Estruturas de Dados e Algoritmos

Algoritmos de Ordenação em Tempo Linear

Adalberto Cajueiro

Departamento de Sistemas e Computação

Universidade Federal de Campina Grande

Comparação dos algoritmos de ordenação por comparação

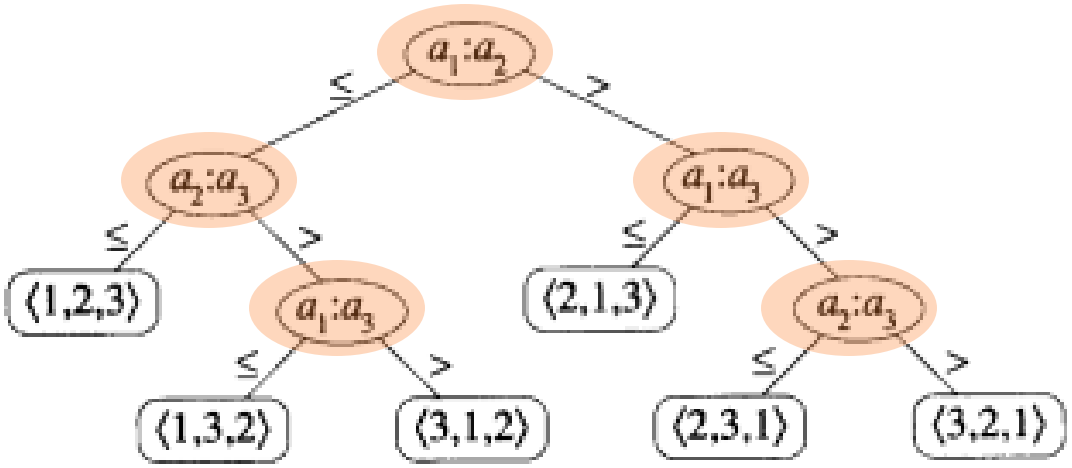
Algoritmo	Pior Caso	Caso médio
Bubble Sort	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n^2)$	$O(n^2)$
Merge Sort	O(nlog n)	O(nlog n)
Quick Sort	$O(n^2)$	O(nlog n)

Ordenação por comparação

- Diversos algoritmos de ordenação por comparação tem tempo de execucao Θ(n.log n)
- Ordenação por comparacao: *a ordem que eles determinam é baseada na comparação dos elementos da entrada*.
- Qual o limite de uma ordenação por comparação?
- Existe algoritmo mais rápido (em tempo linear) que os de ordenação por comparação?

Árvore de Decisão

- Representa as possíveis permutacoes da entrada
- Comparacoes realizadas: $a_i \leq a_j$
- Exemplo:



- A execucao de um algoritmo de ordenacao é um caminho da raiz até alguma folha

Limite para o pior caso

- O caminho mais longo da raiz a uma folha representa o pior caso (altura da árvore de decisão).
- Teorema: uma árvore de decisao que ordena n elementos tem altura Ω(n.log n).

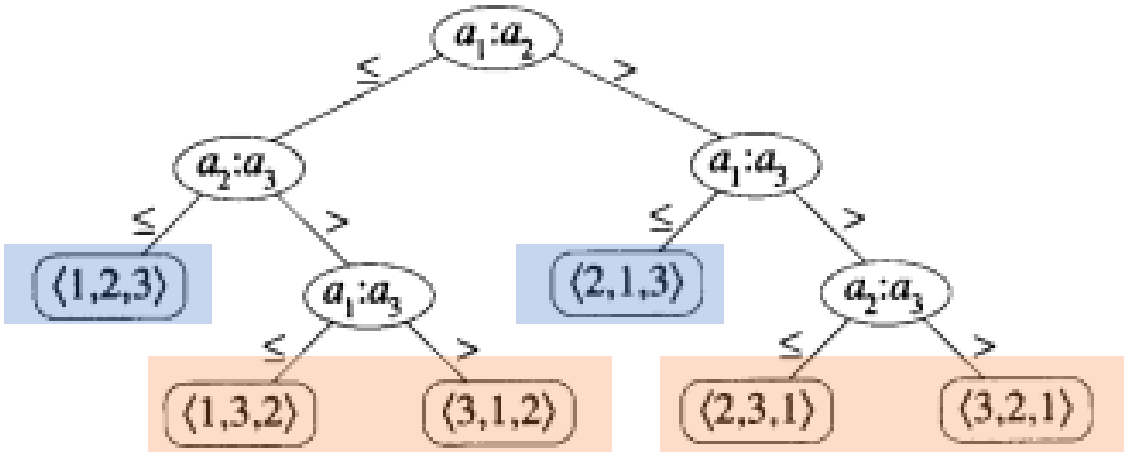
$$\#\ folhas \geq n!$$

$$\#\ folhas = 2^h$$

$$2^h \geq n!$$

$$h \geq \log n!$$

$$\begin{aligned} h &\geq \log(n/e)^n \\ &\geq n\log(n/e) \\ &\geq \frac{n}{e}\log n \\ &= \Omega(n\log n) \end{aligned}$$



Counting Sort

Counting Sort

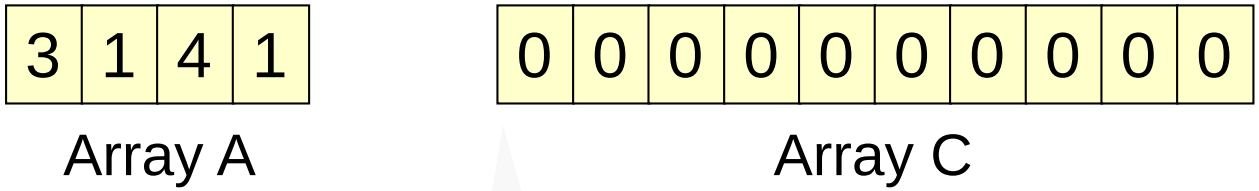
- Não utiliza comparações
- Toma vantagem sabendo do intervalo de valores 1..k a serem ordenados
- Idéia básica
  - Determinar, para cada elemento *x* da entrada, o número de elementos menores que *x*.
  - Essa informacao é usada para colocar o *x* na posicao correta do array de resposta.
- Requer dois arrays extras:
  - B[1..n] para a resposta
  - C[1..k] para armazenamento temporario

Counting Sort

- Idéia
  1. Criamos um array C de k elementos inicializados com 0
  2. Percorremos cada um dos n do array A números e adicionamos C[n-1]++;
  3. Soma dos prefixos C[i + 1] = C[i+1] + C[i]
  4. Depois ordenamos o arra y A de acordo com C

