ACH2002

Aula 17

Cota inferior para alg. baseados em comparação, e ordenação em tempo linear

(adaptados dos slides de aula da Profa. Fátima L. S. Nunes)

EACH USP

Aulas passadas

- Algoritmos de ordenação elementares
  - InsertionSort
  - SelectionSort
  - BubbleSort
  - ShellSort
- Algoritmos de ordenação eficientes
  - MergeSort
  - HeapSort
  - QuickSort (ordenação rápida)

Complexidades?

- Algoritmos de ordenação elementares
  - InsertionSort
  - SelectionSort
  - BubbleSort
  - ShellSort
- Algoritmos de ordenação eficientes
  - MergeSort
  - HeapSort
  - QuickSort (ordenação rápida)

Complexidades?

- Algoritmos de ordenação elementares
  - InsertionSort - $O(n^2)$
  - SelectionSort - $O(n^2)$
  - BubbleSort - $O(n^2)$
  - ShellSort – $O(n^2)$ – talvez um pouco menos, mas mais que O(n lg n)
- Algoritmos de ordenação eficientes
  - MergeSort – O(n lg n)
  - HeapSort – O(n lg n)
  - QuickSort (ordenação rápida) – O(n lg n) ***no caso médio***

Algoritmos de Ordenação

- Algoritmos vistos até agora compartilham uma propriedade interessante:
  - ordenação se baseia somente em comparações entre os elementos de entrada.
  - por isso, são chamados de algoritmos por ordenação por comparação.
  - veremos que qualquer ordenação por comparação deve efetuar **Ω(n lg n)** comparações no pior caso para ordenar ***n*** elementos.

O que significa isso?

Algoritmos de Ordenação

- Algoritmos vistos até agora compartilham uma propriedade interessante:
  - ordenação se baseia somente em comparações entre os elementos de entrada.
  - por isso, são chamados de algoritmos por ordenação por comparação.
  - veremos que qualquer ordenação por comparação deve efetuar **Ω(n lg n)** comparações no pior caso para ordenar ***n*** elementos.

Logo o *MergeSort* e o *HeapSort* são algoritmos assintoticamente ótimos: não existe nenhuma ordenação por comparação que seja mais rápida por mais de um fator constante.

Limites inferiores para ordenação

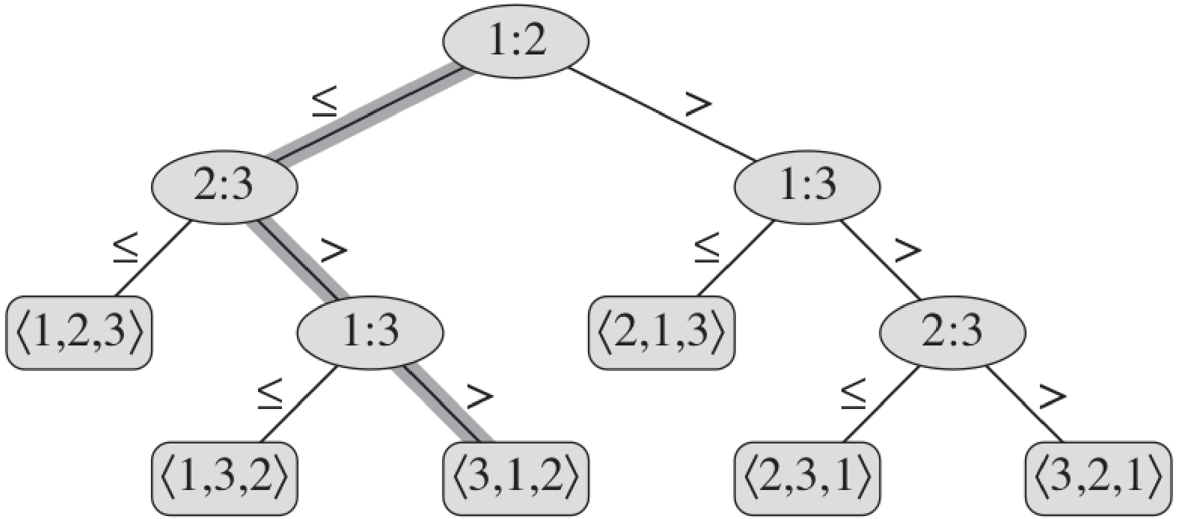
- Ordenação por comparação:
  - usamos apenas comparações entre elementos para obter informações de ordem sobre uma sequência de entrada $<a_1, a_2, \ldots, a_n>$
  - dados dois elementos de entrada a_i e a_j, usamos um teste para determinar sua ordem relativa no conjunto de dados:
    $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$, $a_i > a_j$.

