

## ALG 09

**Radix sort (příhradkové řazení)**

**Counting sort**

**Přehled asymptotických rychlostí jednotlivých řazení**

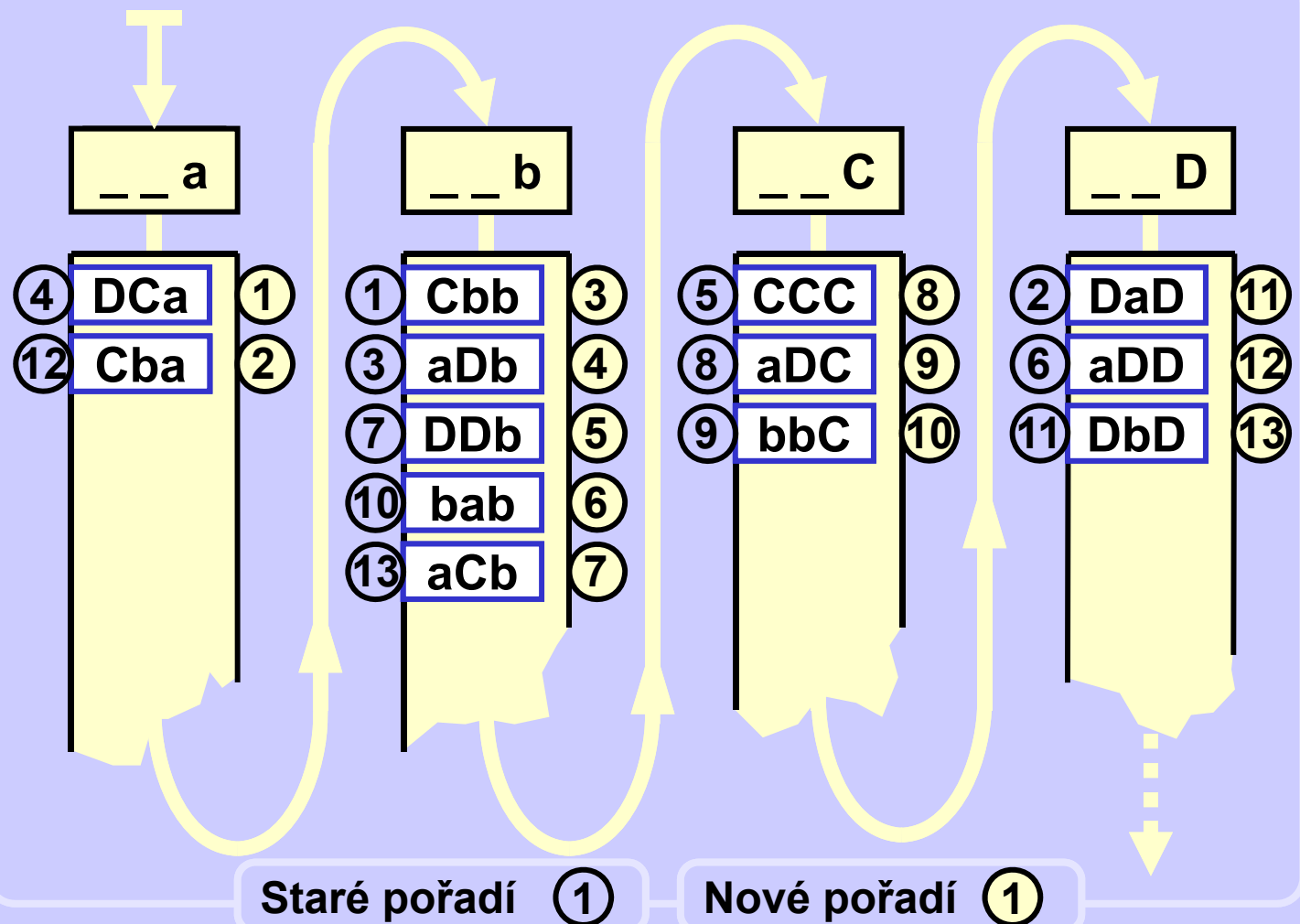
**Ilustrační experiment řazení**

## Radix sort

Neseřazeno

- ① Cbb
- ② DaD
- ③ aDb
- ④ DCa
- ⑤ CCC
- ⑥ aDD
- ⑦ DDb
- ⑧ aDC
- ⑨ bbC
- ⑩ bab
- ⑪ DbD
- ⑫ Cba
- ⑬ aCb