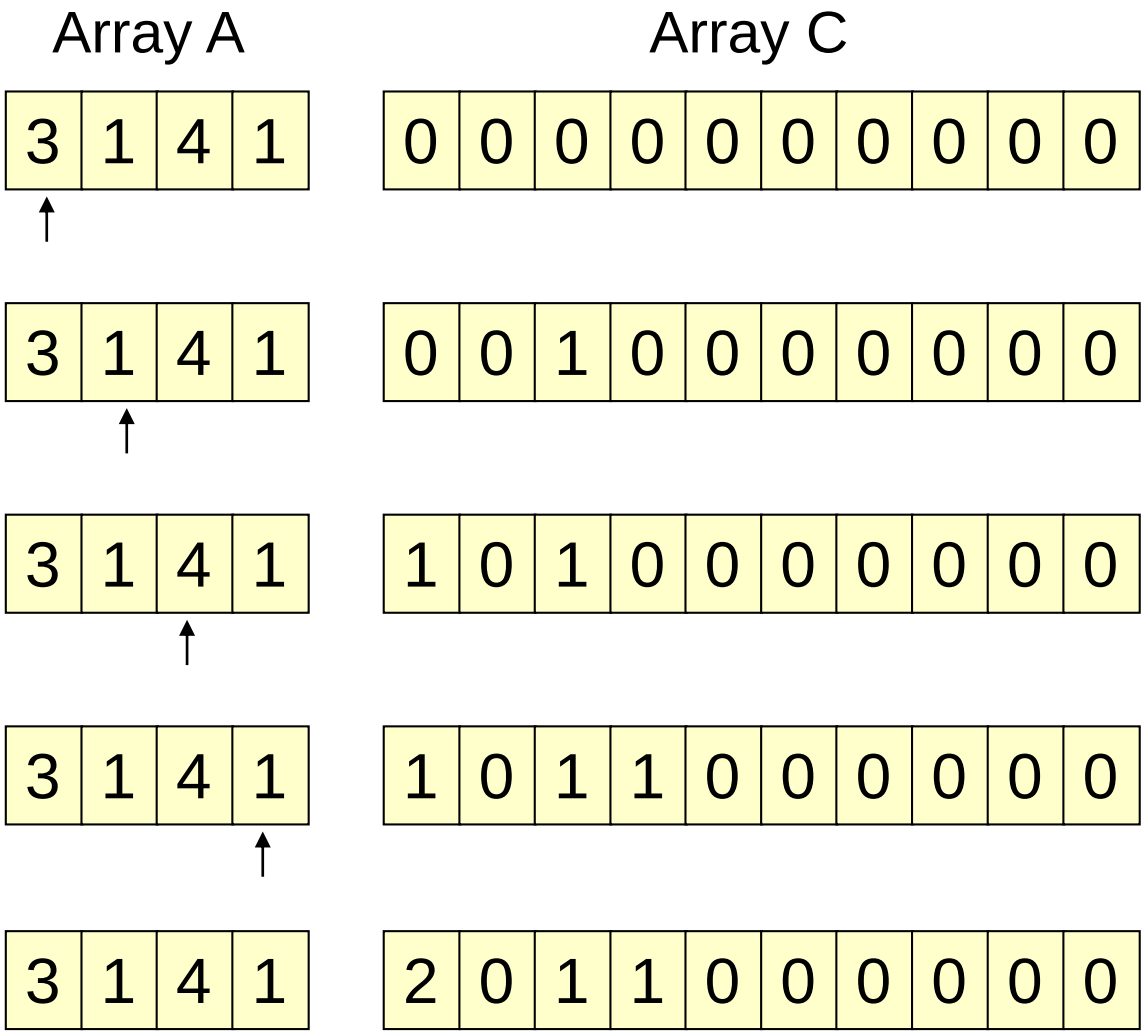
Counting Sort

- Exemplo
  - Suponha um exemplo de um domínio de 1-10

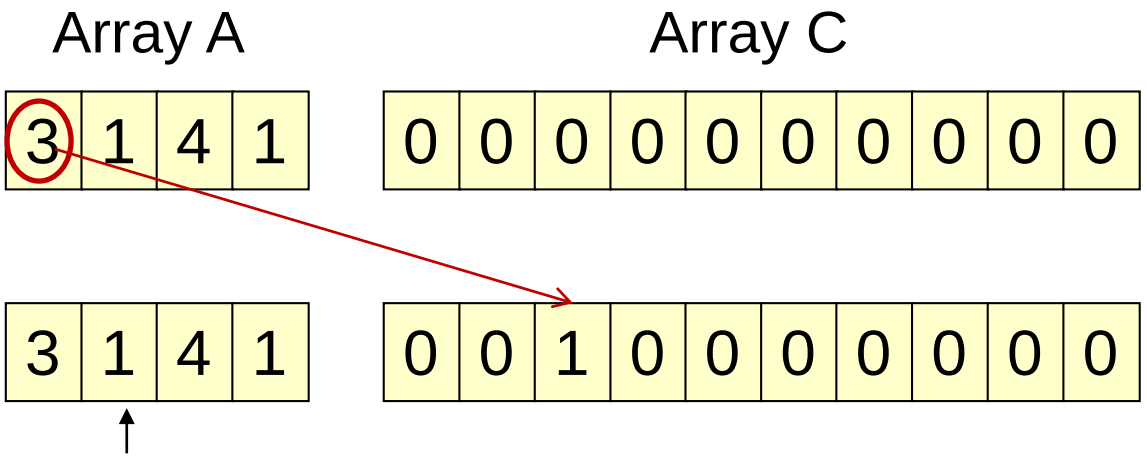


Os índices do array C começam em 1

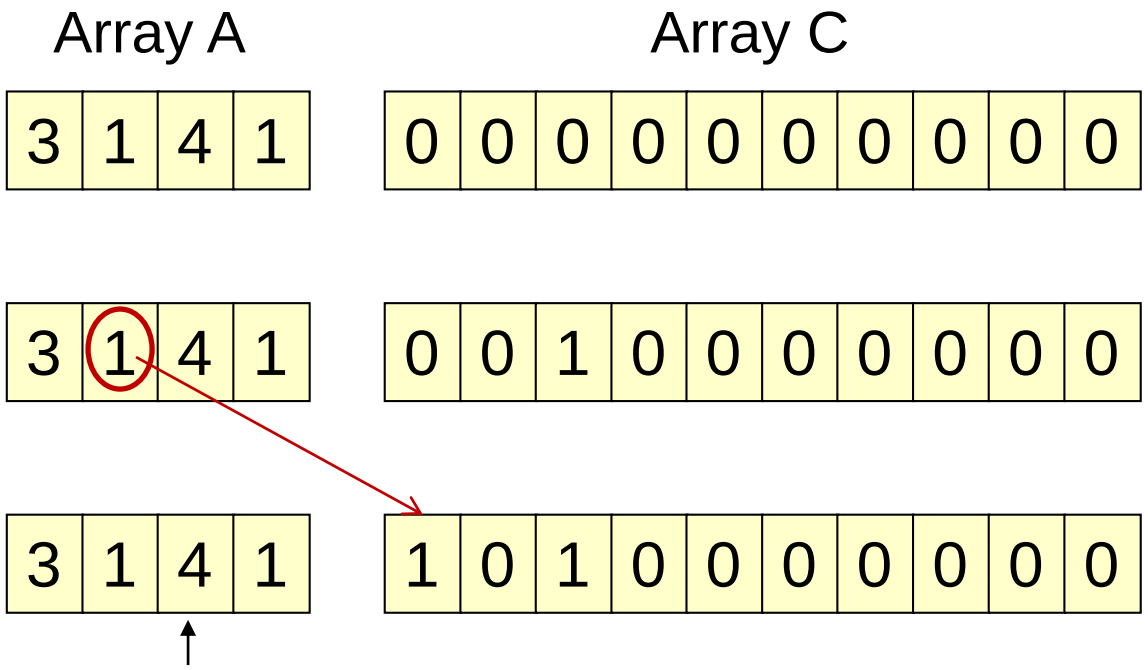
Counting Sort (contagem dos elementos)



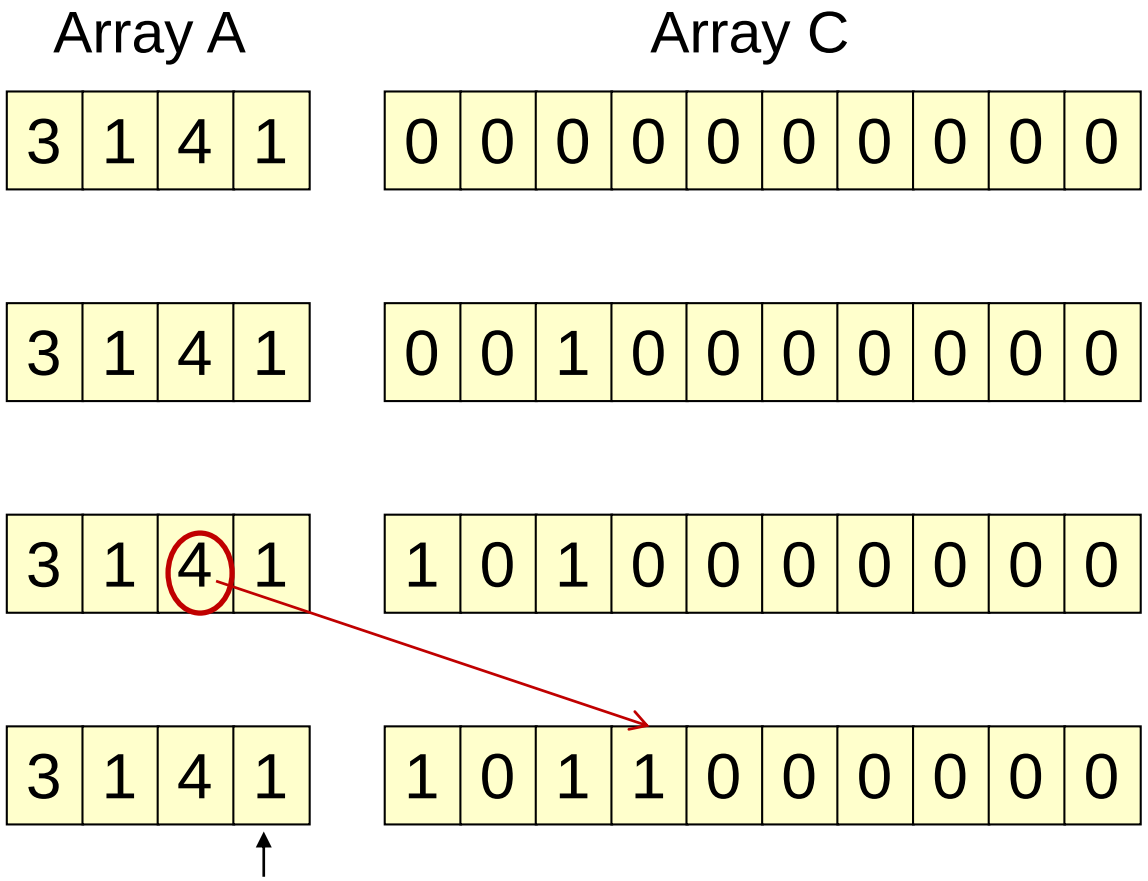
Counting Sort (contagem dos elementos)



