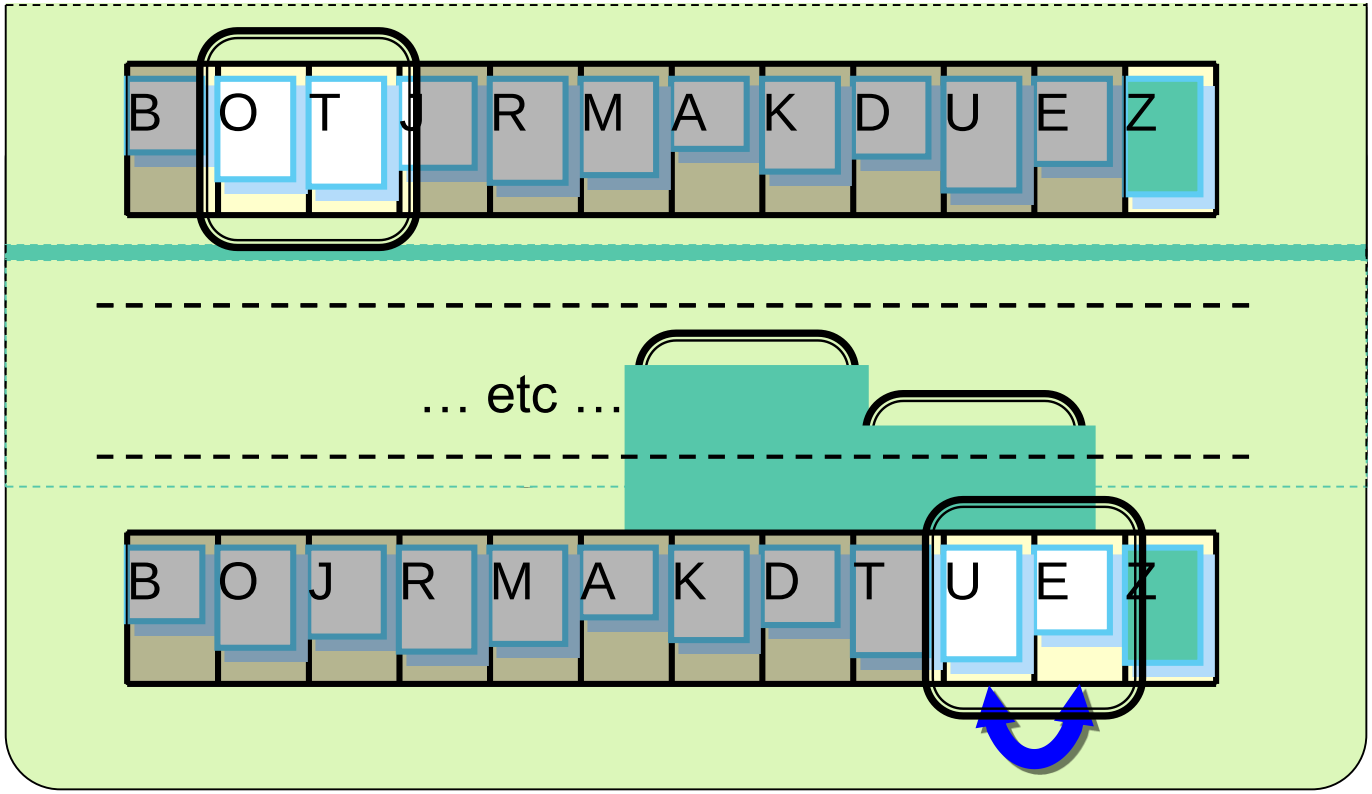


Fáze 2

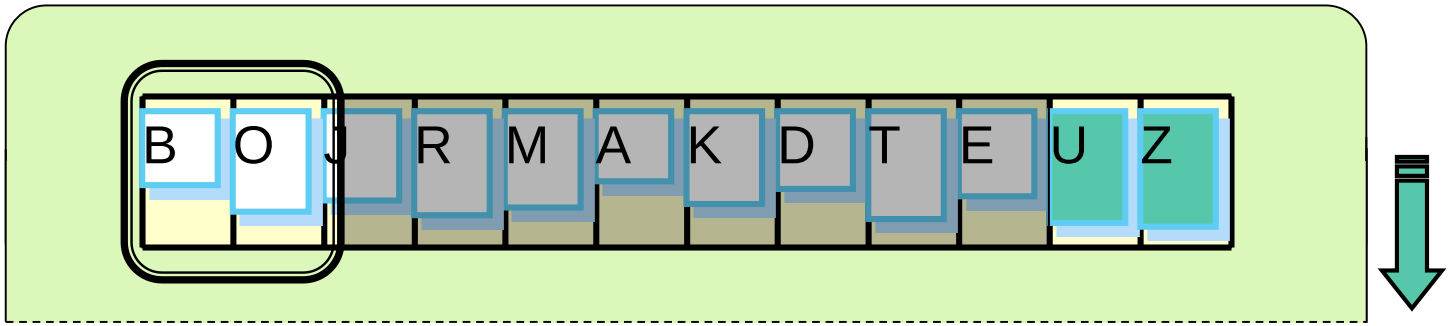


Bubble-Sort

Fáze 2

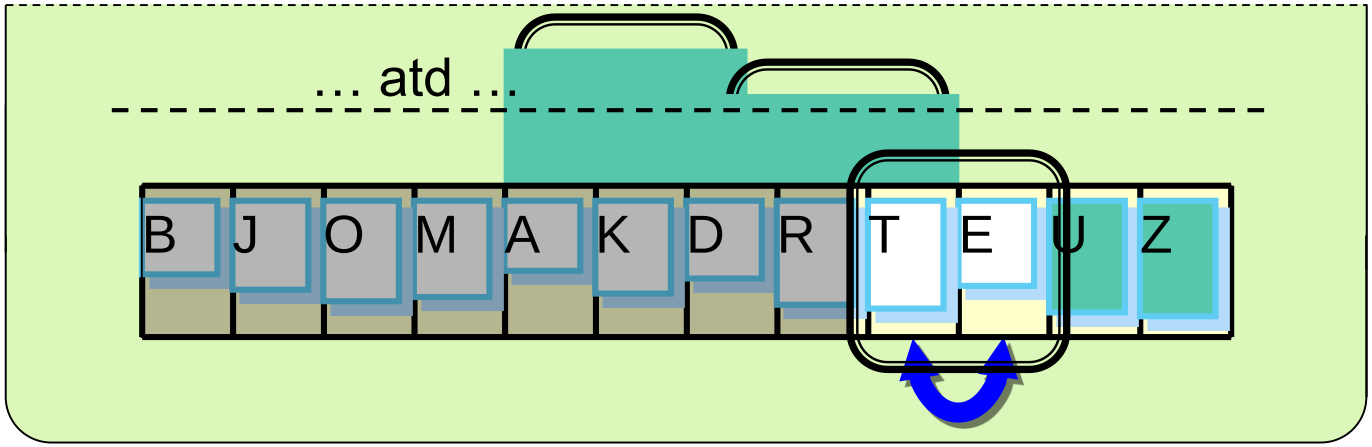


Fáze 3

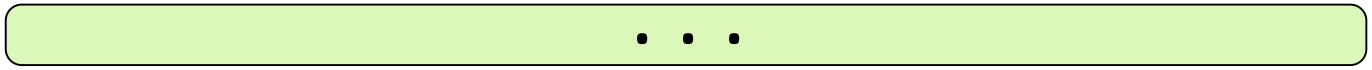


BUBBLE-SORT

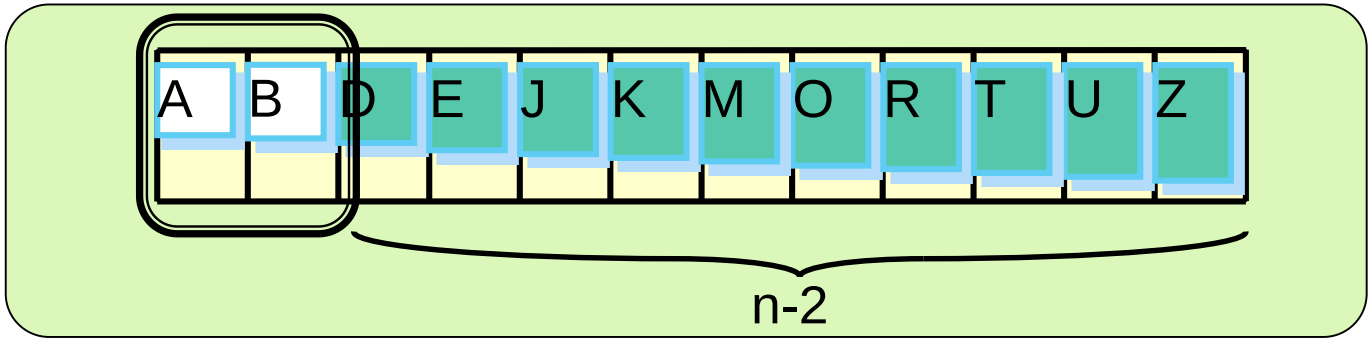
Fáze 3



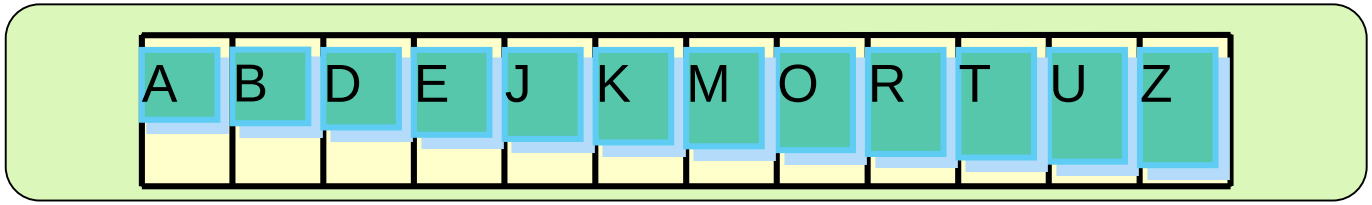
...