Řad' podle 3. znaku

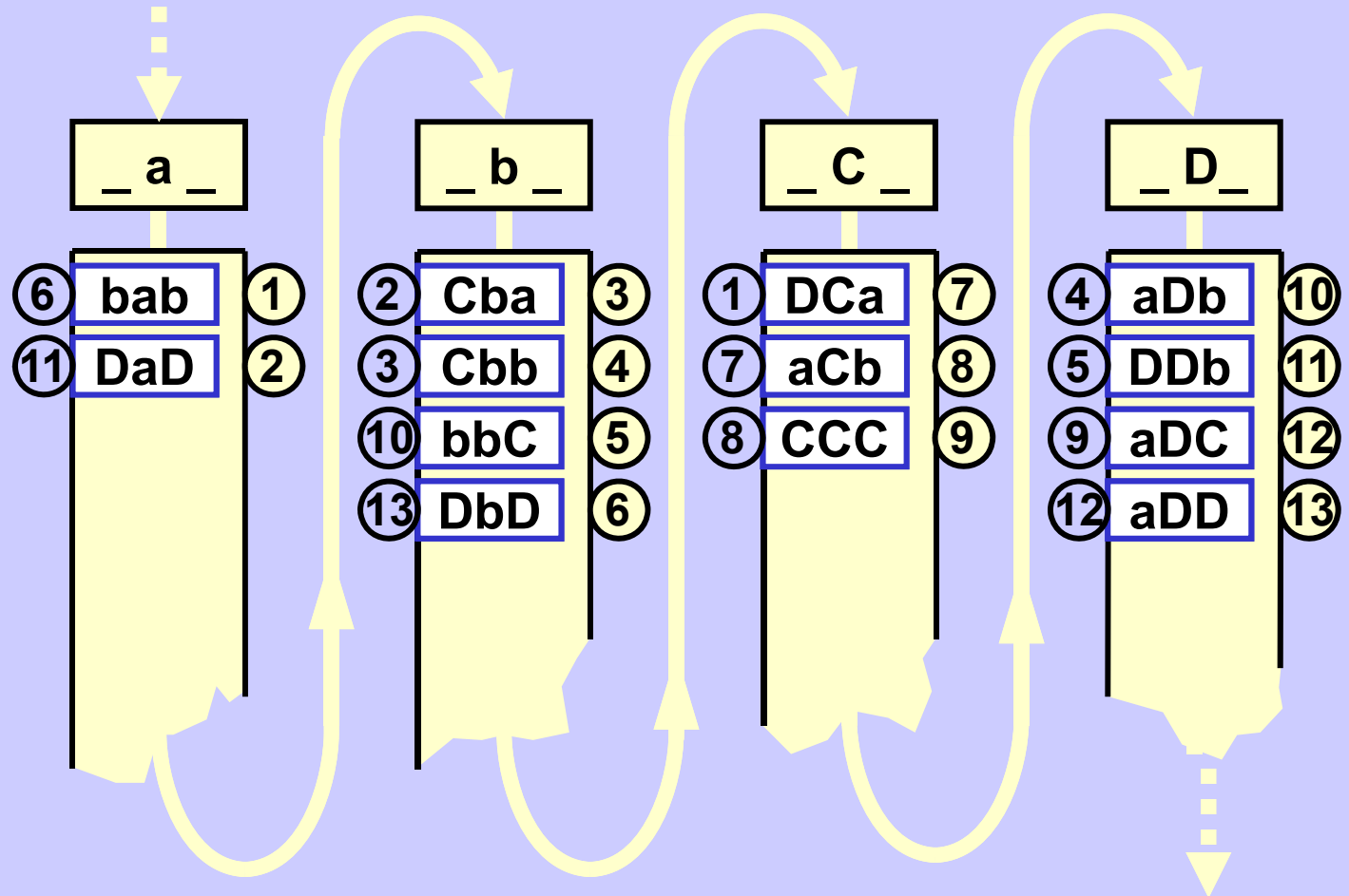


## Radix sort

Seřazeno  
od 3. znaku

- |   |     |
|---|-----|
| ① | DCa |
| ② | Cba |
| ③ | Cbb |
| ④ | aDb |
| ⑤ | DDb |
| ⑥ | bab |
| ⑦ | aCb |
| ⑧ | CCC |
| ⑨ | aDC |
| ⑩ | bbC |
| ⑪ | DaD |
| ⑫ | aDD |
| ⑬ | DbD |