Limites inferiores para ordenação

- Ordenação por comparação:
  - vamos supor que todos os elementos sejam distintos $\Rightarrow$ eliminam-se comparações $a_i = a_j$
  - as demais comparações são equivalentes porque produzem a mesma informação: ordem relativa de a_i *e* a_j
  - Então, supomos que todas as comparações são do tipo $a_i \leq a_j$

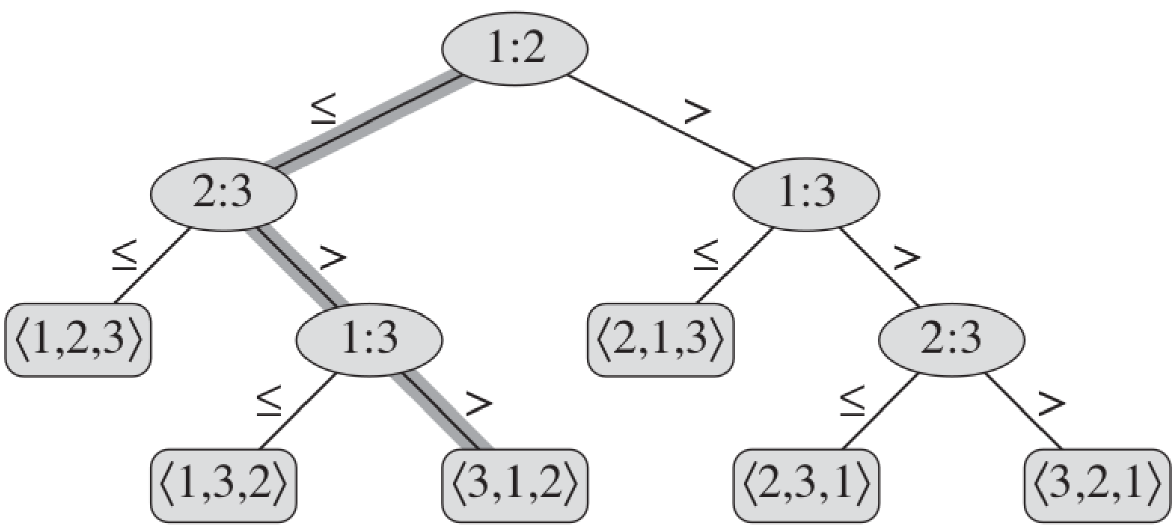
Modelo de árvore de decisão

- Ordenações por comparação podem ser vistas como árvores de decisão:
  - árvore binária cheia que representa as comparações executadas pelo algoritmo sobre uma entrada de dados de tamanho ***n***;
  - outros aspectos do algoritmo são ignorados.
  - cada nó interno da árvore é denotado por *i*:*j* (as posições dos dois elementos sendo comparados) para *i e j* no intervalo ***1 ≤ i, j ≤ n***;