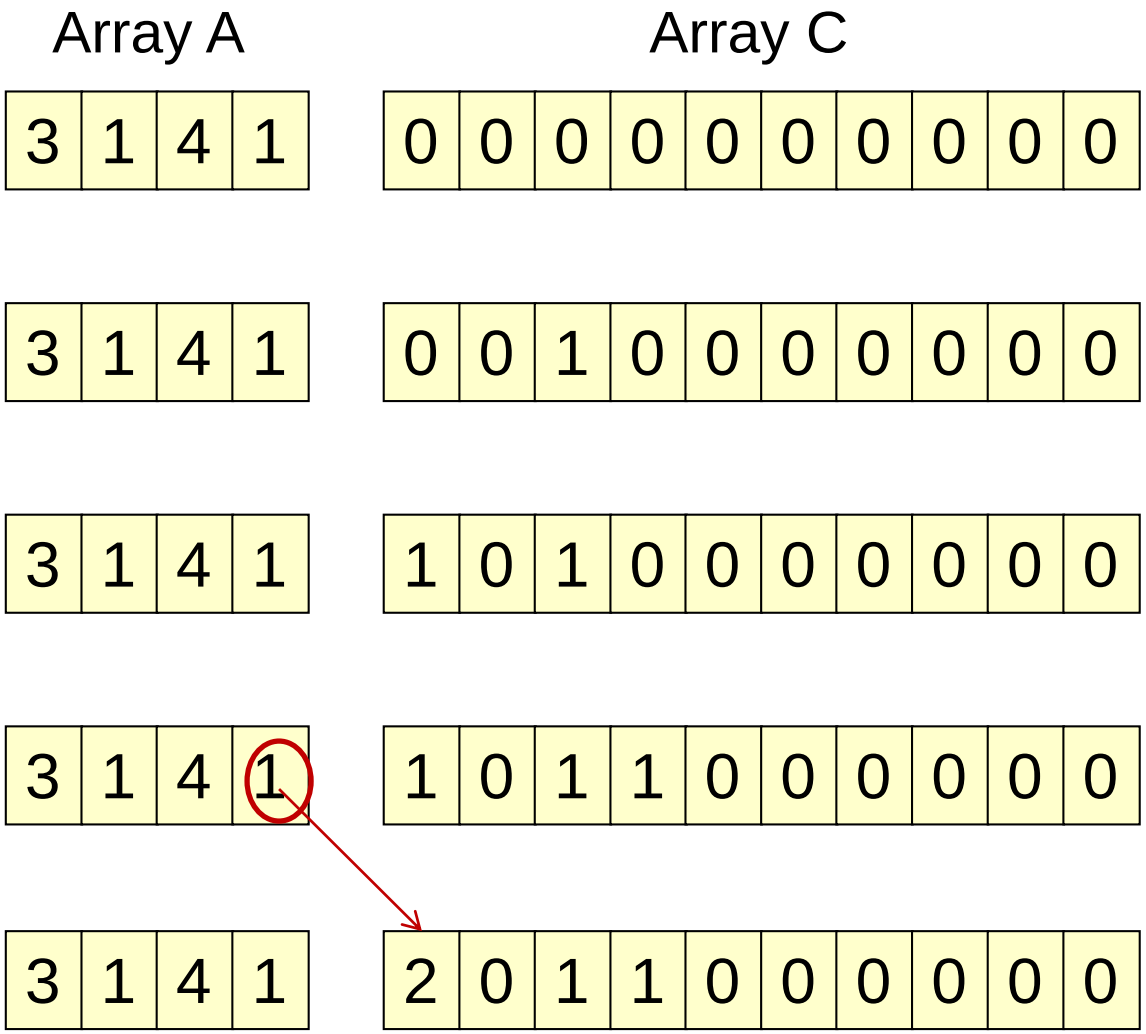
Counting Sort (contagem dos elementos)



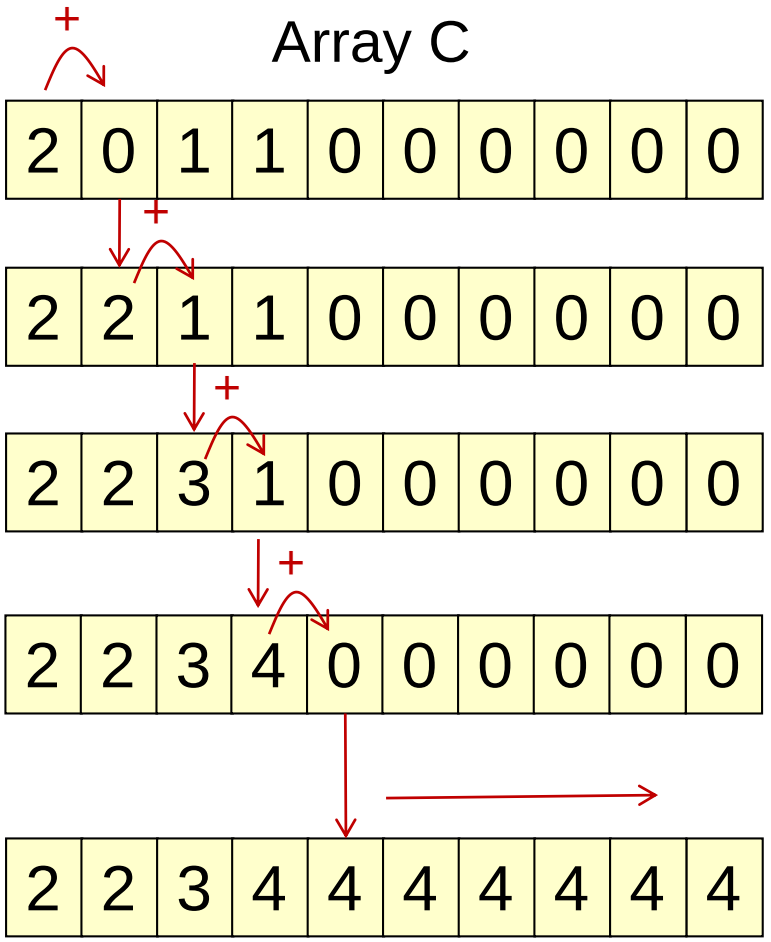
Counting Sort (contagem dos elementos)



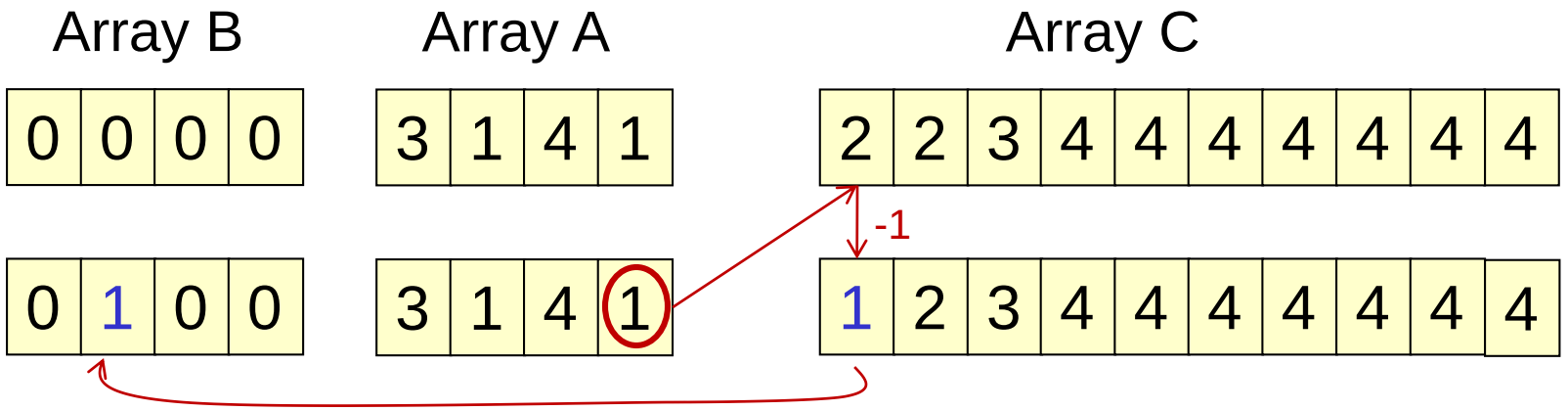
Counting Sort (contagem dos elementos)