Řad' podle 2. znaku

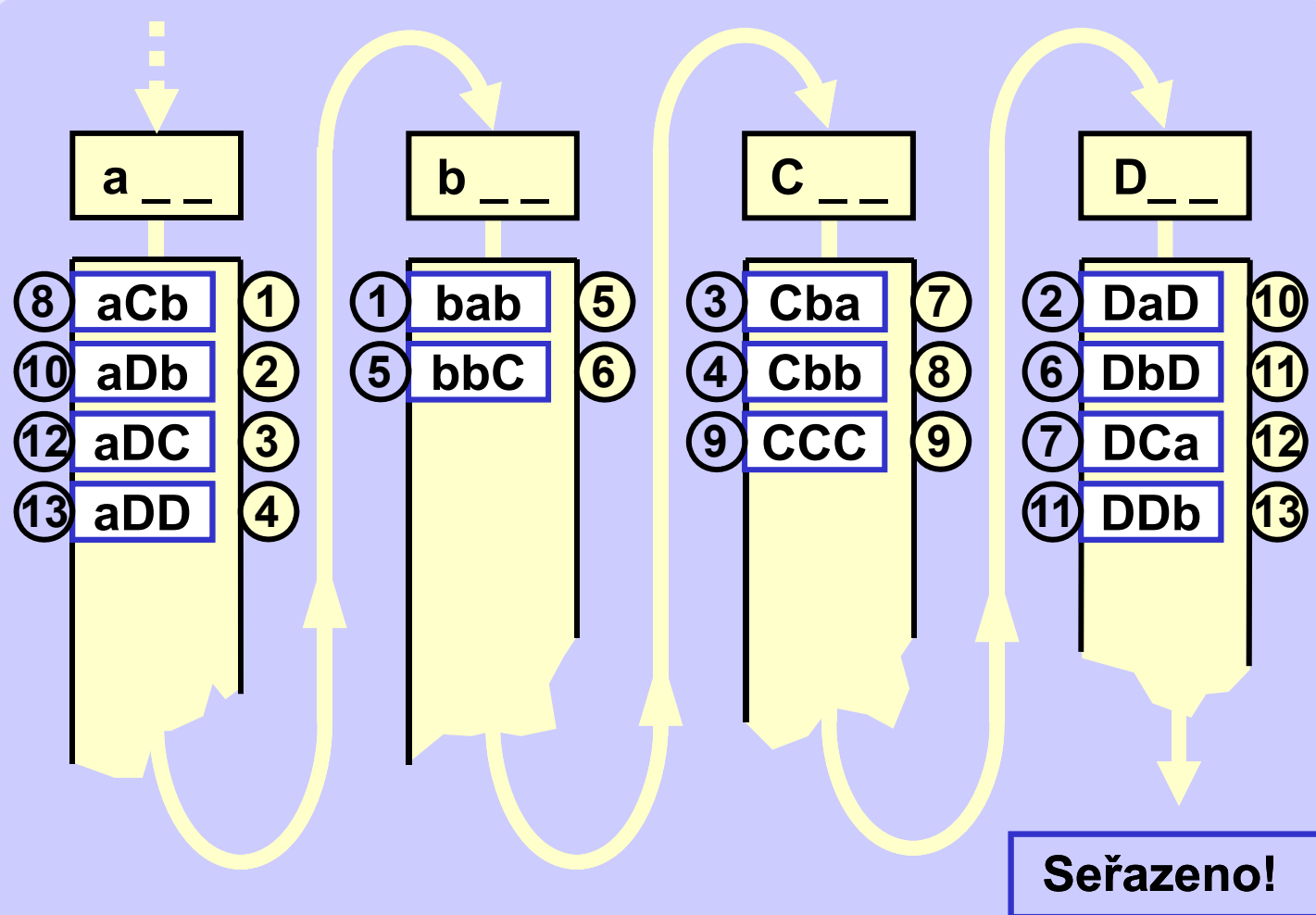


## Radix sort

Seřazeno  
od 2. znaku

①	bab
②	DaD
③	Cba
④	Cbb
⑤	bbC
⑥	DbD
⑦	DCa
⑧	aCb
⑨	CCC
⑩	aDb
⑪	DDb
⑫	aDC
⑬	aDD

Řad' podle 1. znaku

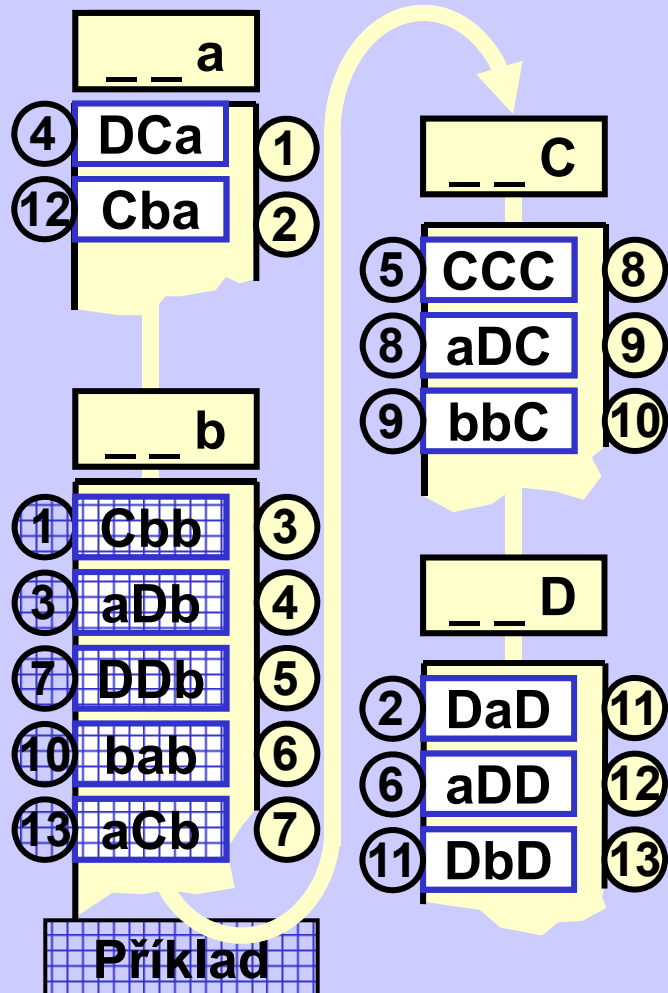


# Implementace radix sortu

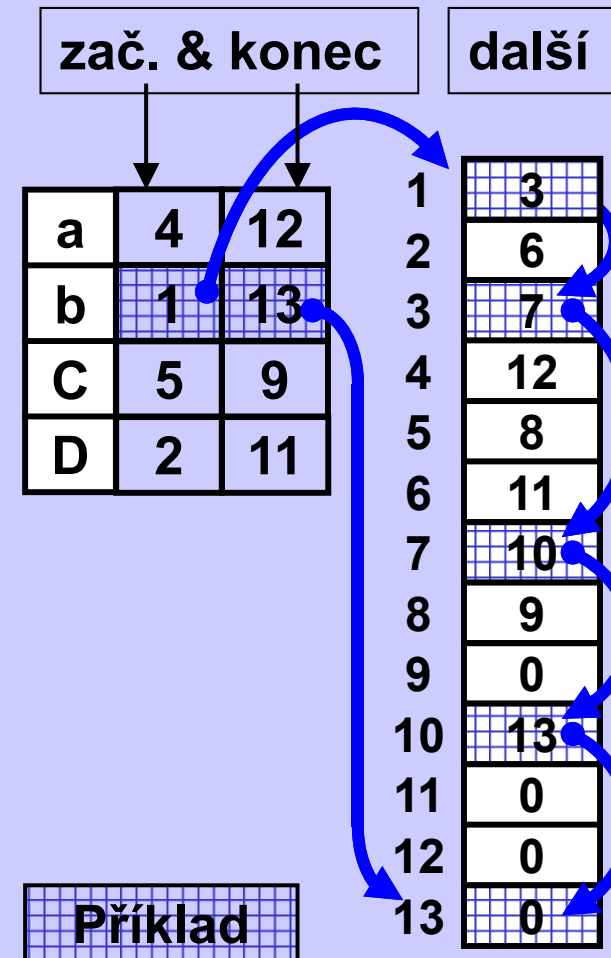
Neseřazeno

- ① Cbb
- ② DaD
- ③ aDb
- ④ DCa
- ⑤ CCC
- ⑥ aDD
- ⑦ DDb
- ⑧ aDC
- ⑨ bbC
- ⑩ bab
- ⑪ DbD
- ⑫ Cba
- ⑬ aCb

Seřazeno dle 3. znaku



Pomocná pole indexů registrují nové pořadí.



## Implementace radix sortu

Neseřazeno

1	Cbb
2	DaD
3	aDb
4	DCa
5	CCC
6	aDD
7	DDb
8	aDC
9	bbC
10	bab
11	DbD
12	Cba
13	aCb

Jedno pole pro všechny seznamy

3
6
7
12
8
11
10
9
0
13
0
0
0

	z	k
a	4	12
b	1	13
C	5	9
D	2	11

Ukázka seznamu pro 'b'



Pole ukazatelů na začátek a konec seznamu pro každý znak

Aktuálně obě pole přesně registrují stav po seřazení podle 3. znaku.



Radix sort lze provést bez přesouvání původních dat, pouze manipulací s uvedenými celočíselnými poli, která obsahují veškerou informaci o aktuálním stavu řazení.

## Implementace radix sortu

Neseřazeno

①	Cbb
②	DaD
③	aDb
④	DCa
⑤	CCC
⑥	aDD
⑦	DDb
⑧	aDC
⑨	bbC
⑩	bab
⑪	DbD
⑫	Cba
⑬	aCb

Stav po seřazení  
podle 2. znaku.

9
0
7
13
0
0
8
6
11
2
0
1
5

	z	k
a	10	2
b	12	11
C	4	5
D	3	6

Stav po seřazení  
podle 1. znaku = seřazeno.

5
11
8
7
0
0
0
6
0
9
4
1
3

	z	k
a	13	6
b	10	9
C	12	5
D	2	7

Ukázka  
seznamů  
pro 'b'