Fáze n-1



seřazeno



BUBBLE-SORT

```
for (lastPos = n-1; lastPos > 0; lastPos--)
  for (j = 0; j < lastPos; j++)
    if (a[j] > a[j+1]) swap(a, j, j+1);
```

Shrnutí

Celkem testů

$$(n-1) + (n-2) + \ldots + 2 + 1 = \frac{1}{2}(n^2 - n) = \Theta(n^2)$$

Celkem přesunů

$$0 = \Theta(1) \quad \text{nejlepší případ}$$

$$\frac{1}{2}(n^2 - n) = \Theta(n^2) \quad \text{nejhorší případ}$$

Asymptotická složitost BUBBLE-SORT je $\Theta(n^2)$

SHELL-SORT
řazení se snižujícím se příspěvkem

Řazení metodou SHELLSORT

- **SHELLSORT** nebo též řazení se snižujícím se přírůstkem je řadicí algoritmus založený na principu bublinkového vkládání, který objevil a v roce 1959 publikoval Donald Shell.
- V několika průchodech se řadí prvky od sebe stejně vzdálené pomocí bublinkového vkládání. Vzdálenost prvků od sebe *(krok)* se pak postupně snižuje, až je rovna jedné, tj. klasický BUBBLESORT.
- Bublinkové řazení porovnává a vyměňuje sousední prvky. **SHELLSORT** se snaží nejprve vyměnit prvky vzdálenější a teprve poté prvky bližší. Vzdálené prvky pak nemusí „probublávat“ celým polem.
- Praktický problém je stanovení „délky“ kroku.

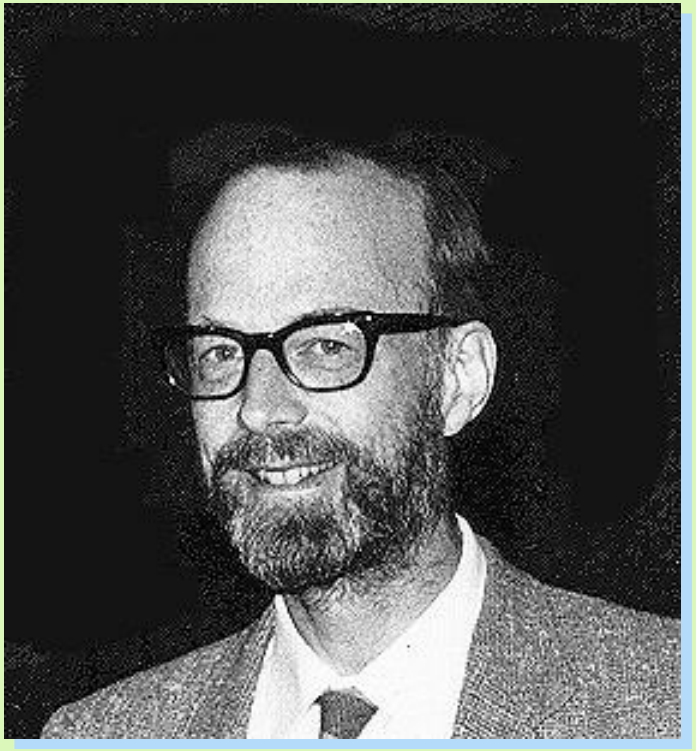
Implementace v Javě

```
public static void shellSort(int[] a) {
  for (int krok = a.length / 2;
       krok > 0;
       krok = (krok == 2 ? 1 :
                 (int)Math.round(krok / 2.2))) {
    for (int i = krok; i < a.length; i++) {
      int temp = a[i];
       for (int j = i;
            j >= krok && a[j - krok] > temp;
            j -= krok) {
          a[j] = a[j - krok]; // prohoď prvky v nesprávném
          a[j - krok] = temp; // pořadí
       }
    }
  }
}
```

Složitost SHELLSORT

- Teoretické analýzy nenašly nejvhodnější řadu snižujících se kroků. Kerninghan a Ritchie ve své implementaci používali krok n/2, který pak postupně dělili dvěma.
- Zde je použita úprava kroku koeficientem 2.2 podle Marcina Ciury, což se zdá vést k nejlepším výsledkům.
- **SHELLSORT** je nestabilní řadicí metoda (tj. nezachovává původní pořadí dvou prvků se stejným klíčem.
- Časová složitost metody **SHELLSORT** je přibližně rovna $n^{3/2}$. Tím se zdá být nejlepší v kategorii metod složitosti $\Theta(n^2)$.

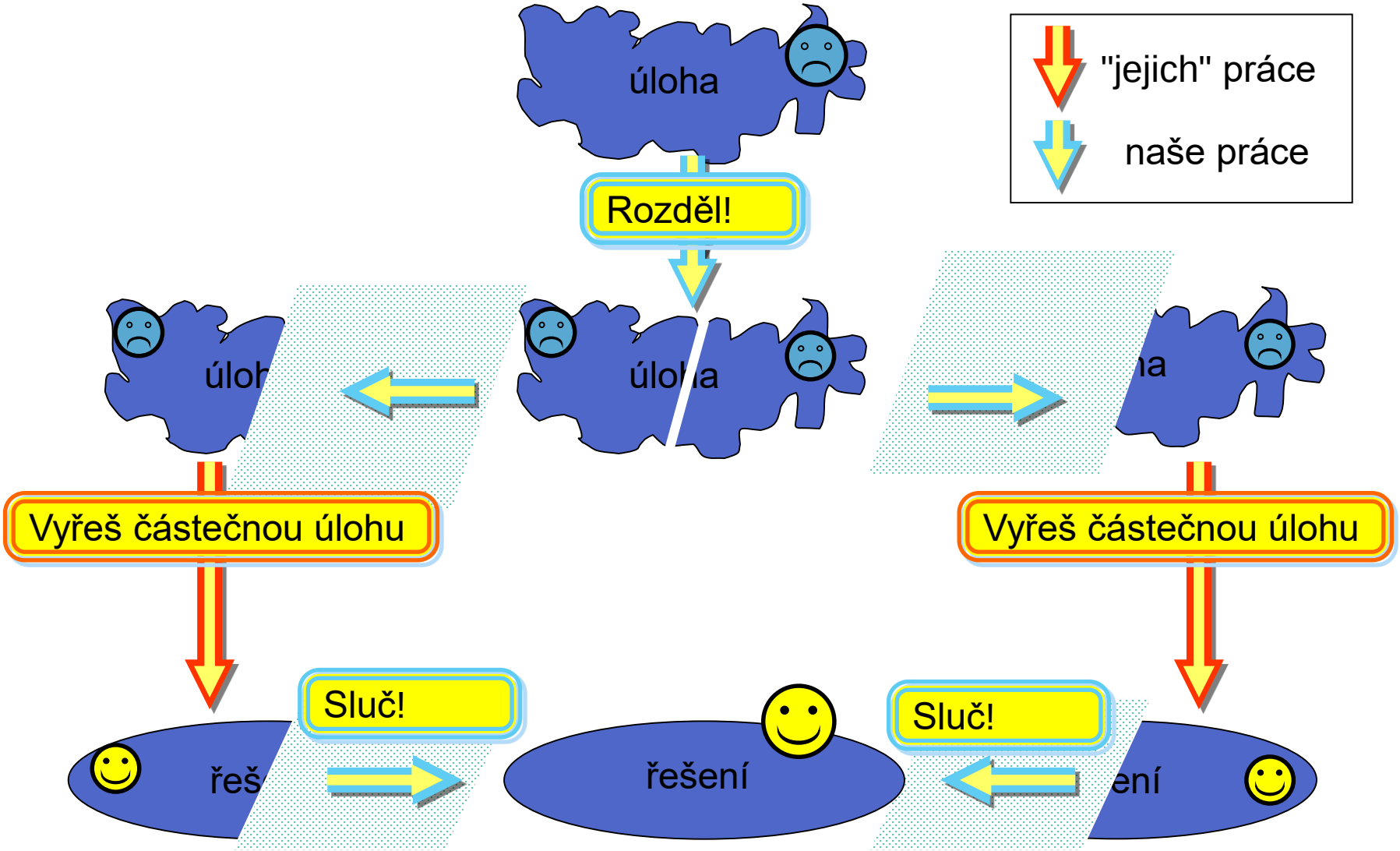
QUICK-SORT



Sir Charles Antony Richard Hoare

C. A. R. Hoare: Quicksort. Computer Journal, Vol. 5, 1, 10-15 (1962)

Rozděl a Panuj! Divide & Conquer! Divide et Impera!