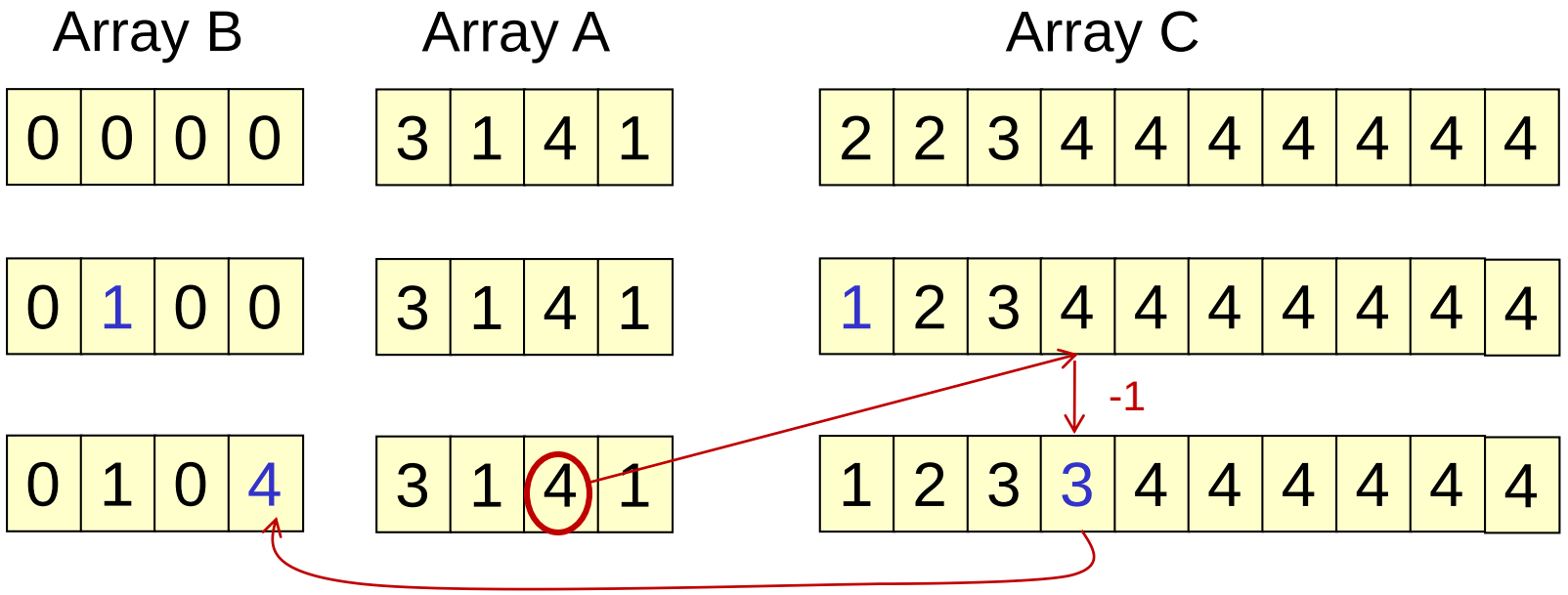
Counting Sort (Soma dos prefixos)



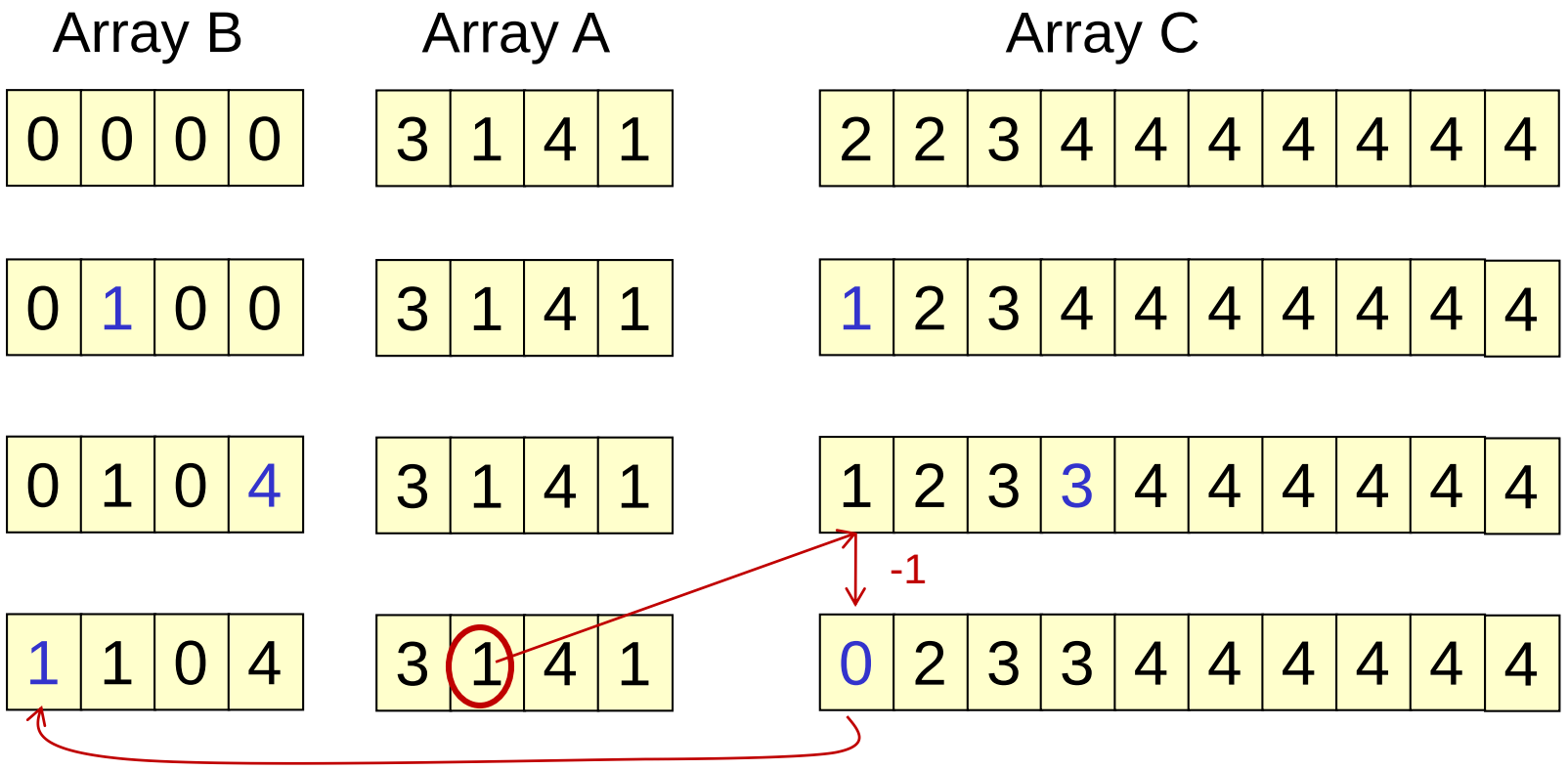
Counting Sort (Array Ordenado)



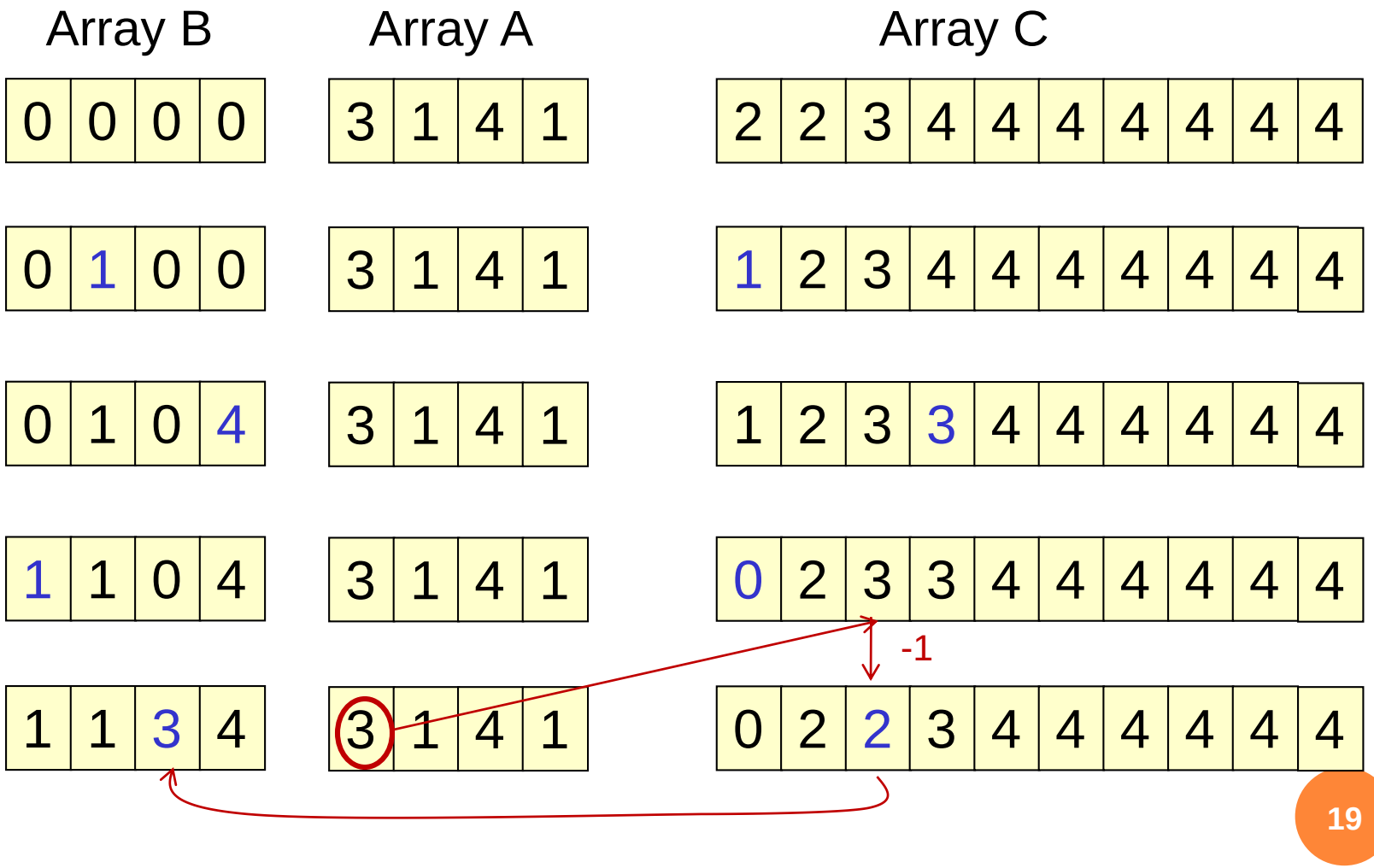
Counting Sort (Array Ordenado)