# Implementace radix sortu

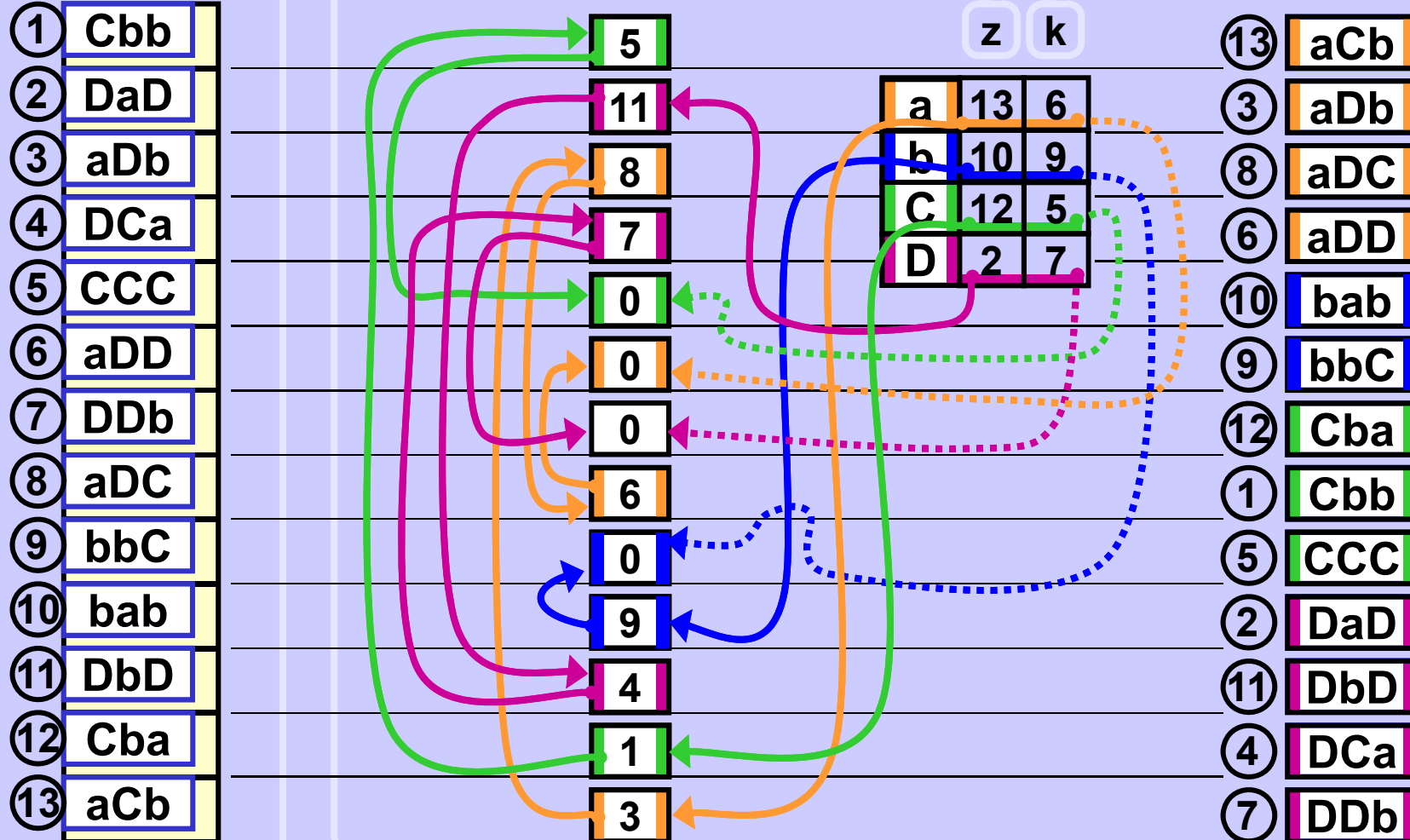
Neseřazeno

- ① Cbb
- ② DaD
- ③ aDb
- ④ DCa
- ⑤ CCC
- ⑥ aDD
- ⑦ DDb
- ⑧ aDC
- ⑨ bbC
- ⑩ bab
- ⑪ DbD
- ⑫ Cba
- ⑬ aCb

Stav po seřazení podle  
1. znaku = seřazeno.

Stačí vypsát data  
v pořadí daném seznamy:

a → b → C → D →





## Od seřazení podle 2. znaku k seřazení podle 1. znaku

Pole obsahují uspořádání podle 2. znaku.

	z	k
a	10	2
b	12	11
C	4	5
D	3	6

1	Cbb	9	4
2	DaD	0	2
3	aDb	7	10
4	DCa	13	7
5	CCC	0	9
6	aDD	0	13
7	DDb	8	11
8	aDC	6	12
9	bbC	11	5
10	bab	2	1
11	DbD	0	6
12	Cba	1	3
13	aCb	5	8

d

	z1	k1
a	0	0
b	0	0
C	0	0
D	0	0

Pole budou obsahovat uspořádání podle 1. znaku

0
0
0
0
0
0
0
0
0
0
0
0
0
0
0

d1

Aktualizace polí z, k, d proběhne tak, že naplníme nová pole z1, k1, d1, která nakonec zkopírujeme zpět do z, k, d.

Implementačně ovšem není třeba cokoli kopírovat, stačí záměna referencí (ukazatelů, pointerů) na tato pole.

# Od seřazení podle 2. znaku k seřazení podle 1. znaku

Usp. dle 2. zn.

	z	k
a	10	2
b	12	11
C	4	5
D	3	6

1	Cbb	9	4
2	DaD	0	2
3	aDb	7	10
4	DCa	13	7
5	CCC	0	9
6	aDD	0	13
7	DDb	8	11
8	aDC	6	12
9	bbC	11	5
10	bab	2	1
11	DbD	0	6
12	Cba	1	3
13	aCb	5	8

d

	z1	k1
a	0	0
b	0	0
C	0	0
D	0	0

d1

	z1	k1
a	0	0
b	10	10
C	0	0
D	0	0

d1

	z1	k1
a	0	0
b	10	10
C	0	0
D	2	2

d1

	z1	k1
a	0	0
b	10	10
C	12	12
D	2	2

d1

	z1	k1
a	0	0
b	10	10
C	12	1
D	2	2

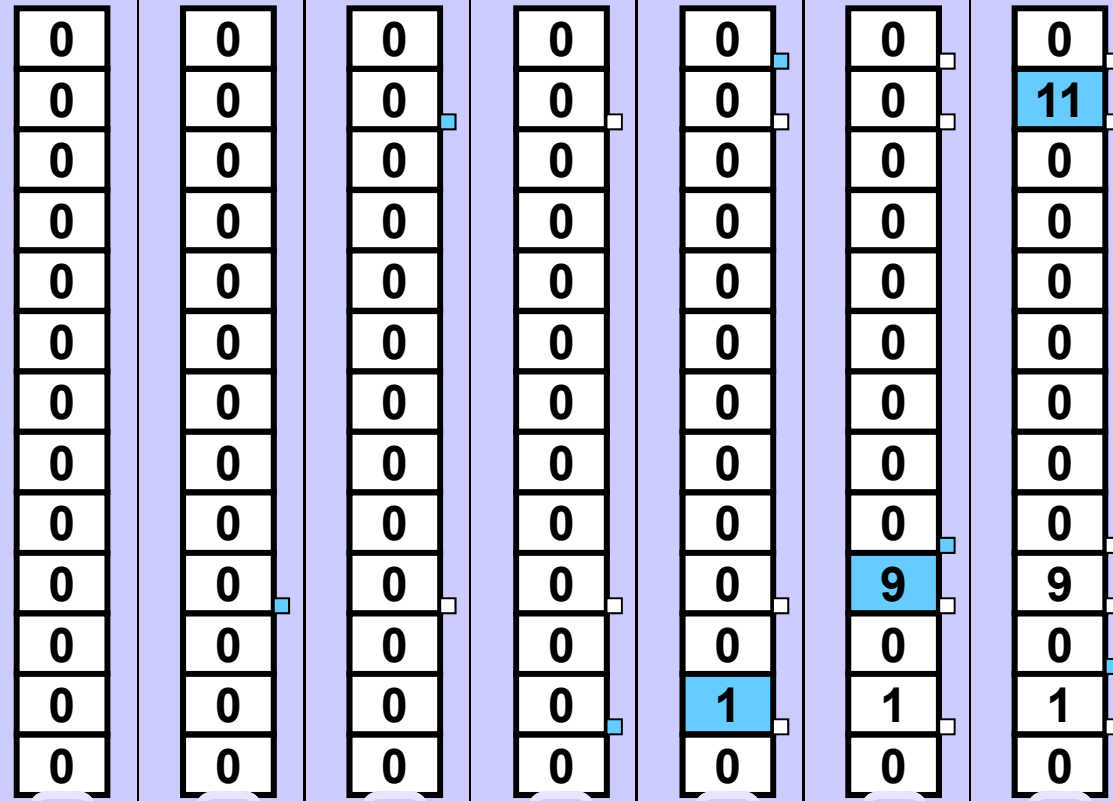
d1

	z1	k1
a	0	0
b	10	9
C	12	1
D	2	2

d1

	z1	k1
a	0	0
b	10	9
C	12	1
D	2	11

d1



## Od seřazení podle 2. znaku k seřazení podle 1. znaku

Usp.  
dle  
2. zn.

