

Algoritmos de Ordenação

Ordenação em tempo linear

Professora:

Fátima L. S. Nunes

Algoritmos de Ordenação

- Algoritmos de ordenação que já conhecemos e suas complexidades:
 - *Insertion Sort* (Ordenação por Inserção) $O(n^2)$
 - *Selection Sort* (Ordenação por Seleção) $O(n^2)$
 - *Bubble Sort* (Ordenação pelo método da Bolha) $O(n^2)$
 - *MergeSort* (Ordenação por intercalação) $O(n \lg n)$
 - *QuickSort* (Ordenação rápida) $O(n \lg n)$ *no caso médio*
 - *HeapSort* (Ordenação por monte) $O(n \lg n)$

Algoritmos de Ordenação

- Algoritmos de ordenação que já conhecemos e suas complexidades:
 - *Insertion Sort* (Ordenação por Inserção) $O(n^2)$
 - *Selection Sort* (Ordenação por Seleção) $O(n^2)$
 - *Bubble Sort* (Ordenação pelo método da Bolha) $O(n^2)$
 - *MergeSort* (Ordenação por intercalação) $O(n \lg n)$
 - *QuickSort* (Ordenação rápida) $O(n \lg n)$ *no caso médio*
 - *HeapSort* (Ordenação por monte) $O(n \lg n)$
- Todos são algoritmos por

Algoritmos de Ordenação

- Algoritmos de ordenação que já conhecemos e suas complexidades:
 - *Insertion Sort* (Ordenação por Inserção) $O(n^2)$
 - *Selection Sort* (Ordenação por Seleção) $O(n^2)$
 - *Bubble Sort* (Ordenação pelo método da Bolha) $O(n^2)$
 - *MergeSort* (Ordenação por intercalação) $O(n \lg n)$
 - *QuickSort* (Ordenação rápida) $O(n \lg n)$ *no caso médio*
 - *HeapSort* (Ordenação por monte) $O(n \lg n)$
- Todos são algoritmos por COMPARAÇÃO

Algoritmos de Ordenação

- Será que teria outra forma de ordenar conjuntos de valores?

Algoritmos de Ordenação

- Ordenação por contagem (*Counting Sort*):
 - pressupõe que cada um dos n elementos de entrada é um inteiro no intervalo de 0 a k , para algum inteiro k .
 - ideia básica:
 - para cada elemento de entrada x , determinar o número de elementos menores que x ;
 - a informação pode ser usada para inserir o elemento x diretamente em sua posição no arranjo de saída.
 - Exemplo: se há 5 elementos menores que x , então x será inserido na 6ª posição.

Counting sort

- $A[1..n]$ é o vetor de entrada
- $B[1..n]$ é a saída ordenada
- $C[0..k]$ espaço auxiliar para fazer a contagem

CountingSort

- Algoritmo:

```
CountingSort(A[], B[], k)
para i ← 0 até k
    C[i] ← 0
fim para
para j ← 1 até tamanho(A)
    C[A[j]] ← C[A[j]] + 1 // c[i] contém número de elementos iguais a i
fim para
para i ← 1 até k
    C[i] ← C[i] + C[i-1]
    // c[i] contém número de elementos menores ou iguais a i
fim para
para j ← tamanho(A) até 1
    B[C[A[j]]] ← A[j]
    C[A[j]] ← C[A[j]] - 1
fim para
```


CountingSort

- Algoritmo:

```
CountingSort(A[], B[], k)
para i ← 0 até k
    C[i] ← 0
fim para
para j ← 1 até tamanho(A)
    C[A[j]] ← C[A[j]] + 1
// c[i] contém número de elementos
// iguais a i
fim para
para i ← 1 até k
    C[i] ← C[i] + C[i-1]
// c[i] contém número de
// elementos menores ou iguais a i
fim para
para j ← tamanho(A) até 1
    B[C[A[j]]] ← A[j]
    C[A[j]] ← C[A[j]] - 1
fim para
```

A	1	2	3	4	5	6	7	8
	2	5	3	0	2	3	0	3
C	0	1	2	3	4	5		
	2	0	2	3	0	1		

CountingSort

- Algoritmo:

```
CountingSort(A[], B[], k)
para i ← 0 até k
    C[i] ← 0
fim para
para j ← 1 até tamanho(A)
    C[A[j]] ← C[A[j]] + 1
// c[i] contém número de elementos
// iguais a i
fim para
para i ← 1 até k
    C[i] ← C[i] + C[i-1]
// c[i] contém número de
// elementos menores ou iguais a i
fim para
para j ← tamanho(A) até 1
    B[C[A[j]]] ← A[j]
    C[A[j]] ← C[A[j]] - 1
fim para
```