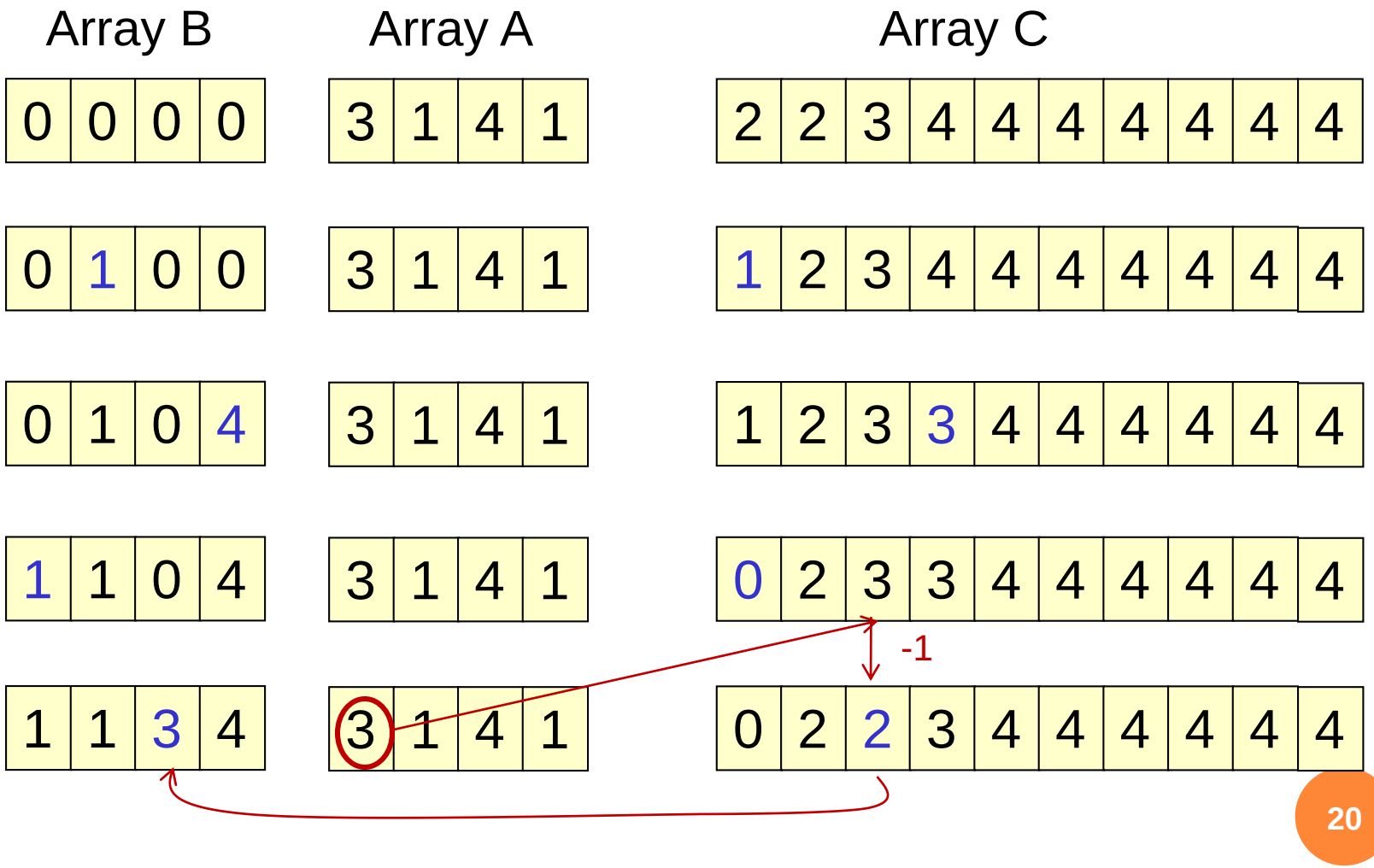
Counting Sort (Array Ordenado)



Counting Sort (Array Ordenado)



Counting Sort (Array Ordenado)



Counting Sort

- Como seria o algoritmo do Counting Sort?

```
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  ▷ C[i] now contains the number of elements equal to i.
 6  for i ← 2 to k
 7       do C[i] ← C[i] + C[i − 1]
 8  ▷ C[i] now contains the number of elements less than or equal to i.
 9  for j ← length[A] downto 1
10       do B[C[A[j]]] ← A[j]
11          C[A[j]] ← C[A[j]] − 1
```

Counting Sort

- Qual o tempo do Counting Sort?

```
COUNTING-SORT(A, B, k)
 1  for i ← 1 to k                                  } Θ(k)
 2      do C[i] ← 0
 3  for j ← 1 to length[A]                          } Θ(n)
 4      do C[A[j]] ← C[A[j]] + 1
 5  ▷ C[i] now contains the number of elements equal to i.
 6  for i ← 2 to k                                  } Θ(k)
 7      do C[i] ← C[i] + C[i − 1]
 8  ▷ C[i] now contains the number of elements less than or equal to i.
 9  for j ← length[A] downto 1                      } Θ(n)
10      do B[C[A[j]]] ← A[j]
11         C[A[j]] ← C[A[j]] − 1
```

Características

- Fácil de implementar
- Fácil de entender
- Stable (mantém a ordem)
- Precisa de mais memória (não é in place)
  - Pior caso: O(k+n)
    - k: Array de contagem no intervalo 1..k
    - Nao faz comparacoes, usa o indice do array
    - Ótimo quando k=O(n)
- Assume que os elementos estão distribuídos em um intervalo pequeno (1..k)

Características

- Otimização
  - Ver o valor mínimo e máximo de A: Θ(n)
  - Depois criamos C
  - Isso evita trabalhar com a faixa de valores 1..max(A)
- O tamanho de C depende do domínio dos números.
  - Isso pode se tornar impraticável (tempo e memória) se tivermos um número k muito grande.