	z	k
a	10	2
b	12	11
C	4	5
D	3	6

①	Cbb	9	④
②	DaD	0	②
③	aDb	7	⑩
④	DCa	13	⑦
⑤	CCC	0	⑨
⑥	aDD	0	⑬
⑦	DDb	8	⑪
⑧	aDC	6	⑫
⑨	bbC	11	⑤
⑩	bab	2	①
⑪	DbD	0	⑥
⑫	Cba	1	③
⑬	aCb	5	⑧

d

	z1	k1
a	0	0
b	10	9
C	12	1
D	2	4

0	0	5	5	5	5	5
11	11	11	11	11	11	11
0	0	0	0	0	8	8
0	0	0	0	7	7	7
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
9	9	9	9	9	9	9
4	4	4	4	4	4	4
1	1	1	1	1	1	1
0	0	0	3	3	3	3

d1

d1

d1

d1

d1

d1

d1

Hotovo

## Implementace radix sortu

```

void radix_sort( String [] a ) {
    int charCount = ...;    // number of chars used (2^16?)
    int [] z = new int [charCount];
    int [] k = new int [charCount];
    int [] z1 = new int [charCount];
    int [] k1 = new int [charCount];
    int [] d = new int [a.length];
    int [] d1 = new int [a.length];
    int [] aux;

    initStep( a, z, k, d );    // 1st pass with last char

    for( int p = a[0].length()-2; p >= 0; p-- ) {
        radixStep( a, p, z, k, d, z1, k1, d1 ); // do the job
        aux = z; z = z1; z1 = aux;           // just swap arrays
        aux = k; k = k1; k1 = aux;           // dtto
        aux = d; d = d1; d1 = aux;           // dtto
    }
    output( a, z, k, d );    // print sorted array
}

```

## Implementace radix sortu

```

void initStep( String[] a, int [] z, int [] k, int [] d ){
    int pos = a[0].length()-1;           // last char in string
    int c;                               // char as array index for Radix sort
    for( int i = 0; i < z.length; i++ ) // init arrays
        z[i] = k[i] = -1;                // empty
    for( int i = 0; i < a.length; i++ ){ // all last chars
        c = (int) a[i].charAt(pos);     // char to index
        if( z[c] == -1 )
            k[c]= z[c] = i;              // start new list
        else {
            d[k[c]] = i;                 // extend existing list
            k[c] = i;
        }
    } } }

```

Řetězce různé délky nutno uvést na stejnou délku  
připojením "nevýznamných znaků", např. mezer.

V ukázkovém kódu všechny indexy polí začínají 0,  
v obrázcích začínají 1.

## Implementace radix sortu

```

void radixStep( String [] a, int pos, int [] z, int [] k,
                int [] d, int [] z1, int [] k1, int [] d1 ){
    int j;           // index traverses old lists
    int c;           // char as array index for Radix sort
    for( int i = 0; i < z.length; i++ ) // init arrays
        z1[i] = k1[i] = -1;           //
    for( int i = 0; i < z.length; i++ ) // for all used chars
        if (z[i] != -1) {           // unempty list
            j = z[i];
            while( true ){           // scan the list
                c = (int) a[j].charAt(pos); // char to index
                if( z1[c] == -1 )
                    k1[c]= z1[c] = j; // start new list
                else {
                    d1[k1[c]] = j; // extend existing list
                    k1[c] = j;
                }
                if( j == k[i] ) break;
                j = d[j];           // next string index
            }
        }
    } }

```

## Radix sort

## Shrnutí

d znaků ..... d cyklů

cyklus .....  $\Theta(n)$  operací

---

celkem .....  $\Theta(d \cdot n)$  operací

$d \ll n \Rightarrow$  .....  $\Theta(n)$  operací

---

Radix sort nemění pořadí stejných hodnot

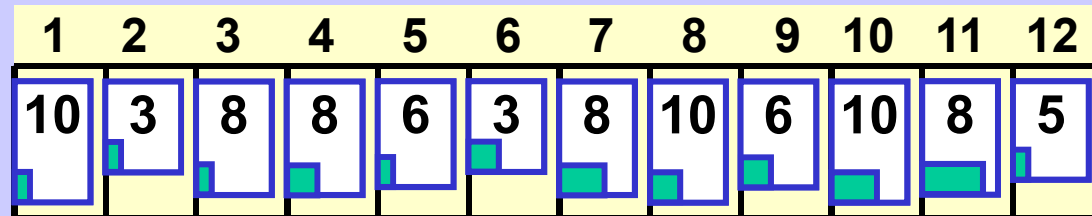
Asymptotická složitost Radix sortu je  $\Theta(n \cdot d)$