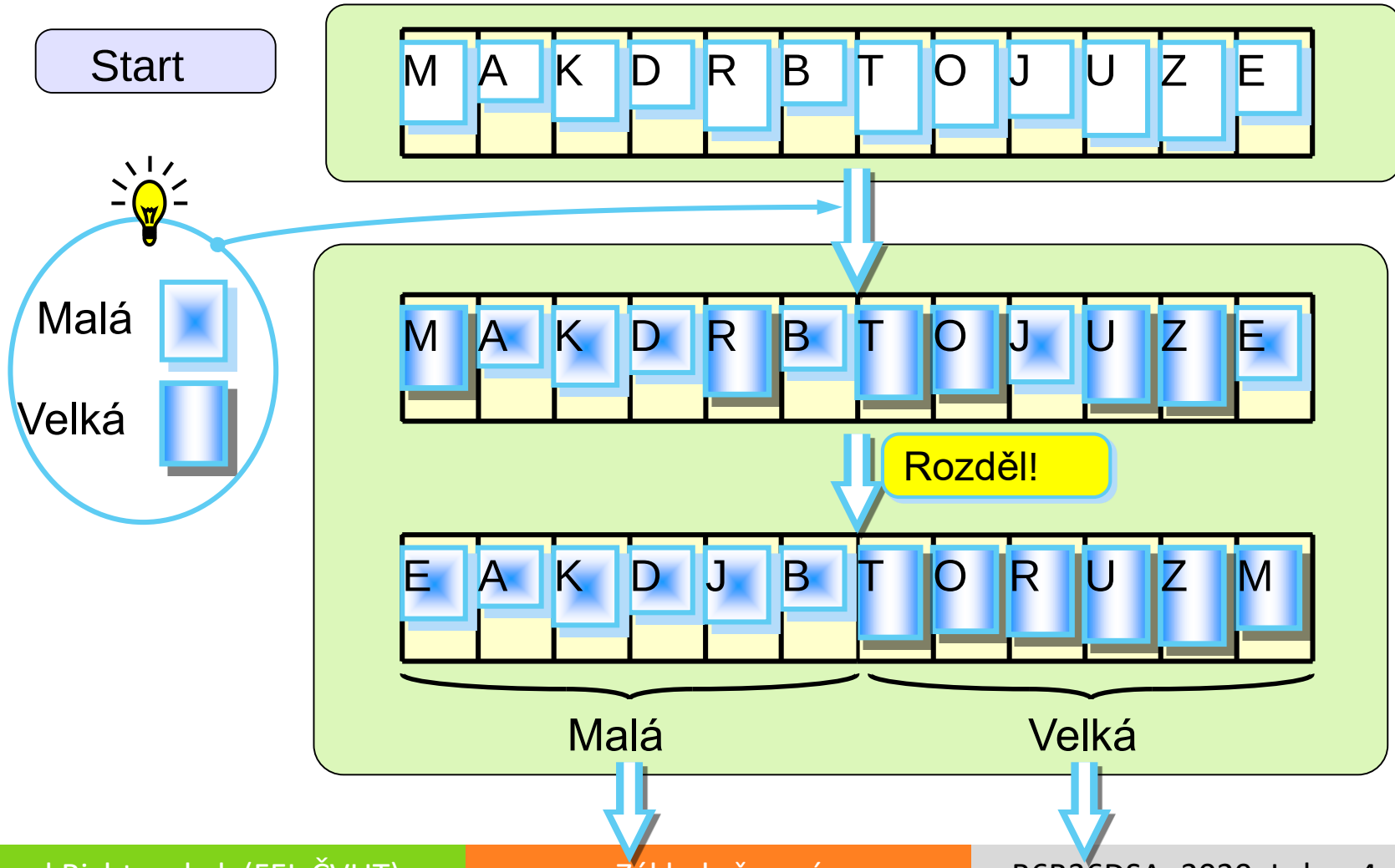


QUICK-SORT

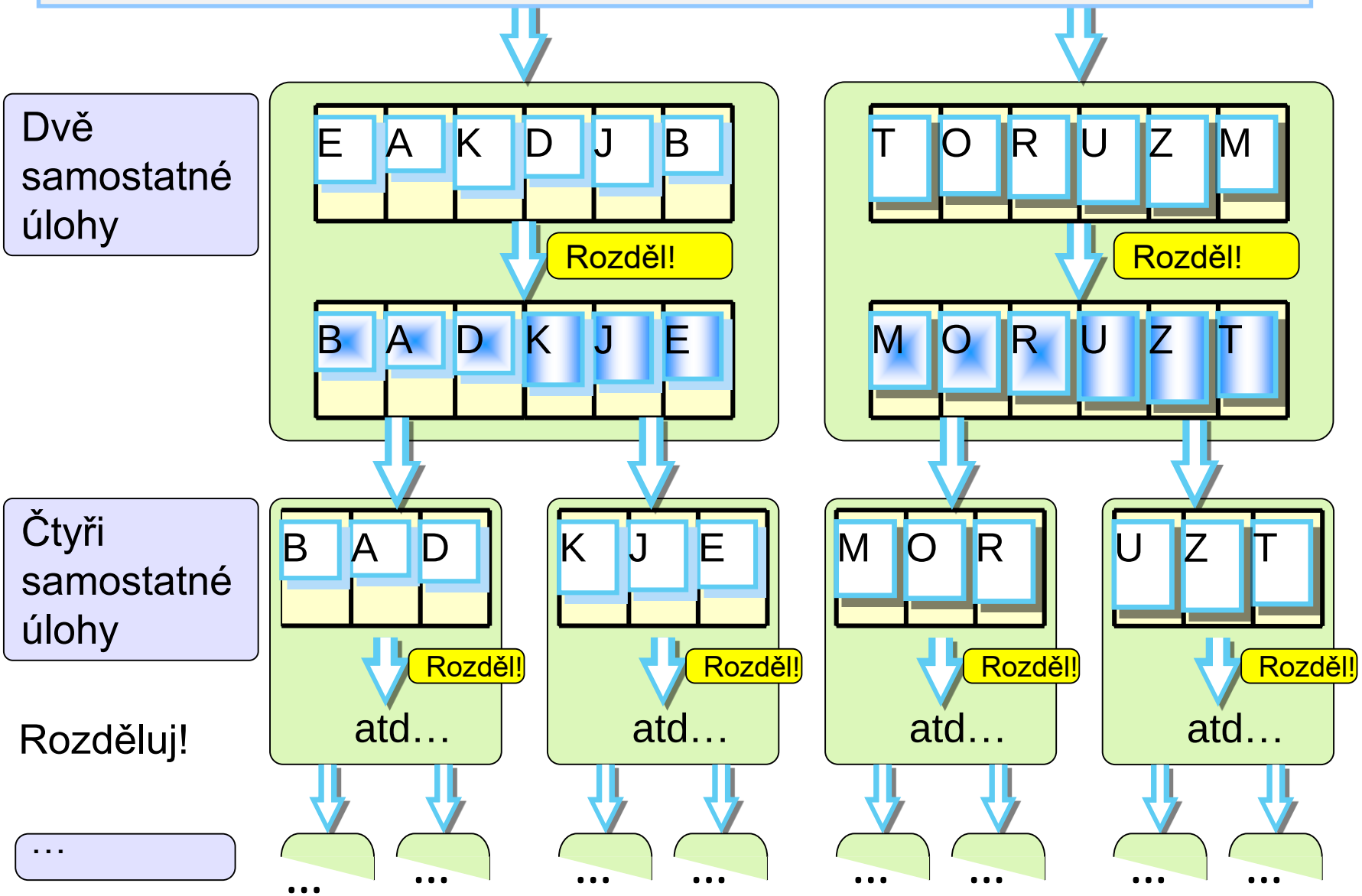
Myšlenka

Divide & Conquer!

Start

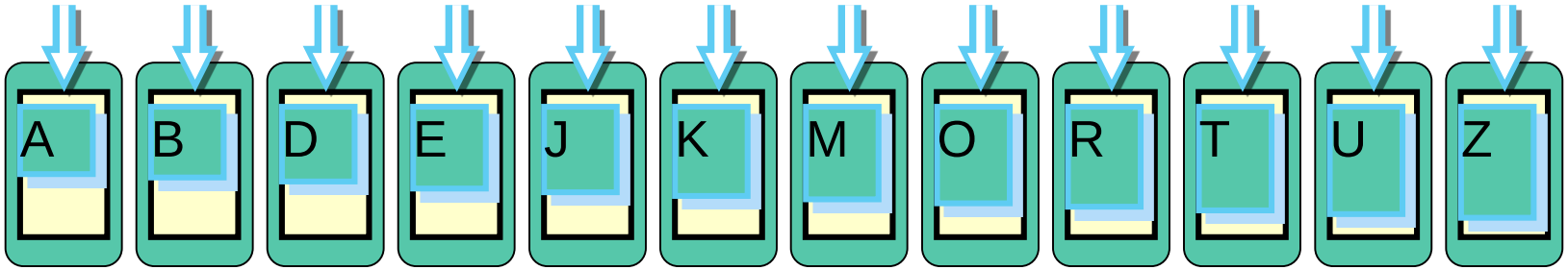


QUICK-SORT


Dvě samostatné úlohy
E A K D J B
Rozděl!
B A D K J E
T O R U Z M
Rozděl!
M O R U Z T
Čtyři samostatné úlohy
B A D
Rozděl!
atd…
K J E
Rozděl!
atd…
M O R
Rozděl!
atd…
U Z T
Rozděl!
atd…
Rozděluj!
…
…
…
…
…
…
…
…
…

QUICK-SORT

… … … … … … … … … … … … … … … … …

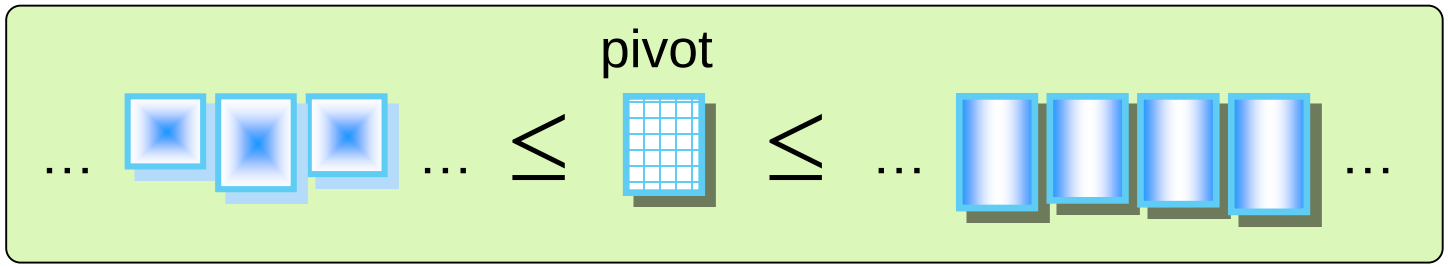


Opanováno!