A	1	2	3	4	5	6	7	8
	2	5	3	0	2	3	0	3

C	0	1	2	3	4	5
	2	0	2	3	0	1

C	0	1	2	3	4	5
	2	2	4	7	7	8

CountingSort

- Algoritmo:

```
CountingSort(A[], B[], k)
para i ← 0 até k
    C[i] ← 0
fim para
para j ← 1 até tamanho(A)
    C[A[j]] ← C[A[j]] + 1
    // c[i] contém número de elementos
    // iguais a i
fim para
para i ← 1 até k
    C[i] ← C[i] + C[i-1]
    // c[i] contém número de
    // elementos menores ou iguais a i
fim para
para j ← tamanho(A) até 1
    B[C[A[j]]] ← A[j]
    C[A[j]] ← C[A[j]] - 1
fim para
```

A

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

C

0	1	2	3	4	5
2	0	2	3	0	1

C

0	1	2	3	4	5
2	2	4	7	7	8

B

1	2	3	4	5	6	7	8
						3	

C

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

Iteração 1

CountingSort

- Algoritmo:

```
CountingSort(A[], B[], k)
para i ← 0 até k
    C[i] ← 0
fim para
para j ← 1 até tamanho(A)
    C[A[j]] ← C[A[j]] + 1
    // c[i] contém número de elementos
    // iguais a i
fim para
para i ← 1 até k
    C[i] ← C[i] + C[i-1]
    // c[i] contém número de
    // elementos menores ou iguais a i
fim para
para j ← tamanho(A) até 1
    B[C[A[j]]] ← A[j]
    C[A[j]] ← C[A[j]] - 1
fim para
```

A

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

C

0	1	2	3	4	5
2	0	2	3	0	1

C

0	1	2	3	4	5
2	2	4	7	7	8

B

1	2	3	4	5	6	7	8
	0					3	

C

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

Iteração 2

CountingSort

- Algoritmo:

```
CountingSort(A[], B[], k)
para i ← 0 até k
    C[i] ← 0
fim para
para j ← 1 até tamanho(A)
    C[A[j]] ← C[A[j]] + 1
    // c[i] contém número de elementos
    // iguais a i
fim para
para i ← 1 até k
    C[i] ← C[i] + C[i-1]
    // c[i] contém número de
    // elementos menores ou iguais a i
fim para
para j ← tamanho(A) até 1
    B[C[A[j]]] ← A[j]
    C[A[j]] ← C[A[j]] - 1
fim para
```

A

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

C

0	1	2	3	4	5
2	0	2	3	0	1

C

0	1	2	3	4	5
2	2	4	7	7	8

B

1	2	3	4	5	6	7	8
	0				3	3	

C

0	1	2	3	4	5
1	2	4	5	7	8

Iteração 3

CountingSort

- Algoritmo:

```
CountingSort(A[], B[], k)
para i ← 0 até k
    C[i] ← 0
fim para
para j ← 1 até tamanho(A)
    C[A[j]] ← C[A[j]] + 1
    // c[i] contém número de elementos
    // iguais a i
fim para
para i ← 1 até k
    C[i] ← C[i] + C[i-1]
    // c[i] contém número de
    // elementos menores ou iguais a i
fim para
para j ← tamanho(A) até 1
    B[C[A[j]]] ← A[j]
    C[A[j]] ← C[A[j]] - 1
fim para
```

A

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

C

0	1	2	3	4	5
2	0	2	3	0	1

C

0	1	2	3	4	5
2	2	4	7	7	8

B

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

C

0	1	2	3	4	5
1	2	3	5	7	8

Iteração 4

CountingSort

- Algoritmo:

```
CountingSort(A[], B[], k)
para i ← 0 até k
    C[i] ← 0
fim para
para j ← 1 até tamanho(A)
    C[A[j]] ← C[A[j]] + 1
    // c[i] contém número de elementos
    // iguais a i
fim para
para i ← 1 até k
    C[i] ← C[i] + C[i-1]
    // c[i] contém número de
    // elementos menores ou iguais a i
fim para
para j ← tamanho(A) até 1
    B[C[A[j]]] ← A[j]
    C[A[j]] ← C[A[j]] - 1
fim para
```