Pro malé konstantní d lze psát  $\Theta(n)$

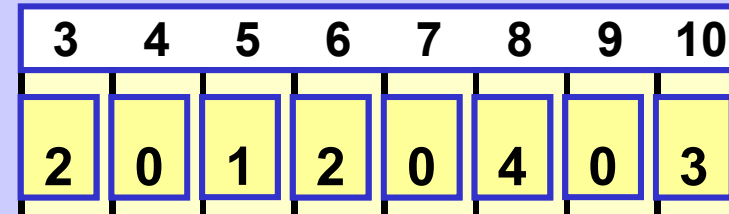
Je to stabilní řazení

## Counting sort

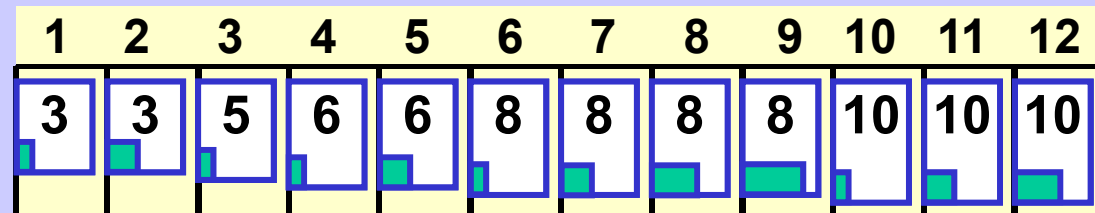
**vstup**  
vstup.length == N



**cetnost**  
cetnost.length == k  
 $k = \max(\text{vstup}) - \min(\text{vstup}) + 1$



**vystup**  
vstup.length == N





# Counting sort

Krok 1

Vynulování pole četností

3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0

Jeden průchod vstupním polem

----->

10	3	8	8	6	3	8	10	6	10	8	5
----	---	---	---	---	---	---	----	---	----	---	---

Naplnění pole četností

3	4	5	6	7	8	9	10
2	0	1	2	0	4	0	3

# Counting sort

## Krok 2

jeden průchod →

3	4	5	6	7	8	9	10
2	0	1	2	0	4	0	3

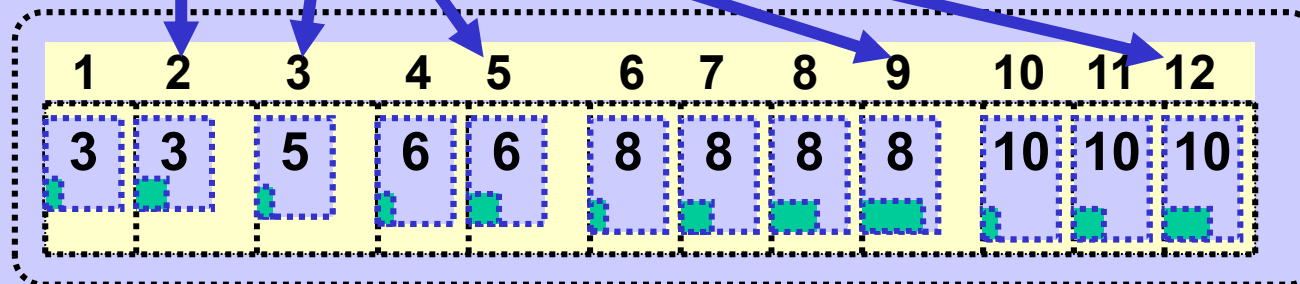
Pole četností  
mění svou roli

Úprava pole četností

```
for( int i = dMez+1; i <= hMez; i++ )
    cetnost[i] += cetnost[i-1]
```

3	4	5	6	7	8	9	10
2	2	3	5	5	9	9	12

Prvek **cetnost[j]** obsahuje pozici  
posledního prvku s hodnotou **j**  
v budoucím výstupním poli.

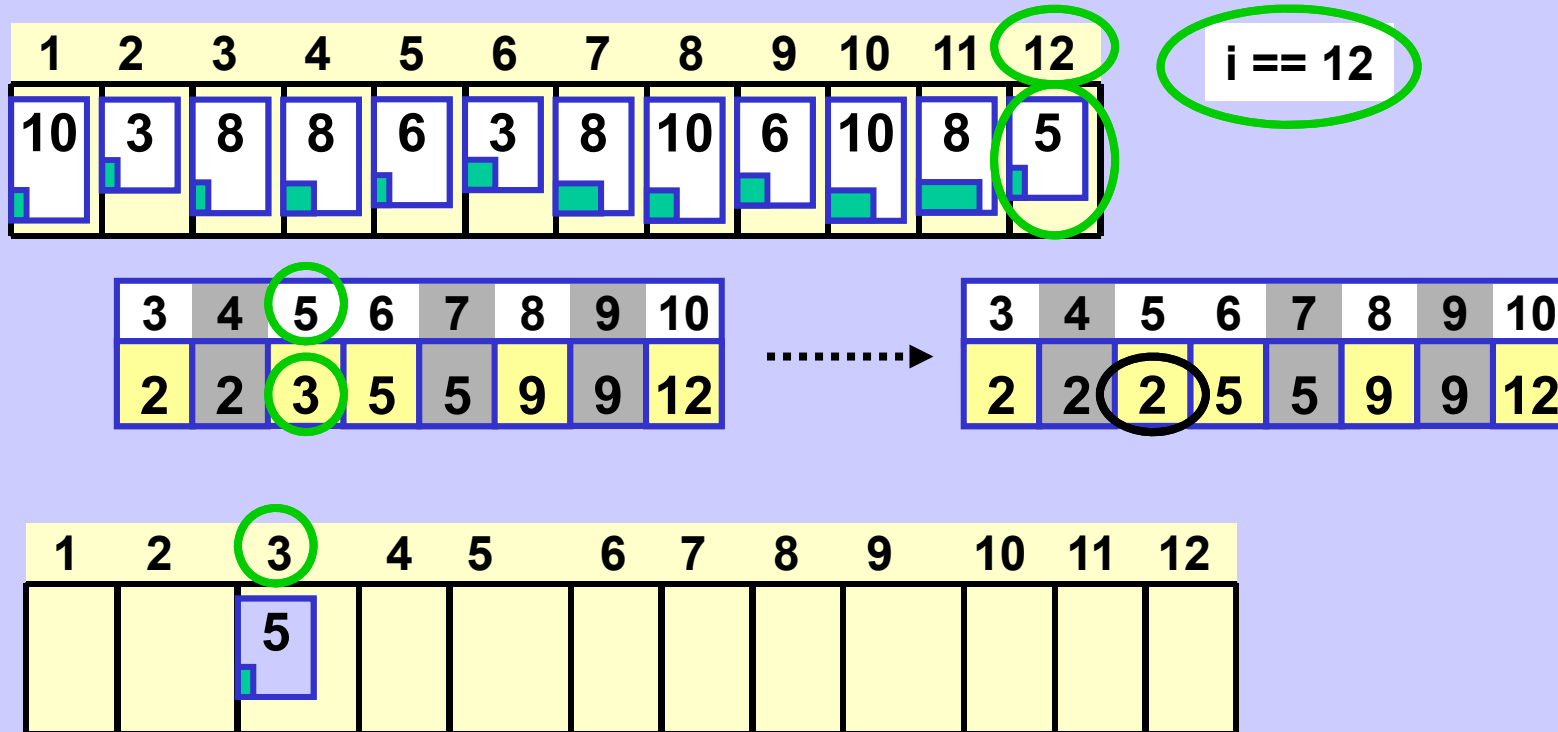


# Counting sort

## Krok 3

$i == N$

```
for( int i = N; i > 0; i-- ) {
    vystup[cetnost[vstup[i]]] = vstup[i];
    cetnost[vstup[i]]--;
}
```