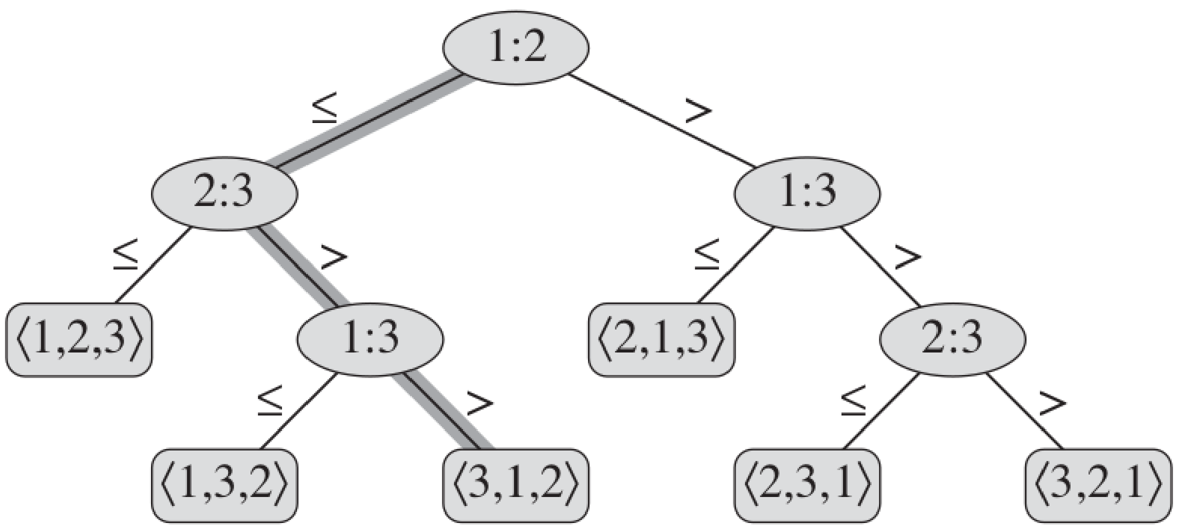
Modelo de árvore de decisão

- Ordenações por comparação podem ser vistas como árvores de decisão:
  - cada folha é denotada por uma permutação < π(1), π(2), ..., π(3)>, sendo π(i) a posição do i-ésimo elemento da sequência ordenada, ou seja, as decisões após as comparações executadas pelo algoritmo (representa a própria sequência ordenada).
  - Cada execução de um algoritmo: traçar um caminho desde a raiz até um nó folha
  - Exemplo:
    caminho para ordenar
    $<a_1=6, a_2=8, a_3=5>$



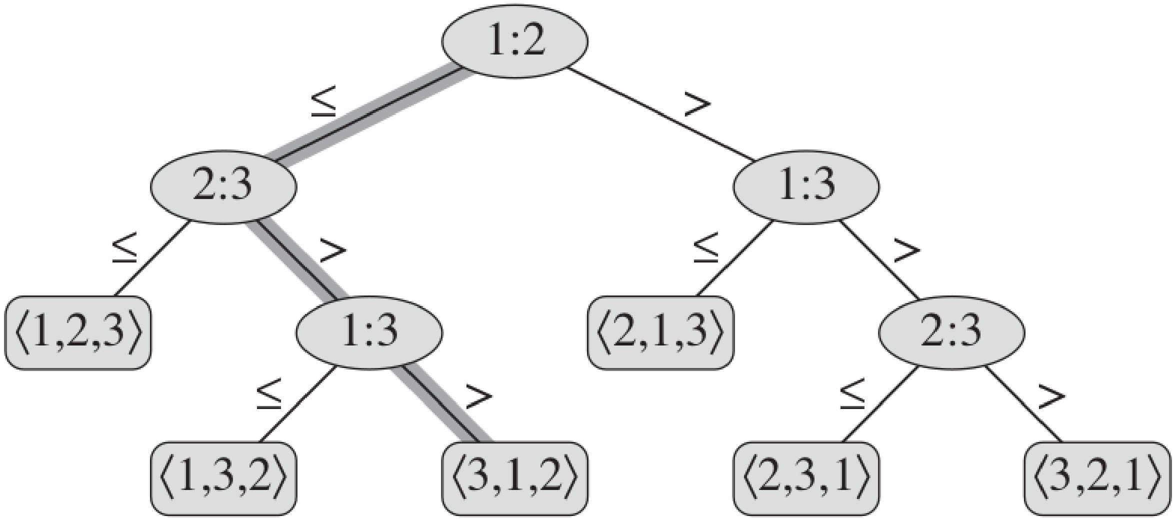
Modelo de árvore de decisão

- Ao final do algoritmo por comparação, sempre se atingirá uma folha:
- Condições necessárias para o algoritmo estar correto:
  - cada uma das ***n!*** permutações sobre ***n*** elementos deve aparecer como uma das folhas da árvore de decisão;
  - cada uma das folhas deve ser acessível por um caminho a partir da raiz.