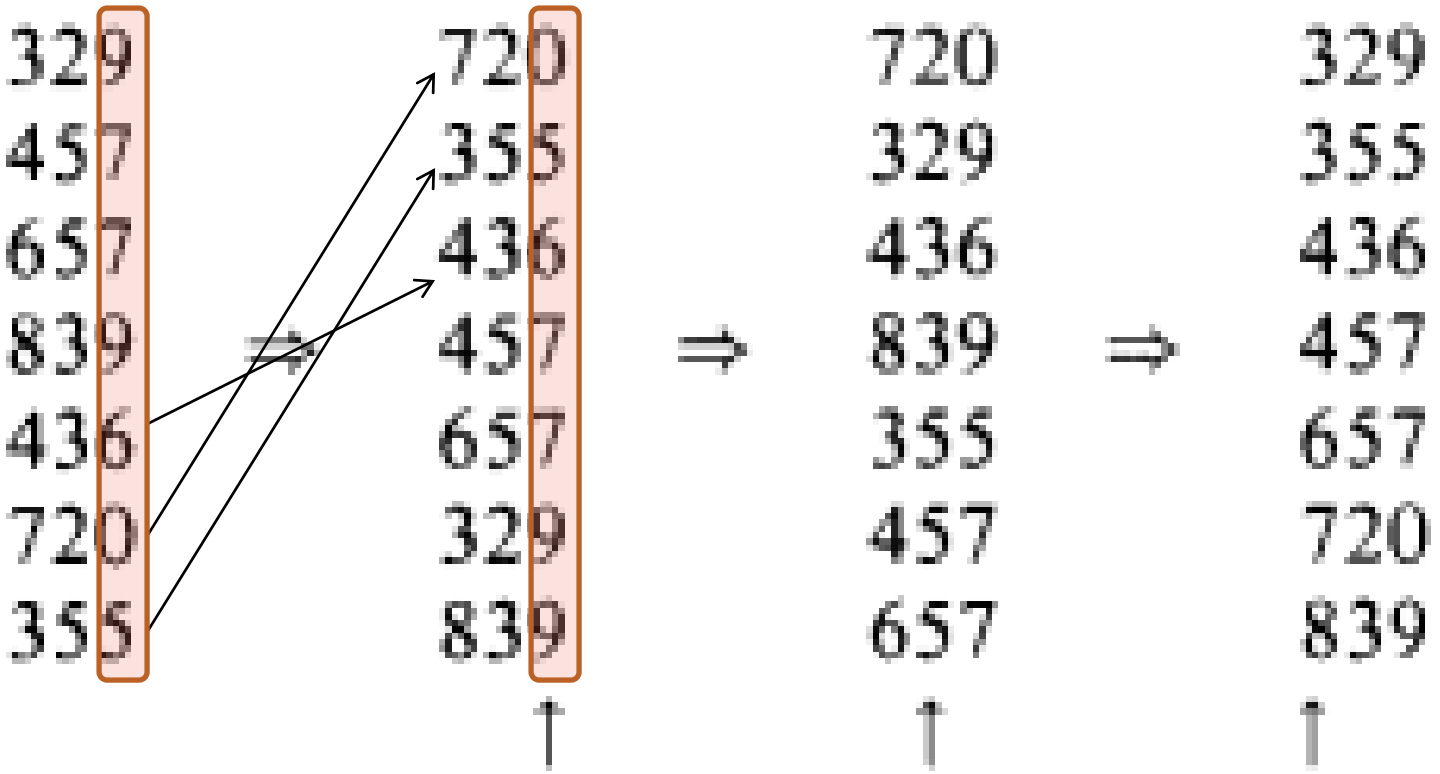
Radix Sort

Radix Sort

- Origens na máquina para ordenar cartões (1890)
- Censo americano
- Base do sistema de numeração que trabalha dígito-a-dígito
  - A partir do menos significativo (LSD)
  - A partir do mais significativo (MSD)
    - Pode ser implementado de forma recursiva

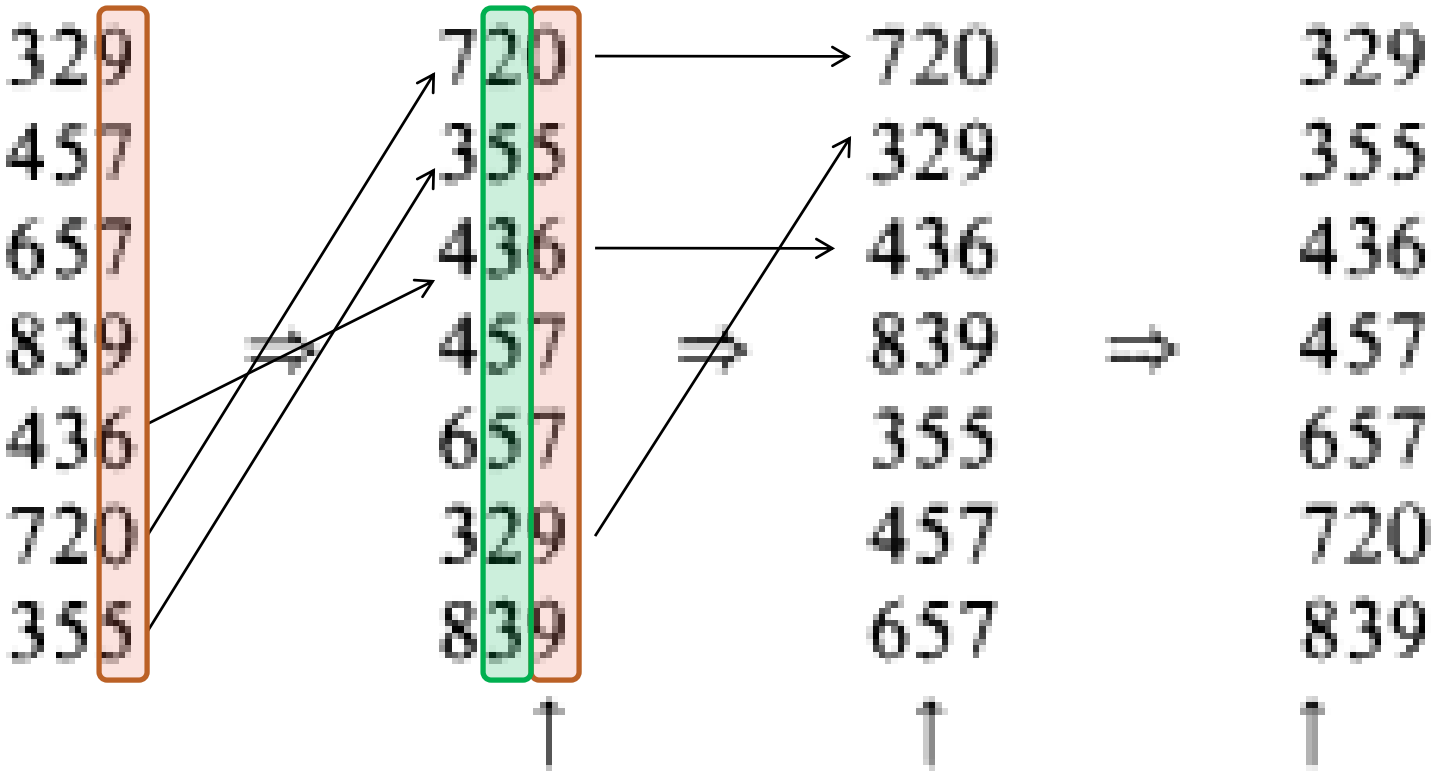
Radix Sort (LSD)