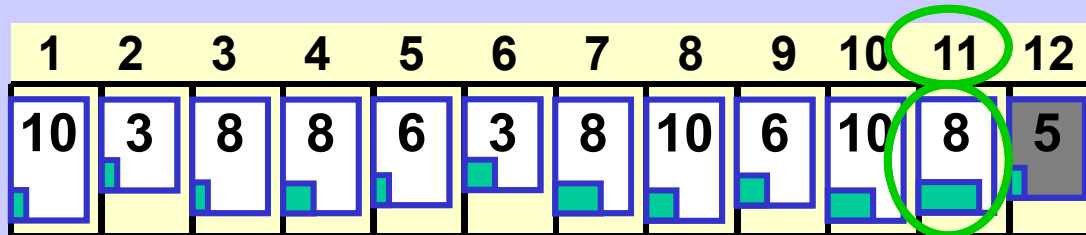


# Counting sort

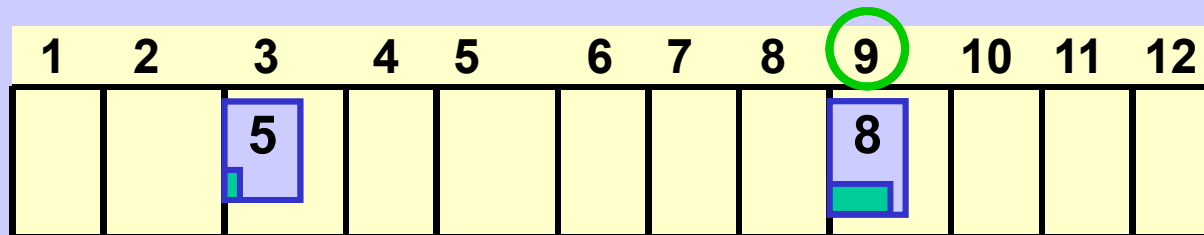
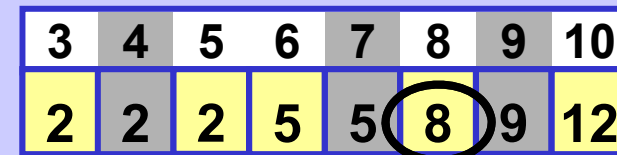
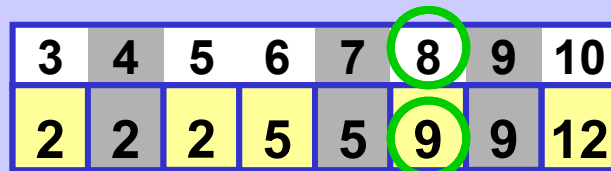
## Krok 3

$i == N-1$

```
for( int i = N; i > 0; i-- ) {
    vystup[cetnost[vstup[i]]] = vstup[i];
    cetnost[vstup[i]]--;
}
```



$i == 11$

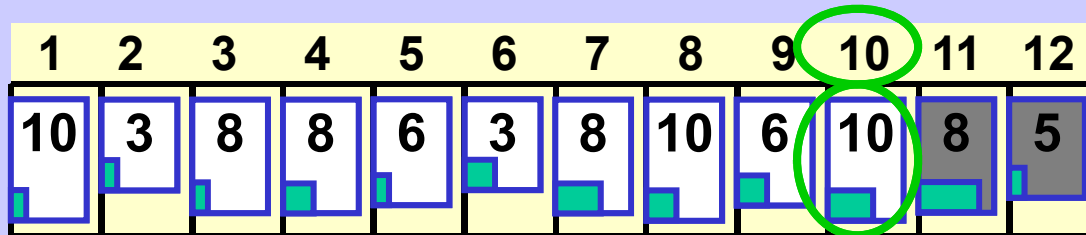


# Counting sort

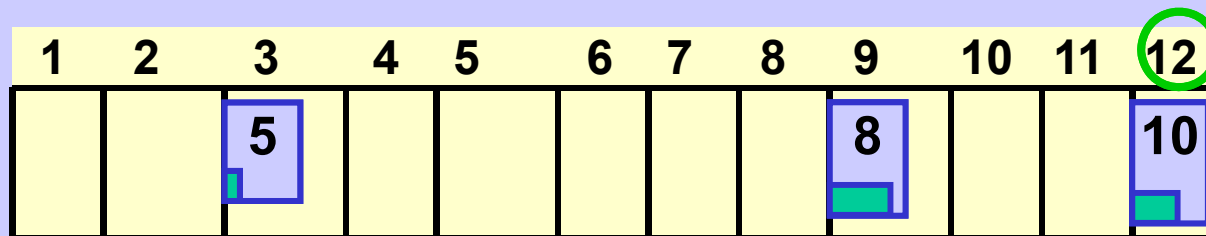
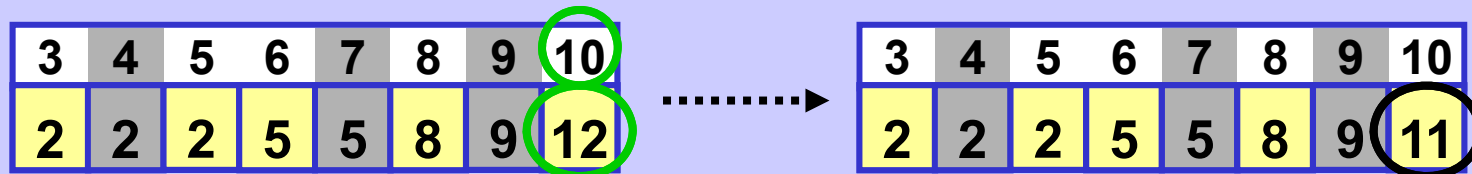
## Krok 3

$i == N-2$

```
for( int i = N; i > 0; i-- ) {
    vystup[cetnost[vstup[i]]] = vstup[i];
    cetnost[vstup[i]]--;
}
```



$i == 10$



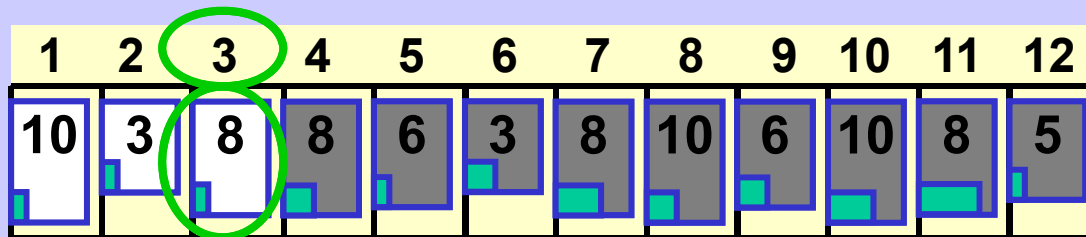
atd...