A

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

C

0	1	2	3	4	5
2	0	2	3	0	1

C

0	1	2	3	4	5
2	2	4	7	7	8

B

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

C

0	1	2	3	4	5
0	2	3	5	7	8

Iteração 5

CountingSort

- Algoritmo:

```
CountingSort(A[], B[], k)
para i ← 0 até k
    C[i] ← 0
fim para
para j ← 1 até tamanho(A)
    C[A[j]] ← C[A[j]] + 1
    // c[i] contém número de elementos
    // iguais a i
fim para
para i ← 1 até k
    C[i] ← C[i] + C[i-1]
    // c[i] contém número de
    // elementos menores ou iguais a i
fim para
para j ← tamanho(A) até 1
    B[C[A[j]]] ← A[j]
    C[A[j]] ← C[A[j]] - 1
fim para
```

A

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

C

0	1	2	3	4	5
2	0	2	3	0	1

C

0	1	2	3	4	5
2	2	4	7	7	8

B

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

C

0	1	2	3	4	5
0	2	3	4	7	8

Iteração 6

CountingSort

- Algoritmo:

```
CountingSort(A[], B[], k)
para i ← 0 até k
    C[i] ← 0
fim para
para j ← 1 até tamanho(A)
    C[A[j]] ← C[A[j]] + 1
    // c[i] contém número de elementos
    // iguais a i
fim para
para i ← 1 até k
    C[i] ← C[i] + C[i-1]
    // c[i] contém número de
    // elementos menores ou iguais a i
fim para
para j ← tamanho(A) até 1
    B[C[A[j]]] ← A[j]
    C[A[j]] ← C[A[j]] - 1
fim para
```

A	1	2	3	4	5	6	7	8
	2	5	3	0	2	3	0	3

C	0	1	2	3	4	5
	2	0	2	3	0	1

C	0	1	2	3	4	5
	2	2	4	7	7	8

B	1	2	3	4	5	6	7	8
	0	0		2	3	3	3	5

C	0	1	2	3	4	5
	0	2	3	4	7	7

Iteração 7

CountingSort

- Algoritmo:

```
CountingSort(A[], B[], k)
para i ← 0 até k
    C[i] ← 0
fim para
para j ← 1 até tamanho(A)
    C[A[j]] ← C[A[j]] + 1
    // c[i] contém número de elementos
    // iguais a i
fim para
para i ← 1 até k
    C[i] ← C[i] + C[i-1]
    // c[i] contém número de
    // elementos menores ou iguais a i
fim para
para j ← tamanho(A) até 1
    B[C[A[j]]] ← A[j]
    C[A[j]] ← C[A[j]] - 1
fim para
```

A	1	2	3	4	5	6	7	8
	2	5	3	0	2	3	0	3

C	0	1	2	3	4	5
	2	0	2	3	0	1

C	0	1	2	3	4	5
	2	2	4	7	7	8

B	1	2	3	4	5	6	7	8
	0	0	2	2	3	3	3	5

C	0	1	2	3	4	5
	0	2	3	4	7	7

Iteração 8

CountingSort

- Analisando a complexidade:

```
CountingSort(A[], B[], k)
para i ← 0 até k
    C[i] ← 0
fim para
para j ← 1 até tamanho(A)
    C[A[j]] ← C[A[j]] + 1
// c[i] contém número de elementos
// iguais a i
fim para
para i ← 1 até k
    C[i] ← C[i] + C[i-1]
// c[i] contém número de
// elementos menores ou iguais a i
fim para
para j ← tamanho(A) até 1
    B[C[A[j]]] ← A[j]
    C[A[j]] ← C[A[j]] - 1
fim para
```

Complexity analysis annotations:

- $O(k)$ for the first loop (initializing C).
- ??? for the second loop (counting elements).
- ??? for the third loop (cumulative counts).

CountingSort

- Analisando a complexidade:

```
CountingSort(A[], B[], k)
para i ← 0 até k
    C[i] ← 0
fim para
para j ← 1 até tamanho(A)
    C[A[j]] ← C[A[j]] + 1
// c[i] contém número de elementos
// iguais a i
fim para
para i ← 1 até k
    C[i] ← C[i] + C[i-1]
// c[i] contém número de
// elementos menores ou iguais a i
fim para
para j ← tamanho(A) até 1
    B[C[A[j]]] ← A[j]
    C[A[j]] ← C[A[j]] - 1
fim para
```

O (k)

O (n)

O (k)

O (n)

CountingSort

- Analizando a complexidade:

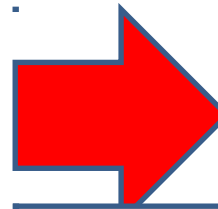
```
CountingSort(A[], B[], k)
para i ← 0 até k
    C[i] ← 0
fim para
para j ← 1 até tamanho(A)
    C[A[j]] ← C[A[j]] + 1
// c[i] contém número de elementos
// iguais a i
fim para
para i ← 1 até k
    C[i] ← C[i] + C[i-1]
// c[i] contém número de
// elementos menores ou iguais a i
fim para
para j ← tamanho(A) até 1
    B[C[A[j]]] ← A[j]
    C[A[j]] ← C[A[j]] - 1
fim para
```

$O(k)$

$O(n)$

$O(k)$

$O(n)$



$\Theta(k+n)$

**Em geral aplicamos
este método
quando $k=O(n)$.
Então: $\Theta(n)$**

Algoritmos de Ordenação

- Ordenação por contagem (*Counting Sort*):
 - supera o limite inferior de $\Omega(n \lg n)$ da ordenação por comparação;
 - propriedade importante: ordenação é **estável** $\Rightarrow$ números com mesmo valor aparecem no arranjo de saída na mesma ordem em que estavam no arranjo de entrada.
 - Quando isso é importante?

Algoritmos de Ordenação

- Ordenação por contagem (*Counting Sort*):
 - supera o limite inferior de $\Omega(n \lg n)$ da ordenação por comparação.
 - propriedade importante: ordenação é **estável** $\Rightarrow$ números com mesmo valor aparecem no arranjo de saída na mesma ordem em que estavam no arranjo de entrada.
 - Quando isso é importante?
 - quando dados adicionais são transportados junto com os números. Exemplo: Bancos de Dados.

RadixSort

- Radix sort (ordenação da raiz):
 - considera um arranjo de n inteiros, onde cada inteiro é representado com no máximo d dígitos, onde d é constante.
 - Exemplo: CEP de localidades – máximo 8 dígitos

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

- Alguma sugestão para ordenar?

RadixSort

- Algoritmo:

```
RadixSort(A[], d)
para i ← 1 até d
    ordenar os elementos de A pelo i-ésimo dígito
    usando um método estável
fim para
```

RadixSort

- Algoritmo:

```
RadixSort(A[], d)
para i ← 1 até d
    ordenar os elementos de A pelo i-ésimo dígito
    usando um método estável
fim para
```

**Por que tem que ser
estável?**

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

RadixSort

- Algoritmo:

```
RadixSort(A[], d)
para i ← 1 até d
    ordenar os elementos de A pelo i-ésimo dígito
    usando um método estável
fim para
```

**Por que tem que ser estável?
Importante manter a ordem após
ordenar cada coluna.**

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

RadixSort

- Algoritmo:

```
RadixSort(A[], d)
para i ← 1 até d
    ordenar os elementos de A pelo i-ésimo dígito
    usando um método estável
fim para
```

Por que tem que ser estável?
Importante manter a ordem após ordenar cada coluna.

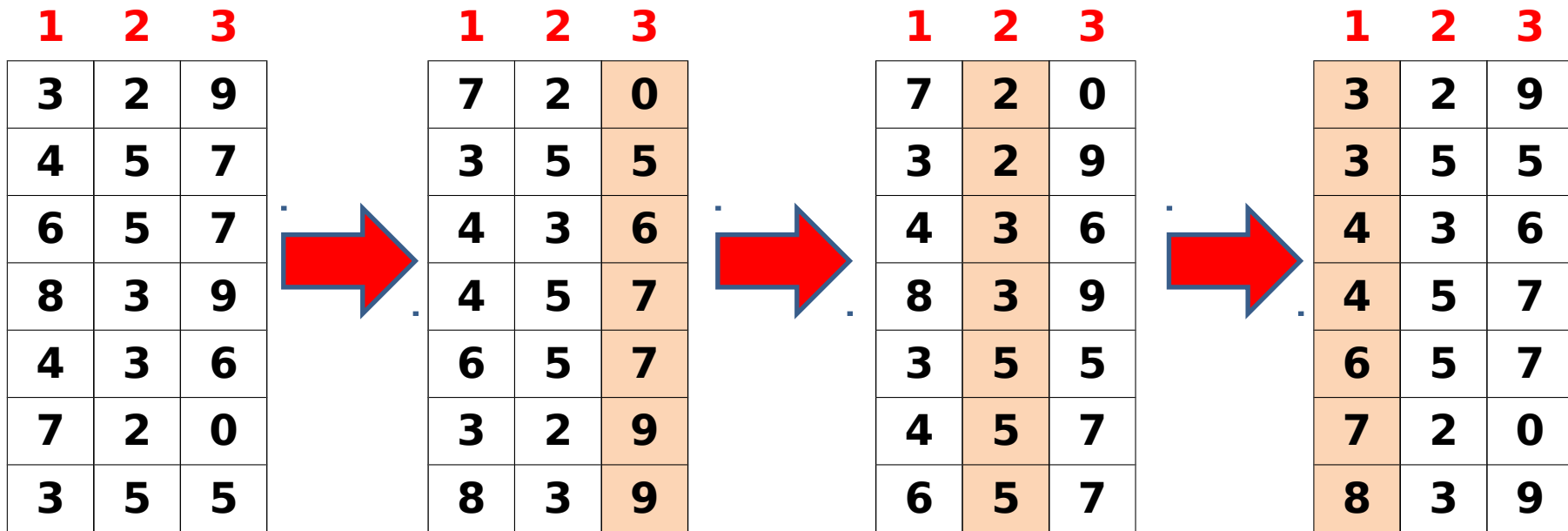
ATENÇÃO: 1 é o elemento de mais baixa ordem
e d é o elemento de mais alta ordem.