- Exemplo



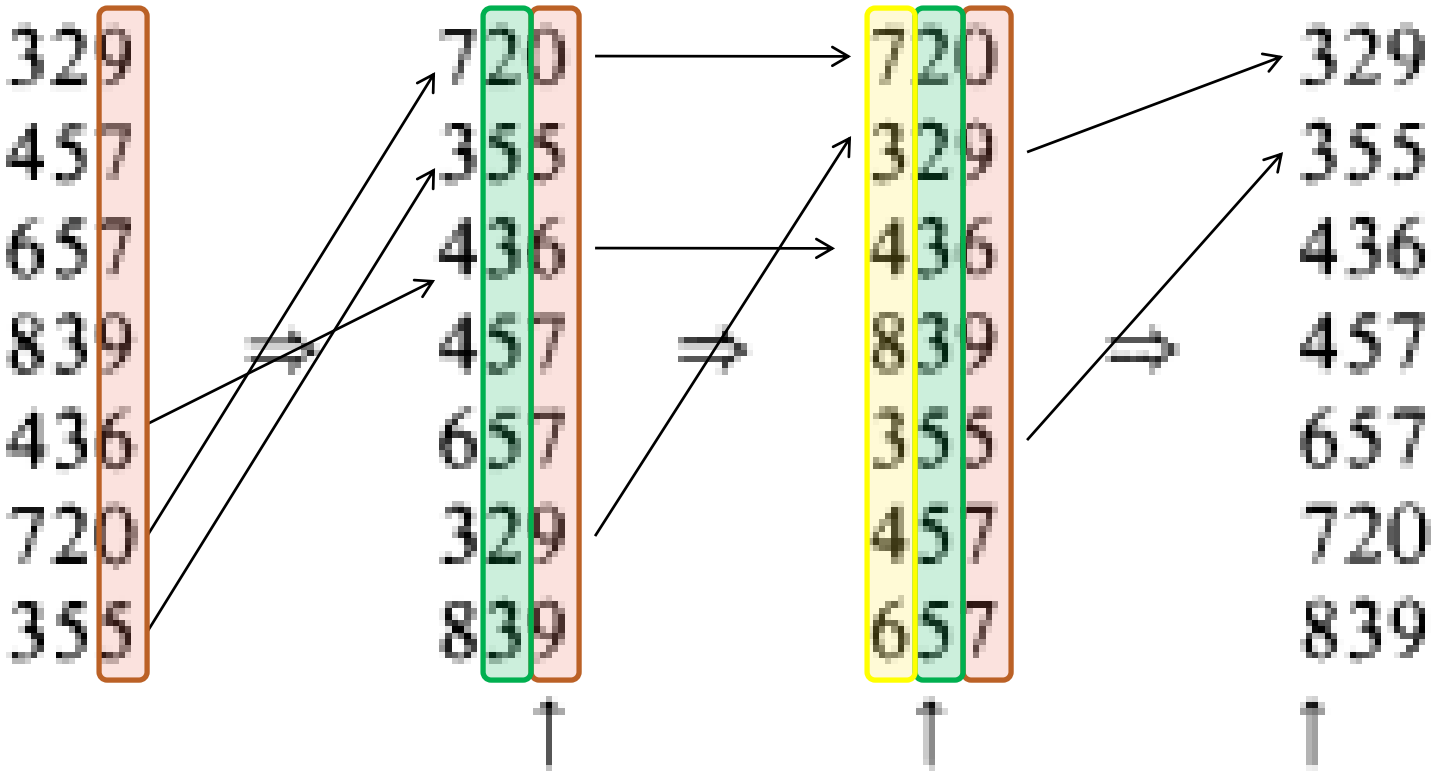
Radix Sort

- Exemplo



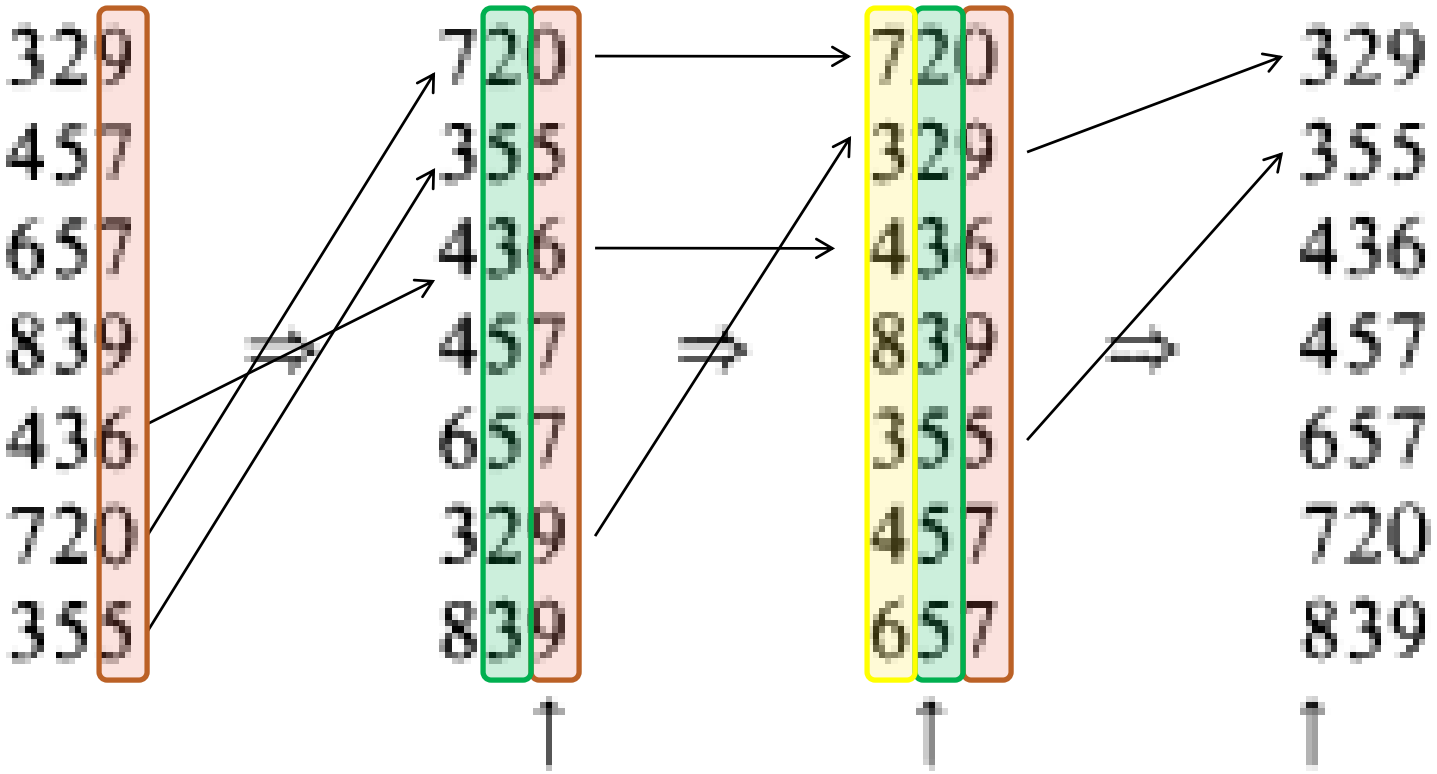
Radix Sort

- Exemplo



Radix Sort

- Exemplo



Radix Sort

- Como seria o algoritmo do Radix Sort?

```
radixSort( int [] A, int d) {
    for (i = 1 to d) //a partir do menos significativo
        ordene (stable) array A no dígito i
}
```

Radix Sort

- Qual o tempo do Radix Sort no pior caso?

```
radixSort( int [] A, int d) {
    for (i = 1 to d)
        ordene (stable) array A no dígito i
}
```

d repeticoes

depende do algoritmo

Counting sort
$\Theta(n+k)$

Radix Sort

- Análise
  - Stable
  - Θ(nd + kd)
    - Counting Sort
      - k: domínio dos possíveis valores
      - n: número de elementos
      - d: número de dígitos
    - Bom quando d for bem menor do que n: O(n)
    - Não é in place (precisa de memória extra) O(n)
  - Particularmente eficiente quando se tem muitos numeros a ordenar mas as chaves são de tamanho pequeno.

POSCOMP 2008

Questão 37

Assinale a afirmativa ***INCORRETA***.

A) Seja $A[1,n]$ um vetor não ordenado de inteiros com um número constante k de valores distintos. Então existe algoritmo de ordenação por contagem que ordena A em tempo linear.

B) Seja $A[1,n]$ um vetor não ordenado de inteiros com um número constante k de valores distintos, então o limite inferior para um algoritmo de ordenação por comparações para ordenar A é de $O(n \lg n)$.

C) Seja $A[1,n]$ um vetor não ordenado de inteiros, cada inteiro com no máximo d dígitos, onde cada dígito assume um valor entre um número constante k de valores distintos. Então o problema de ordenar A tem limite inferior $O(n)$.

D) Seja $A[1,n]$ um vetor não ordenado de inteiros, cada inteiro com no máximo d dígitos, onde cada dígito assume um valor entre $O(n)$ valores distintos. Então o problema de ordenar A tem limite inferior $O(n \lg n)$.

E) Seja $A[1,n]$ um vetor não ordenado de inteiros com um número constante k de valores distintos, então um um algoritmo de ordenação por comparações ótimo para ordenar A tem complexidade $O(n \lg n)$.

POSCOMP 2008

Questão 37

Assinale a afirmativa ***INCORRETA***.

A) Seja $A[1,n]$ um vetor não ordenado de inteiros com um número constante k de valores distintos. Então existe algoritmo de ordenação por contagem que ordena A em tempo linear.

B) Seja $A[1,n]$ um vetor não ordenado de inteiros com um número constante k de valores distintos, então o limite inferior para um algoritmo de ordenação por comparações para ordenar A é de $O(n \lg n)$.

C) Seja $A[1,n]$ um vetor não ordenado de inteiros, cada inteiro com no máximo d dígitos, onde cada dígito assume um valor entre um número constante k de valores distintos. Então o problema de ordenar A tem limite inferior $O(n)$.

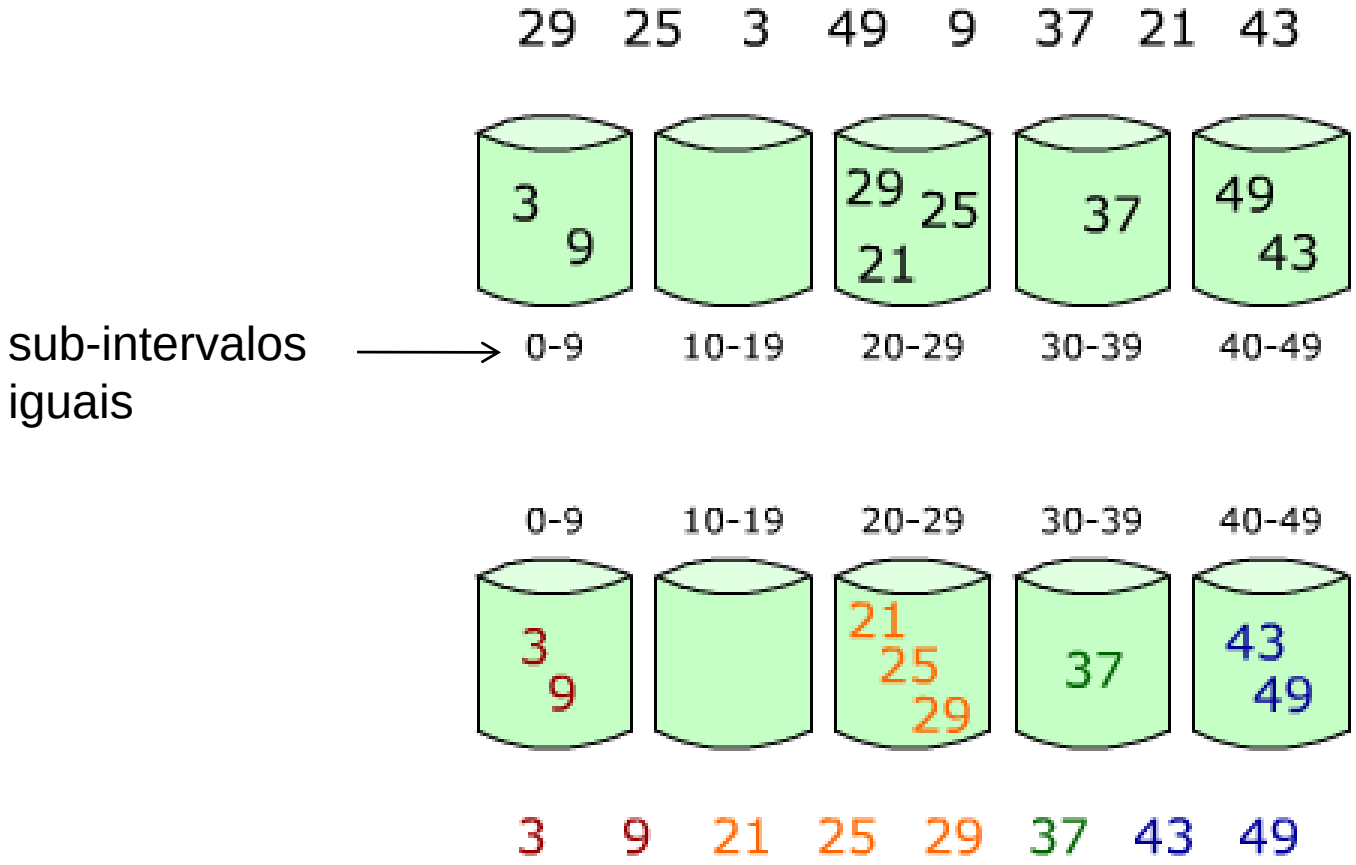
D) Seja $A[1,n]$ um vetor não ordenado de inteiros, cada inteiro com no máximo d dígitos, onde cada dígito assume um valor entre $O(n)$ valores distintos. Então o problema de ordenar A tem limite inferior $O(n \lg n)$.