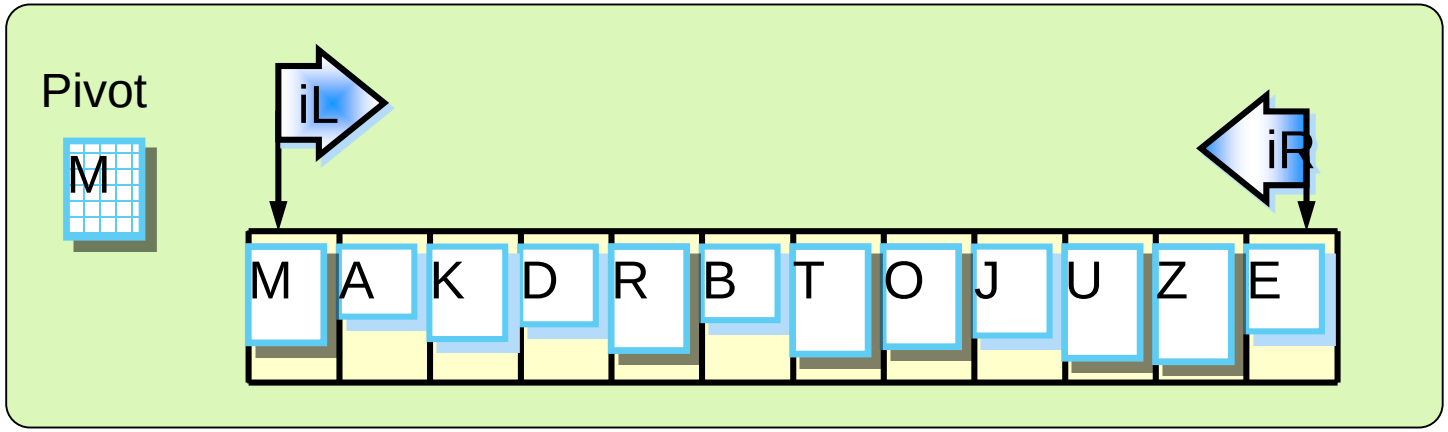
QUICK-SORT

Dělení

pivot



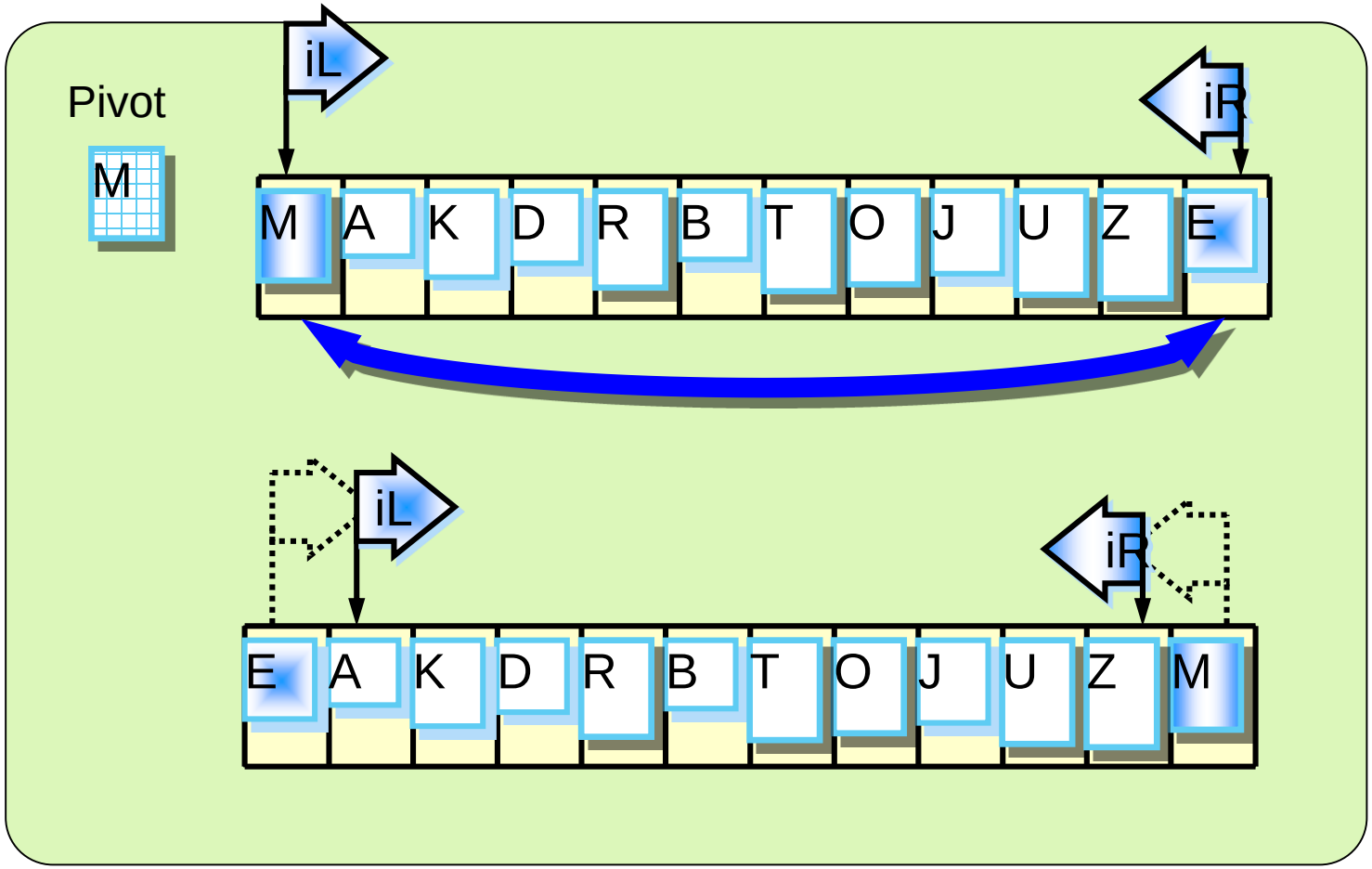
Init



QUICK-SORT

Dělení

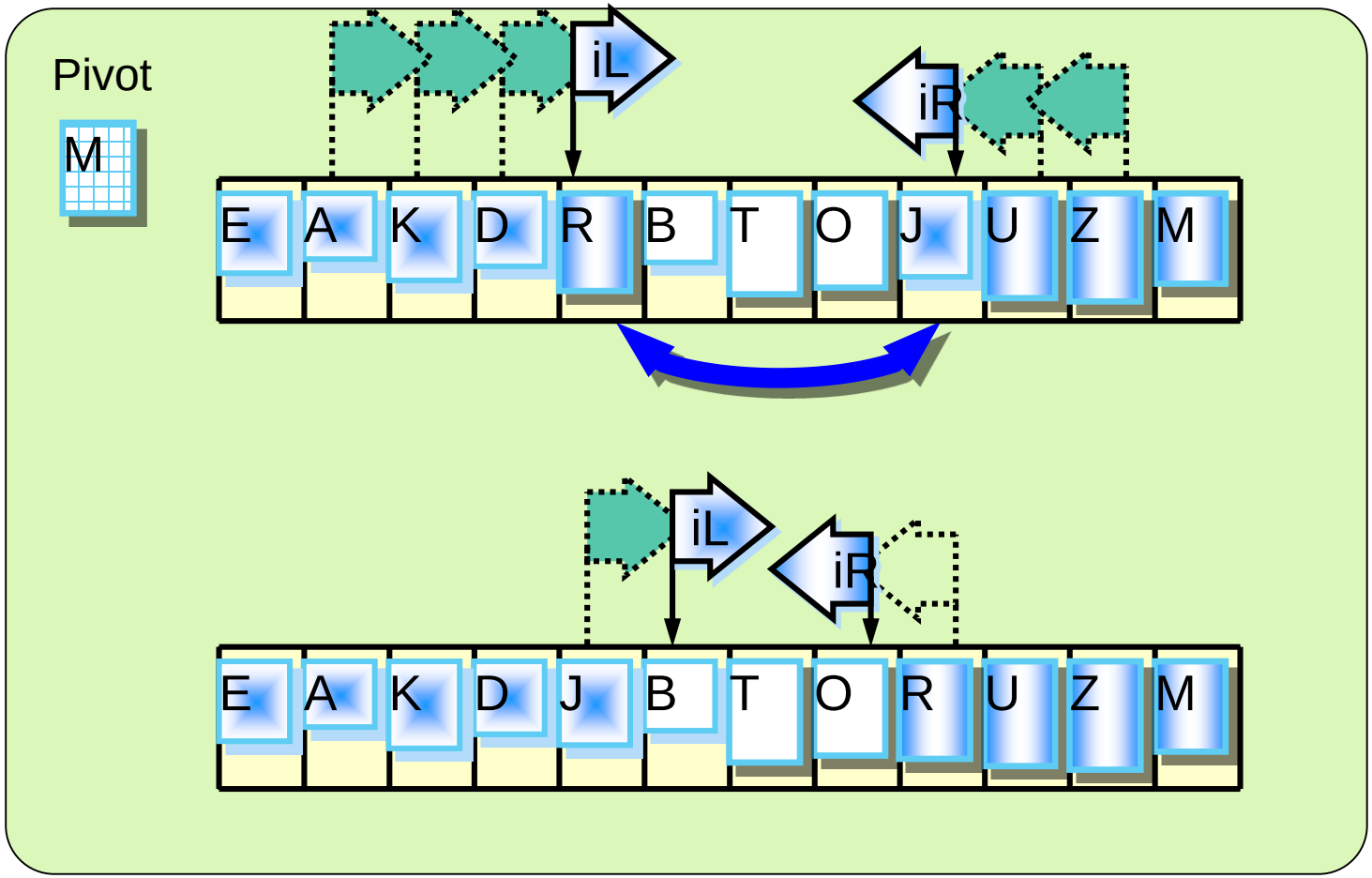