# Counting sort

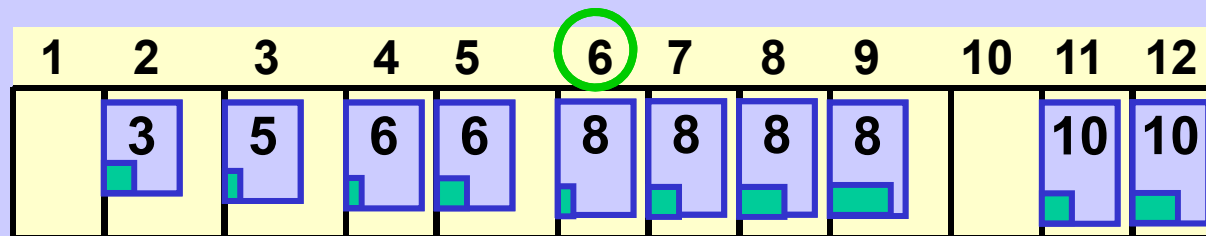
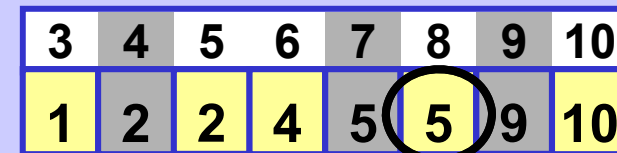
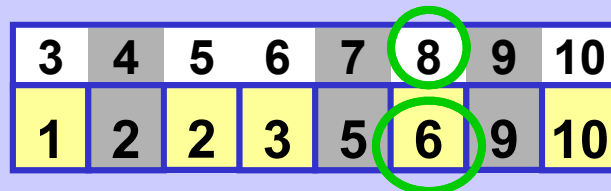
## Krok 3

`i == 3`

```
for( int i = N; i > 0; i-- ) {
    vystup[cetnost[vstup[i]]] = vstup[i];
    cetnost[vstup[i]]--;
}
```



`i == 3`



atd...

## Orientační přehled vlastností řadících algoritmů

Velikost pole $n$	Nejhorší případ	Nejlepší případ	"typický", "běžný" případ	Stabilní	Všeobecné
Selection sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	Ne	Ano
Insertion sort	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$	Ano	Ano
Bubble sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	Ano	Ano
Quick sort	$\Theta(n^2)$	$\Theta(n \cdot \log(n))$	$\Theta(n \cdot \log(n))$	Ne	Ano
Merge sort	$\Theta(n \cdot \log(n))$	$\Theta(n \cdot \log(n))$	$\Theta(n \cdot \log(n))$	Ano	Ano
Heap sort	$\Theta(n \cdot \log(n))$	$O(n \cdot \log(n))$	$\Theta(n \cdot \log(n))$	Ne	Ano
Radix sort	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	Ano	Ne
Counting sort	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	Ano	Ne

## **Ilustrační experiment řazení**

### **Prostředí**

**Intel(R) 1.8 GHz, Microsoft Windows XP SP3, jdk 1.6.0\_16.**

### **Organizace**

**Použita řazení, v nichž se prvky (double) navzájem porovnávají.  
Kódy převzaty z přednášky (žádné další triky).  
Řazení v jednotlivých velikostech prováděna na stejných datech.  
Pole náhodně zamíchána generátorem pseudonáhodných čísel  
s rovnoměrným rozložením.  
Výsledky průměrovány přes větší počet běhů.**

### **Závěr**

**Neexistuje jedno univerzální řazení, které by bylo optimální  
za všech okolností.  
Hraje roli stabilita, velikost dat i stupeň předběžného seřazení dat.**



## Ilustrační experiment řazení

Délka pole	% seř.	Doba běhu v milisekundách, není-li uvedeno jinak					
		Sort					
		Select	Insert	Bubble	Quick	Merge	Heap
10	0%	0.0005	★ 0.0002	0.0005	0.0004	0.0009	0.0005
10	90%	0.0004	★ 0.0001	0.0004	0.0004	0.0007	0.0005
100	0%	0.028	0.016	0.043	0.081	0.014	★ 0.011
100	90%	0.026	★ 0.003	0.030	0.010	0.011	0.011
1 000	0%	2.36	1.30	4.45	★ 0.12	0.19	0.17
1 000	90%	2.31	0.18	2.86	0.16	★ 0.15	0.16
10 000	0%	228	130	450	★ 1.57	2.40	2.31
10 000	90%	229	17.5	285	1.93	★ 1.68	2.11
100 000	0%	22 900	12 800	45 000	★ 18.7	31.4	31.4
100 000	90%	22 900	1 760	28 500	27.4	★ 24.6	25.5
1 000 000	0%	38 min	22 min	75 min	★ 237	385	570
1 000 000	90%	38 min	2.9 min	47.5 min	336	★ 301	381

Seřazená část pole. Pole je nejprve seřazeno, pak náhodně vybrané prvky změň libovolně hodnotu.

## Ilustrační experiment řazení

Délka pole	% seř.	Koeficient zpomalení ( >1 ) vůči Quick sortu pro danou velikost dat a předběžné seřazení							
		Sort							
		Select	Insert	Bubble	Quick	Merge	Heap		
10	0%	1.3	★ 0.7	1.4	1	✗ 2.5	1.4		
10	90%	1	★ 0.26	0.96	1	✗ 1.8	1.3		
100	0%	3.4	✗ 1.8	5.4	★ 1	1.75	1.35		
100	90%	2.46	★ 0.28	2.9	1	✗ 1.07	1.07		
1 000	0%	20	✗ 11	37.5	★ 1	1.65	1.4		
1 000	90%	15	✗ 1.2	18.5	1	★ 0.95	1.03		
10 000	0%	146	✗ 83	287	★ 1	1.53	1.48		
10 000	90%	118	✗ 9.1	148	1	★ 0.87	1.09		
100 000	0%	1 220	✗ 686	2 410	★ 1	1.7	1.7		
100 000	90%	837	✗ 64.1	1 040	1	★ 0.9	0.93		
1 000 000	0%	9 960	✗ 5 400	19 000	★ 1	1.6	2.41		
1 000 000	90%	6 820	521	8 480	1	★ 0.9	1.14		

Nejrychlejší ★

Nejpomalejší ✗

Stabilní □

Select a Bubble sorty nesoutěží.

## Ilustrační experiment řazení

Délka pole	% seř.	Koefficient zpomalení ( $> 1$ ) při srovnání rychlosti řazení neseřazeného a částečně seřazeného pole					
		Sort <input type="checkbox"/>					
		Select	Insert	Bubble	Quick	Merge	Heap
10	0%	1	1	1	1	1	1
10	90%	0.8	0.5	0.8	1	0.8	1
100	0%	1	1	1	1	1	1
100	90%	0.9	0.2	0.68	1.27	0.78	1
1 000	0%	1	1	1	1	1	1
1 000	90%	0.98	0.14	0.64	1.31	0.75	0.95
10 000	0%	1	1	1	1	1	1
10 000	90%	1.0	0.14	0.63	1.23	0.7	0.91
100 000	0%	1	1	1	1	1	1
100 000	90%	1.0	0.14	0.63	1.46	0.78	0.81
1 000 000	0%	1	1	1	1	1	1
1 000 000	90%	1.0	0.14	0.63	1.42	0.78	0.67

Stabilní