Limite inferior para o pior caso

- Dada uma árvore de decisão, qual seria o tamanho do pior caso para um algoritmo de ordenação por comparação?
  - tamanho do caminho mais longo desde a raiz até qualquer uma de suas folhas.
  - Então: número de comparações do pior caso = altura da árvore.
  - Limite inferior sobre a altura de todas as árvores de decisão em que cada permutação aparece como uma folha acessível é um limite inferior sobre o tempo de execução do algoritmo



Limite inferior para o pior caso

- Teorema:

  Qualquer algoritmo de ordenação por comparação exige **Ω(n lg n)** comparações no pior caso.

- Prova:

  - a partir do exposto anteriormente, basta determinar a altura de uma árvore de decisão em que cada permutação aparece como uma folha acessível.
  - considerando uma árvore de altura ***b*** com ***l*** folhas acessíveis:
    - cada uma das ***n!*** permutações da entrada aparece como alguma folha (isto é, $n! \leq l$)
    - árvore binária de altura ***h*** tem no máximo 2^h folhas;
    - então: $n! \leq l \leq 2^h$
    - $h \geq lg(n!)$
    - $lg(n!) = lg(n) + lg\ (n-1) + \ldots + lg\ (1) = \Theta(n\ lg\ n)$ *(eq 3.19 Cormen)*
    - $h \geq lg(n!) \Rightarrow h = \Omega\ (n\ lg\ n)$

Limite inferior para o pior caso

- Corolário:

  O HeapSort e o MergeSort são ordenações por comparação assintoticamente ótimas

- Prova:

  - Os tempos ***O(n lg n)*** limites superiores para o HeapSort e o MergeSort correspondem ao limite inferior **Ω*(n lg n)*** do pior caso do teorema anterior.

Referências desta primeira parte da aula (com exercícios!)

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein. Algoritmos - 3a. ed. Edição Americana. Editora Campus, 2002. Cap 8.1

Ordenação em tempo linear

Aulas passadas

- Algoritmos de ordenação elementares
  - InsertionSort - $O(n^2)$
  - SelectionSort - $O(n^2)$
  - BubbleSort - $O(n^2)$
  - ShellSort – $O(n^2)$ – talvez um
- Algoritmos de ordenação efi
  - MergeSort – $O(n \lg n)$
  - HeapSort – $O(n \lg n)$
  - QuickSort (ordenação rápida) – $O(n \lg n)$ ***no caso médio***

Característica em comum:

Aulas passadas

- Algoritmos de ordenação elementares
  - InsertionSort - $O(n^2)$
  - SelectionSort - $O(n^2)$
  - BubbleSort - $O(n^2)$
  - ShellSort – $O(n^2)$ – talvez um
- Algoritmos de ordenação efi
  - MergeSort – $O(n \lg n)$
  - HeapSort – $O(n \lg n)$
  - QuickSort (ordenação rápida) – $O(n \lg n)$ ***no caso médio***

Característica em comum:

São todos baseados em **comparação de chaves** → **Ω (n lg n)**

Aula de hoje

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

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*.

Observação: e se tiver números negativos, por ex entre -30 e k'?

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*.

Observação: e se tiver números negativos, por ex entre

-30 e k'?

Soma 30 a todos os números, k = k' + 30

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 ou iguais a *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 ou iguais que *x*, então *x* será inserido na 5ª posição.

CountingSort

Algoritmo: (considere arrays iniciando em 0, mas A e B não utilizam a posição 0; C possui posições de 0 a k)

```
//A possui o vetor original, B possuirá os valores ordenados
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
```

I

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

II

C

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

III

B

1	2	3	4	5	6	7	8
						3	

C

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

IV

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
```

I

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

III

B

1	2	3	4	5	6	7	8
	0					3	

C

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

II

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
```

I

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

III

B

1	2	3	4	5	6	7	8
	0					3	

C

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

IV

Iteração 2

SISTEMAS DE INFORMAÇÃO

EACH

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
```

I

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

III

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

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

II

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
```

I

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

III

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

IV

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
```

I

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

III

B

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

C

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

II

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
```