Krok 1



QUICK-SORT

Dělení

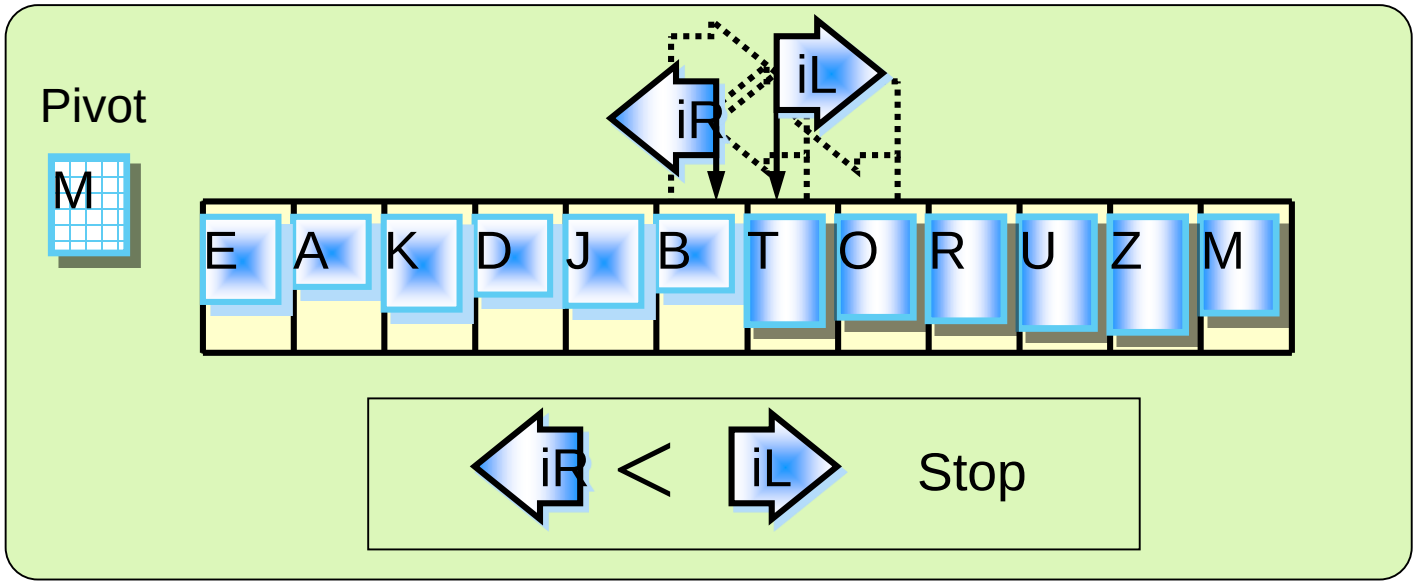
Krok 2



QUICK-SORT

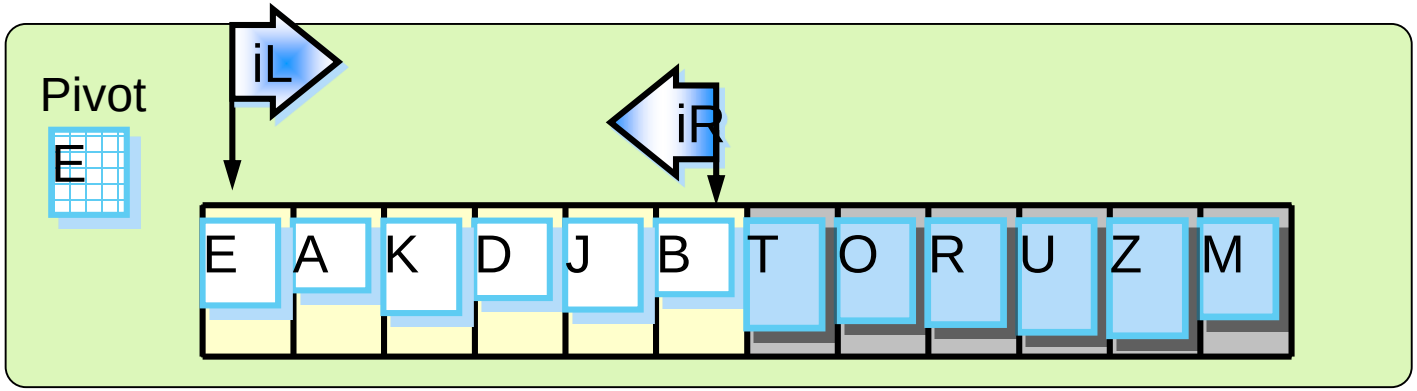
Dělení

Krok 3



Rozděl!