E) Seja $A[1,n]$ um vetor não ordenado de inteiros com um número constante k de valores distintos, então um um algoritmo de ordenação por comparações ótimo para ordenar A tem complexidade $O(n \lg n)$.

Bucket Sort

Bucket Sort

- Intuição



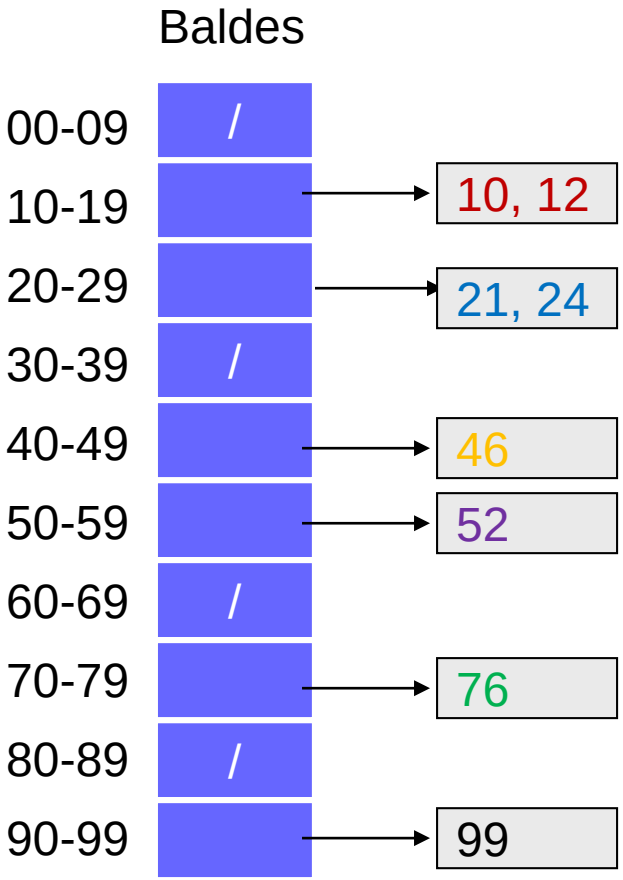
Bucket Sort

- Idéia:

1. Inicialize um vetor de "baldes", inicialmente vazios.
2. Vá para o vetor original, incluindo cada elemento em um balde.
   - Ordene todos os baldes não vazios.
   - Ou insira no balde ordenado
3. Coloque os elementos dos baldes que não estão vazios no vetor original (concatene os baldes em ordem)

Bucket Sort

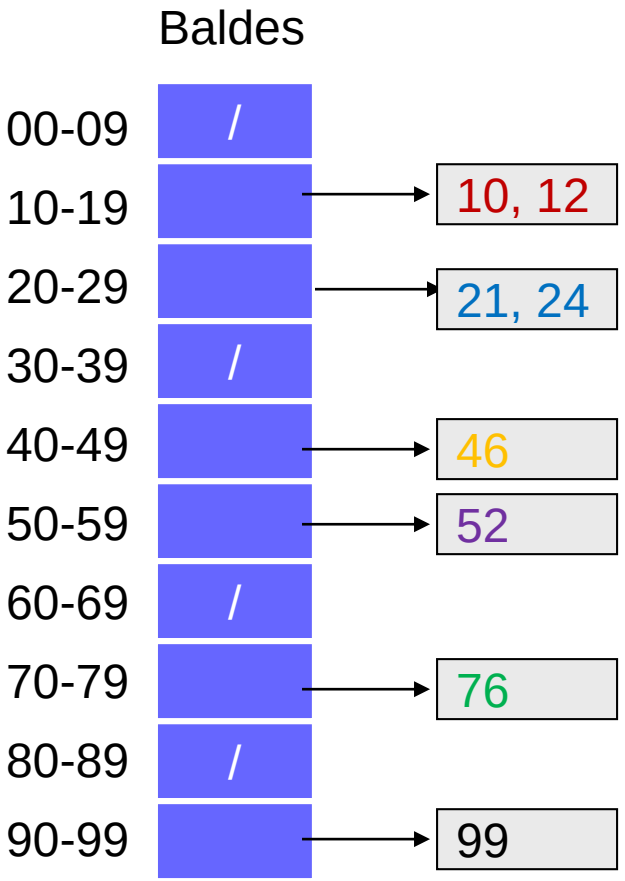
- Exercício:
  - Ordene <10,21,24,12,46,99,52,76> com 10 baldes



<10,12,21,24,46,52,76,99>

Bucket Sort

- Exercício:
  - Ordene <10,21,24,12,46,99,52,76> com 10 baldes



<10,12,21,24,46,52,76,99>

Bucket Sort

- Como seria o algoritmo do Bucket Sort?

```
function bucket-sort(array)
    buckets ← new array of 10 empty lists
    for i = 1 to array.length do
        insert array[i] into buckets[array[i]/10]
    for i = 0 to 9 do
        next-sort(buckets[i])
    return <buckets[0]>; ... ;<buckets[9]>
```

Bucket Sort

- Qual seria o custo do Bucket Sort?

```
function bucket-sort(array)
    buckets ← new array of 10 empty lists
    for i = 1 to array.length do
        insert array[i] into buckets[array[i]/10]
    for i = 0 to 9 do
        next-sort(buckets[i])
    return <buckets[0]>; ... ;<buckets[9]>
```

Características

- Análise
  - Stable
    - Varrendo os elementos da esquerda para a direita inserindo nos buckets
  - Não é in place
    - O(m + n)
      - m: faixa de valores a serem ordenados
      - n: tamanho do array
  - Assume que os elementos estão uniformemente distribuídos
    - Se não fosse tudo cairia em um único bucket no pior caso

Questões de Implementação

- Adaptando os algoritmos para trabalhar com um tipo genérico

Questões de Implementação

- Counting Sort - só funciona com arrays de Integer
- Radix Sort

```
radixSort( int [] A, int d) {
  for (i = 1 to d)
   ordene (stable) array A no dígito i
}
```

```
radixSort( T [] A, int d) {
  for (i = 1 to d)
   ordene (stable) array A no dígito i
}
```

- Bucket Sort

```
function bucket-sort(T[] array)
    buckets ← new array of 10 empty lists
    for i = 1 to array.length do
        insert array[i] into buckets[array[i]/10]
    for i = 0 to 9 do
        next-sort(buckets[i])
    return <buckets[0]>; ... ;<buckets[9]>
```

Comparação de algoritmos de ordenação

Algoritmo	Pior Caso	Caso médio
Bubble Sort	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n^2)$	$O(n^2)$
Merge Sort	O(nlog n)	O(nlog n)
Quick Sort	$O(n^2)$	O(nlog n)
Counting Sort	O(n+k)	O(n+k)
Radix Sort	O(nd+kd)	O(nd+kd)
Bucket Sort	O(m+n)	O(m+n)

Algoritmos Paralelos

- Bitonic sort
  - $O((\log n)^2)$
- Odd-even mergesort
  - $O((\log n)^2)$
- Rotate sort
  - $10n + O(n)$
- Parallel Mergesort
- Parallel Quicksort

Referências

- Capítulo 9