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

RadixSort

- Algoritmo:

```
RadixSort(A[], d)
para i ← 1 até d
    ordenar os elementos de A pelo i-ésimo dígito
    usando um método estável
fim para
```



RadixSort

- Analisando a complexidade:

Lema:

Dados n números de d dígitos em cada dígito pode assumir até k valores possíveis, o algoritmo RadixSort ordena corretamente esses números no tempo $\Theta(d(n+k))$

Prova:

- indução sobre a coluna que está sendo ordenada;
- análise do tempo de execução depende da ordenação estável usada;
- quando cada dígito está no intervalo de 0 a $k-1$, e k não é muito grande, costuma-se usar *CountingSort*;
- cada passagem sobre n números de d dígitos leva tempo $\Theta(n+k)$.
Há d passagens: tempo = $\Theta(d(n+k))$

RadixSort

- Analisando a complexidade:
 - complexidade do RadixSort depende da complexidade do método *estável* usado como intermediário;
 - se o método estável apresentar tempo de execução em $\Theta(f(n))$, então complexidade do RadixSort estará em $\Theta(d \cdot f(n))$.
 - Supondo d constante, complexidade será $\Theta(f(n))$
 - Como visto, se usar o CountingSort como método de ordenação intermediário, a complexidade será $\Theta(n+k)$
 - Se $k \in O(n)$, então complexidade será linear em n .

Algoritmos *in-place*

- O que significa a expressão *in-place*?

Algoritmos *in-place*

- O que significa a expressão *in-place*?
 - “*no local*”
- O que seria um algoritmo de ordenação *in-place*?

Algoritmos *in-place*

- O que significa a expressão *in-place*?
 - “*no local*”
- O que seria um algoritmo de ordenação *in-place*?
 - a memória adicional requerida é independente do tamanho do vetor que está sendo ordenado.
 - ✓ QuickSort e HeapSort são métodos *in-place*
 - ✓ MergeSort e CountingSort não são métodos *in-place*.

Algoritmos *in-place*

- Aprendemos mais dois conceitos: *estável* e *in-place*
- Como escolher o melhor algoritmo de ordenação?

Algoritmos *in-place*

- Aprendemos mais dois conceitos: *estável* e *in-place*?
- Como escolher o melhor algoritmo de ordenação?
 - tentar conhecer o problema!

Algoritmos *in-place*

- Aprendemos mais dois conceitos: *estável* e *in-place*?
- Como escolher o melhor algoritmo de ordenação?
 - tentar conhecer o problema!
 - complexidade – tempo de execução;
 - há algum conhecimento prévio sobre a ordenação?
Dados quase ordenados?
 - há limitação do uso de memória? algoritmo por comparação ($\Omega(n \lg n)$) *in-place* pode ser melhor que algoritmo que apresenta $O(n)$;
 - há necessidade de manter ordem original dos dados?
algoritmo estável...

Referências

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein. Algoritmos - Tradução da 2a. Edição Americana. Editora Campus, 2002.
- Nota de aulas do professor Delano Beder (EACH-USP).

Algoritmos de Ordenação

Ordenação em tempo linear

Professora:

Fátima L. S. Nunes