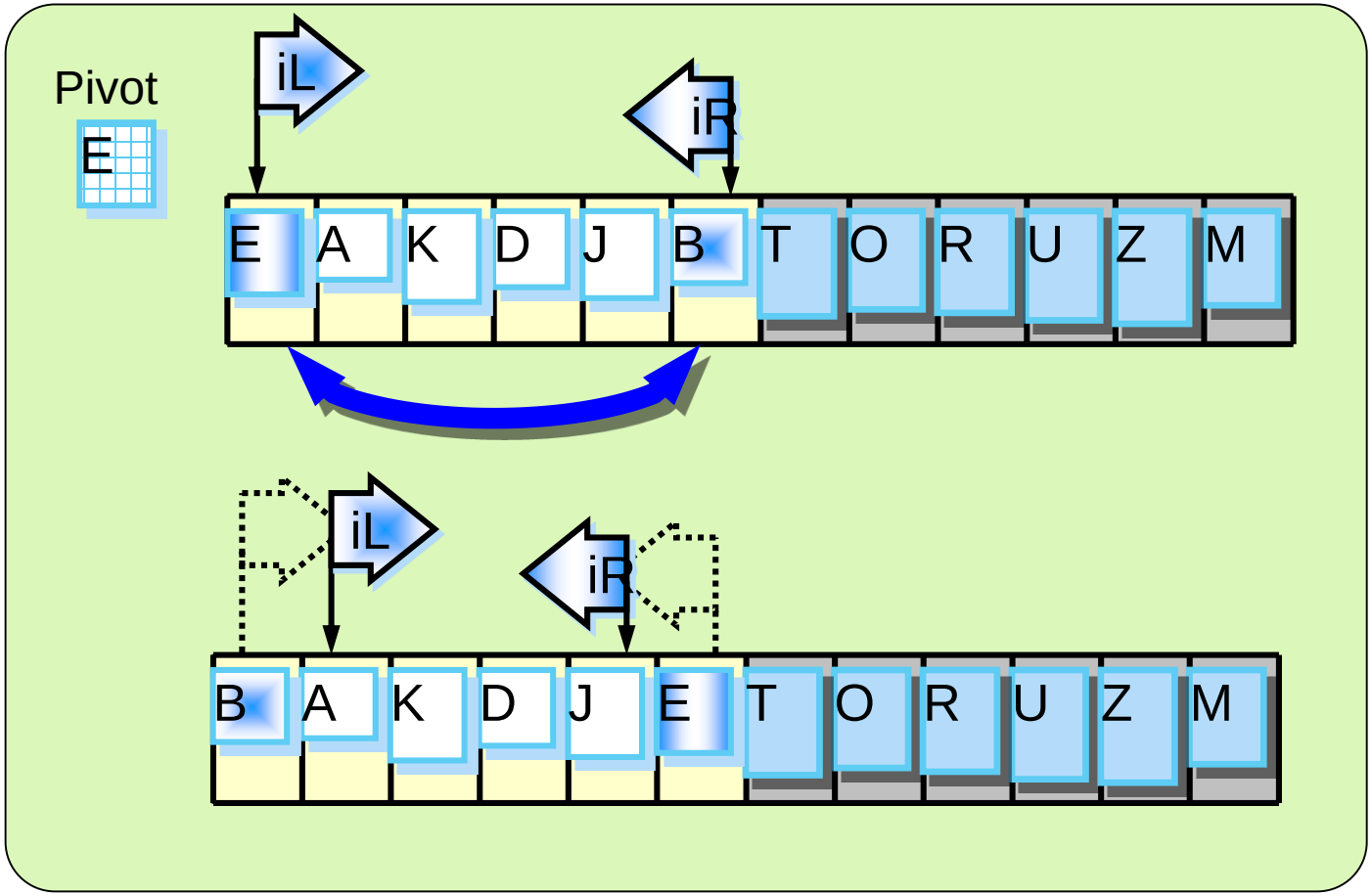
Init



QUICK-SORT

Dělení

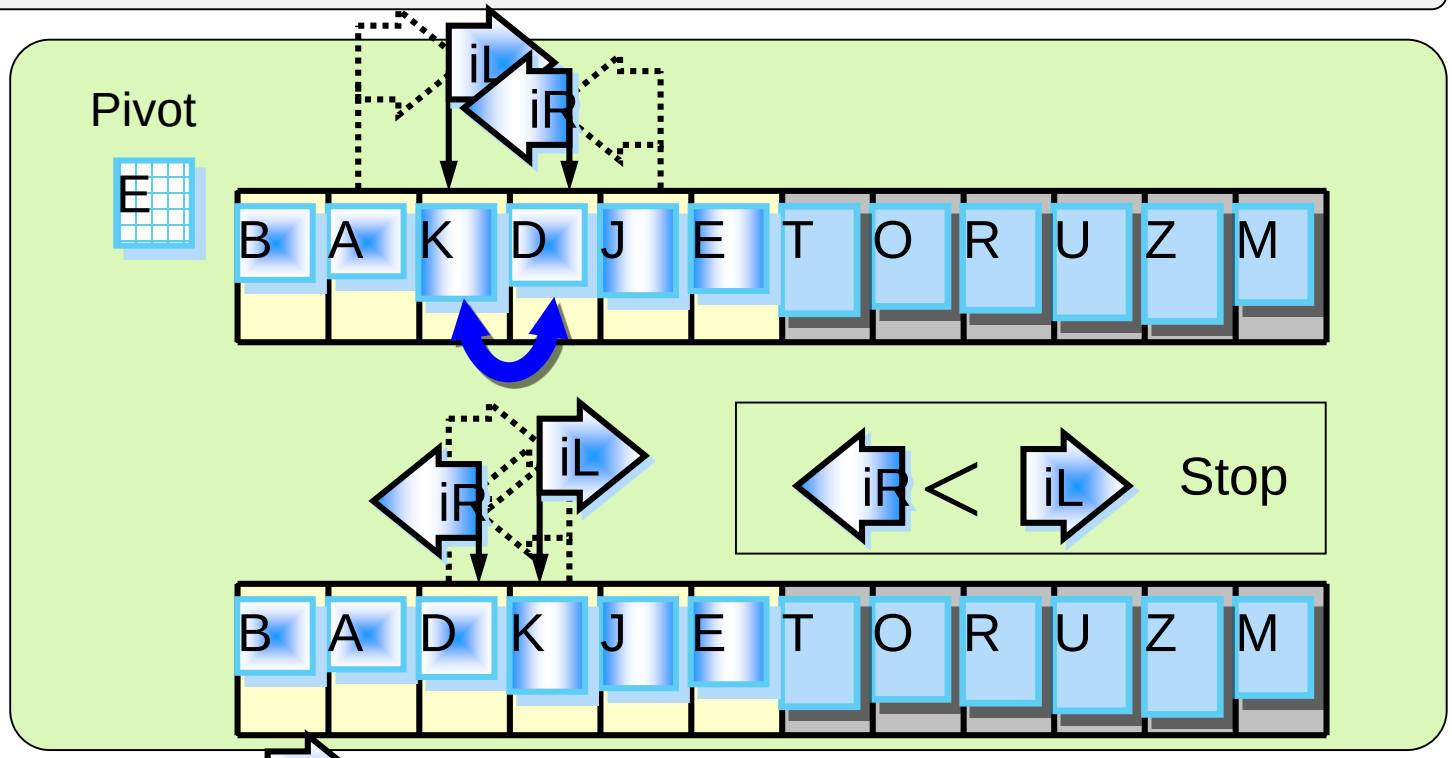
Krok 1



QUICK-SORT

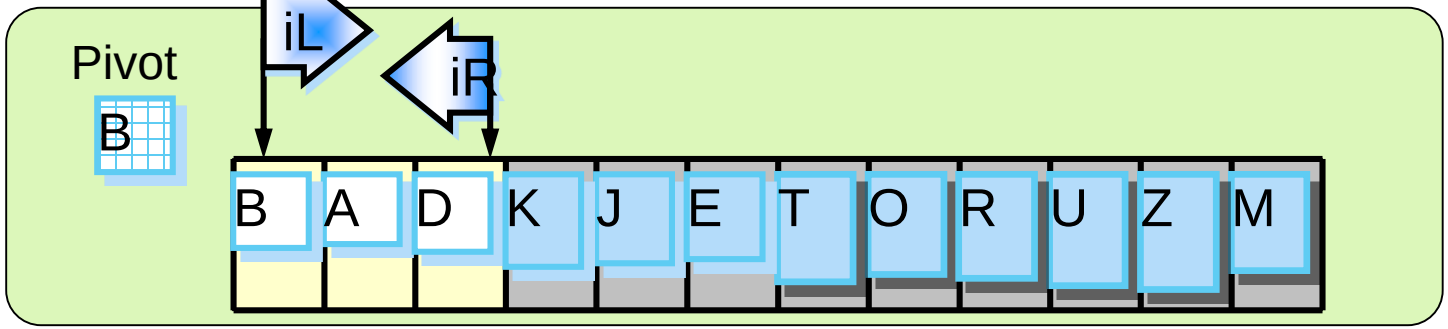
Dělení

Krok 2



Rozděl!

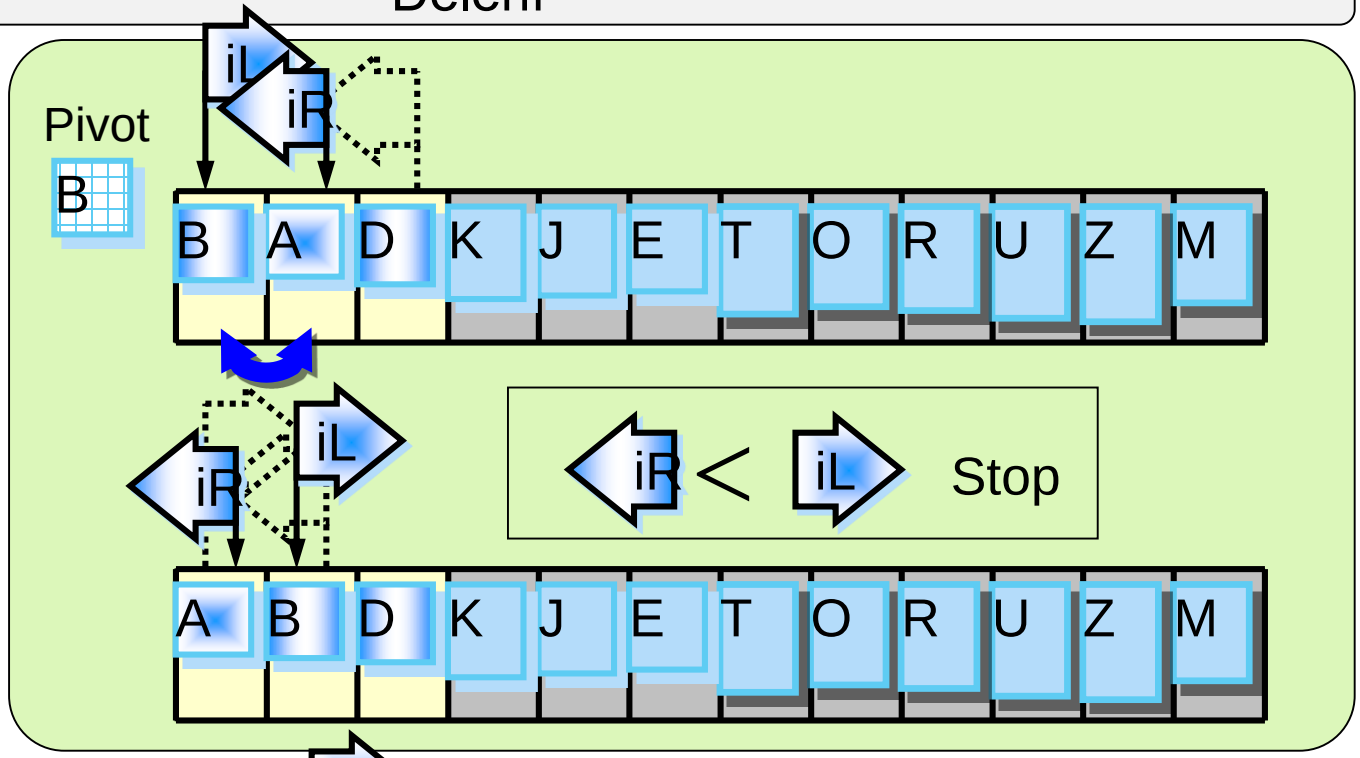
Init



QUICK-SORT

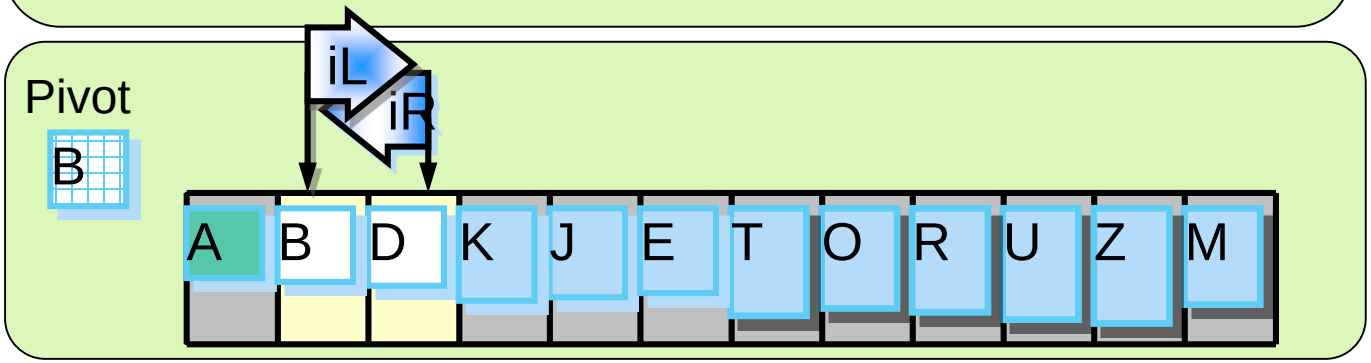
Dělení

Krok 1



Rozděl!

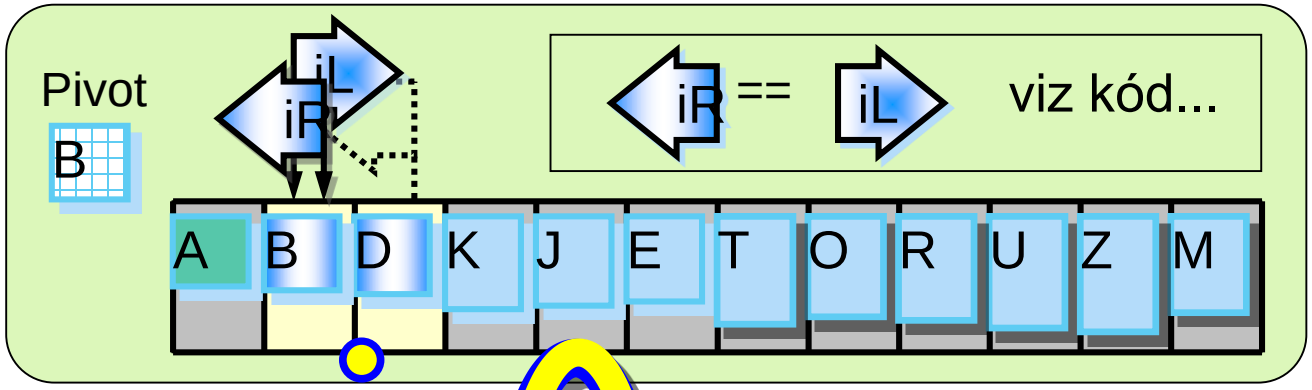
Init



QUICK-SORT

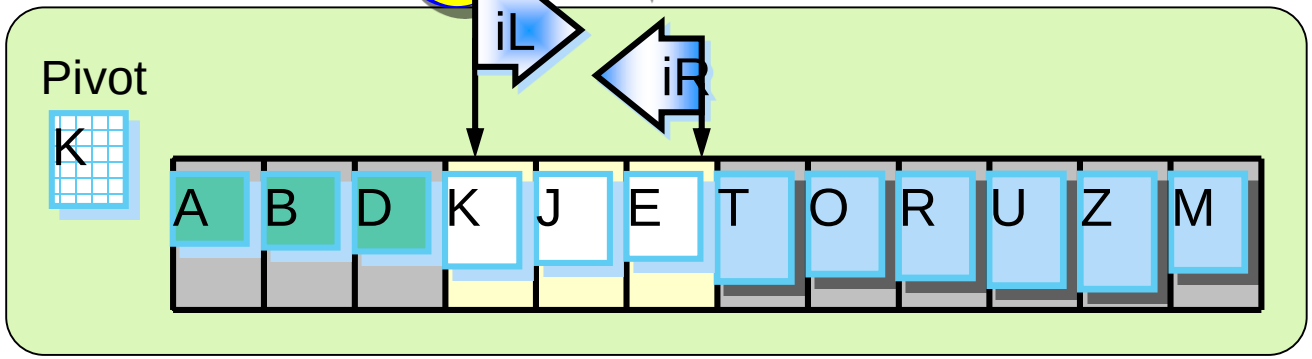
Dělení

Krok 1



Další oddíl

Init



atd…

atd…

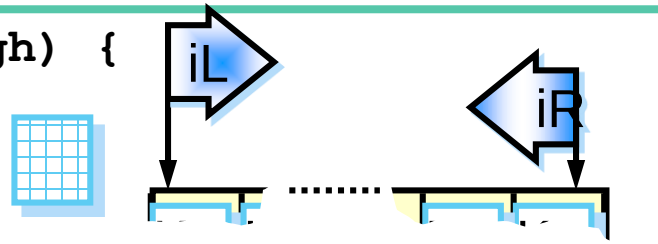
QUICK-SORT

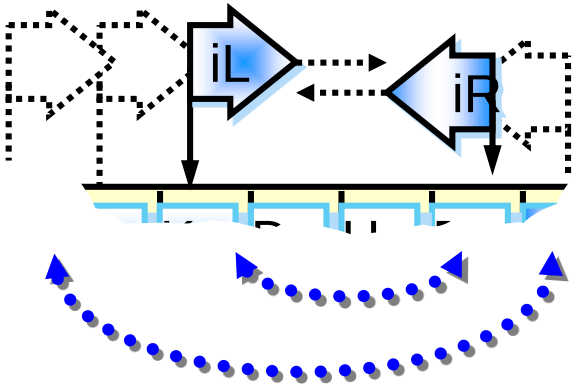
```
void qSort(Item a[], int low, int high) {
  int iL = low, iR = high;
  Item pivot = a[low];

  do {
    while (a[iL] < pivot) iL++;
    while (a[iR] > pivot) iR--;
    if (iL < iR) {
      swap(a,iL, iR);
      iL++; iR--;
    }
    else
      if (iL == iR) { iL++; iR--;}

  } while( iL <= iR);

  if (low < iR)  qSort(a, low, iR);
  if (iL < high) qSort(a, iL, high);
}
```





Rozděl!

QUICK-SORT

Levý index se nastaví na začátek zpracovávaného úseku pole, pravý na jeho konec, zvolí se pivot.

Cyklus (rozdělení na „malé" a „velké") :

Levý index se pohybuje doprava
a zastaví se na prvku vetším nebo rovném pivotovi.

Pravý index se pohybuje doleva
a zastaví se na prvku menším nebo rovném pivotovi.

Pokud je levý index ještě před pravým,
příslušné prvky se prohodí,
a oba indexy se posunou o 1 ve svém směru.

Jinak pokud se indexy rovnají,
jen se oba posunou o 1 ve svém směru.

Cyklus se opakuje, dokud se indexy neprekříží,
tj. pravý se dostane pred levého.

Následuje rekurzivní volání (zpracování „malých" a „velkých" zvlášť)
na úsek od začátku do pravého(!) indexu včetně
a na úsek od levého(!) indexu včetně až do konce,
má-li príslušný úsek délku větší než 1.

QUICK-SORT

Asymptotická složitost

Celkem přesunů a testů

$\Theta(n \cdot \log_2(n))$	nejlepší případ
$\Theta(n \cdot \log_2(n))$	průměrný případ
$\Theta(n^2)$	nejhorší případ

Asymptotická složitost QUICK-SORT je $O(n^2)$, ...

… ale! :

"Očekávaná" složitost QUICK-SORT je $\Theta(n \cdot \log_2(n))$ (!!)

Varianta QUICK-SORT

- Jedno rekurzivní volání v QUICK-SORT lze nahradit iterací (tail-recursion):

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

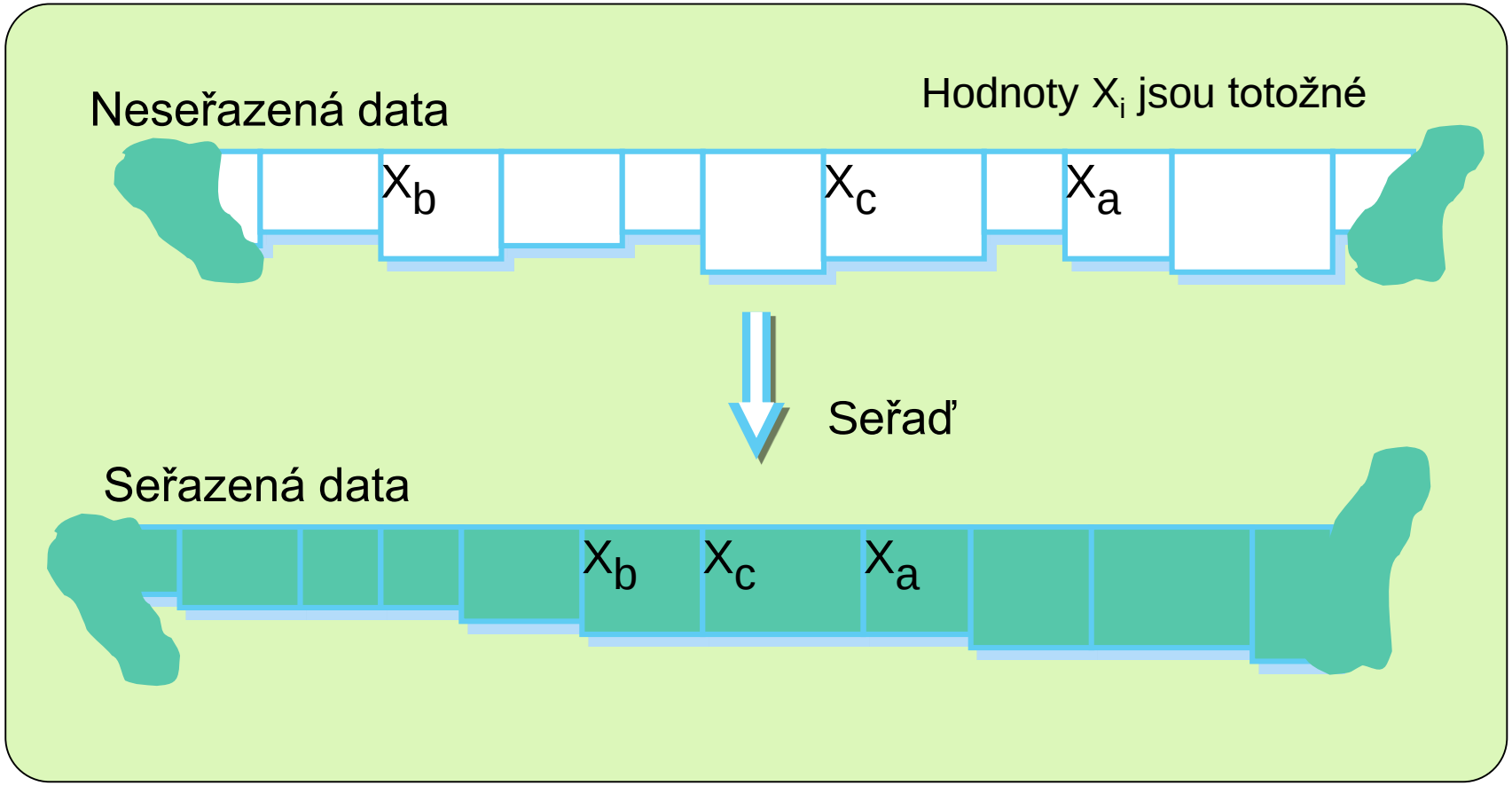
```
TAIL-RECURSIVE-QUICKSORT(A, p, r)
while p < r
    // Partition and sort left subarray.
    q = PARTITION(A, p, r)
    TAIL-RECURSIVE-QUICKSORT(A, p, q − 1)
    p = q + 1
```

Porovnání efektivity řazení

N	N^2	$N \times \log_2(N)$	$\frac{N^2}{N \times \log_2(N)}$	zpoma-lení (1~1sec)
1	**1**	**0**		
10	**100**	**33.2**	**3.0**	**3 sec**
100	**10 000**	**6 64.4**	**15.1**	**15 sec**
1 000	**1 000 000**	**9 965.8**	**100.3**	**1.5 min**
10 000	**100 000 000**	**132 877.1**	**752.6**	**13 min**
100 000	**10 000 000 000**	**1 660 964.0**	**6 020.6**	**1.5 hod**
1 000 000	**1 000 000 000 000**	**19 931 568.5**	**50 171.7**	**14 hod**
10 000 000	**100 000 000 000 000**	**232 534 966.6**	**430 042.9**	**5 dnů**

Stabilita řazení

Stabilní řazení nemění pořadí prvků se stejnou hodnotou.



Stabilní řazení

záznam:

Jméno	Příjmení

Vstup: seznam seřazen pouze podle jména

Andrew	Cook
Andrew	Amundsen
Andrew	Brown
Barbara	Cook
Barbara	Brown
Barbara	Amundsen
Charles	Amundsen
Charles	Cook
Charles	Brown

stabilní řazení

seřaď záznamy pouze podle příjmení

Výstup: seznam seřazen podle jména i příjmení

Andrew	Amundsen
Barbara	Amundsen
Charles	Amundsen
Andrew	Brown
Barbara	Brown
Charles	Brown
Andrew	Cook
Barbara	Cook
Charles	Cook

Pořadí záznamů se stejným příjmením se nezměnilo

Nestabilní řazení

záznam:

Jméno	Příjmení

Vstup: seznam seřazen pouze podle jména

Andrew	Cook
Andrew	Amundsen
Andrew	Brown
Barbara	Cook
Barbara	Brown
Barbara	Amundsen
Charles	Amundsen
Charles	Cook
Charles	Brown

QuickSort

seřaď záznamy pouze podle

příjmení

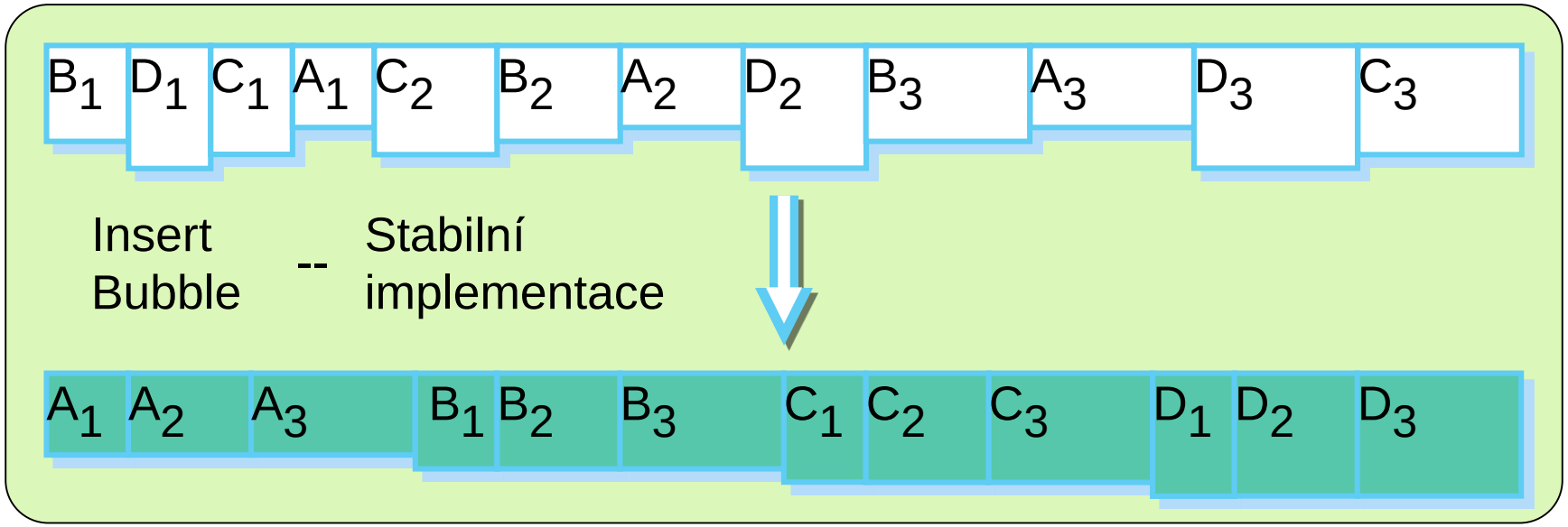
Výstup: původní pořadí jmen je ztraceno

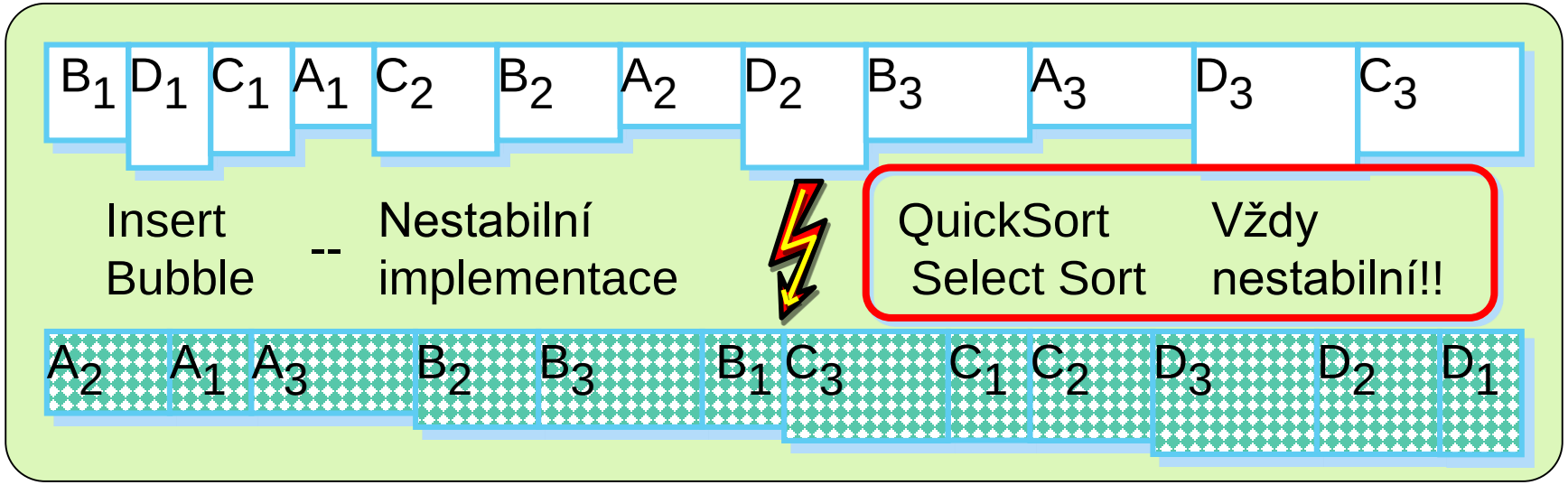
seřazeno

Barbara	Amundsen
Andrew	Amundsen
Charles	Amundsen
Barbara	Brown
Charles	Brown
Andrew	Brown
Charles	Cook
Andrew	Cook
Barbara	Cook

Pořadí záznamů se stejným příjmením se změnilo

Stabilita někdy závisí na implementaci





The End