I

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

III

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

IV

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
```

I

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

III

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
1	2	3	5	7	8

II

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
```

I

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

III

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

IV

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
```

I

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

III

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	5	7	8

II

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
```

I

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

III

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

IV

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
```

I

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

III

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	8

II

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
```

I

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

III

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

IV

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
```

I

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

III

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

II

Iteração 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
```

I

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

III

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	2	4	7	7

IV

Obs: no final c[i] conterá o nr de elementos menores que i

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
```

???

???

???

???

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

- 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)

$\Theta(k+n)$

Em geral aplicamos este método quando k=O(n).
Então: Θ(n)

CountingSort

- Ordenação por contagem (*Counting Sort*):
  - supera o limite inferior de $\Omega(n \lg n)$ da ordenação por comparação;
  - É *in loco*?
  - É estável?

CountingSort

- Ordenação por contagem (*Counting Sort*):
  - supera o limite inferior de $\Omega(n \lg n)$ da ordenação por comparação;
  - É *in loco*? Não, usa os vetores adicionais B e C
  - É estável?

CountingSort

- Ordenação por contagem (*Counting Sort*):
  - supera o limite inferior de $\Omega(n \lg n)$ da ordenação por comparação;
  - É *in loco*? Não, usa os vetores adicionais B e C
  - É estável? Sim! Por quê?

CountingSort

- Ordenação por contagem (*Counting Sort*):
  - supera o limite inferior de Ω(n lg n) da ordenação por comparação;
  - É *in loco*? Não, usa os vetores adicionais B e C
  - É estável? Sim! Por quê?

  Porque o último laço percorre A de trás para frente, e se houver elementos repetidos, os primeiros são adicionados em B em posições anteriores às das posições dos elementos repetidos seguintes

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

- Alguma sugestão para ordenar?
- Vamos considerar esse outro exemplo, com menos dígitos e mais números. Qual sua sugestão?

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

RadixSort

- Alguma sugestão para ordenar?
- Vamos considerar esse outro exemplo, com menos dígitos e mais números. Qual sua sugestão?
- Se ordenar cada coluna, mas começando com o dígito mais significativo, qual o problema?

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

RadixSort

- Alguma sugestão para ordenar?
- Vamos considerar esse outro exemplo, com menos dígitos e mais números. Qual sua sugestão?
- Se ordenar cada coluna, mas começando com o dígito mais significativo, qual o problema?

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

Você terá vários subarranjos para gerenciar, que cresce com o número de dígitos...

RadixSort

- Algoritmo:

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

SISTEMAS DE INFORMAÇÃO

EACH

RadixSort

- Algoritmo:

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

Isso basta?

RadixSort

- Algoritmo:

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

RadixSort

- Algoritmo:

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

ATENÇÃO: d é o elemento de mais baixa ordem e *1* é o elemento de mais alta ordem (os livros mostram o contrário, mas acho que pode confundir)

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 ← d até 1
      ordenar os elementos de A pelo i-ésimo dígito
      usando um método estável
fim para
```

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

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

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

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

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 **Θ(f(n))**, então complexidade do RadixSort estará em **Θ(d.f(n))**.
  - Supondo *d* constante, complexidade será **Θ(f(n))**
  - Como visto, se usar o CountingSort como método de ordenação intermediário, a complexidade será **Θ(n+k)**
  - Se **k ∈ O(n)**, então complexidade será linear em ***n***.

RadixSort x Quicksort (livro Ziviani)

6 Ordem aleatória dos registros com chaves inteiras de 32 bits

	10^4	10^5	10^6	10^7	10^8
Radixsort	1	1	1	1	1
Quicksort	3,1	3,3	2,3	2,6	2,7

Ordem ascendente dos registros com chaves inteiras de 32 bits

	10^4	10^5	10^6	10^7	10^8
Radixsort	1	1	1	1	1
Quicksort	3,1	3,4	2,3	2,6	2,6

Ordem descendente dos registros com chaves inteiras de 32 bits

	10^4	10^5	10^6	10^7	10^8
Radixsort	1	1	1	1	1
Quicksort	3,2	3,3	2,3	2,6	2,6

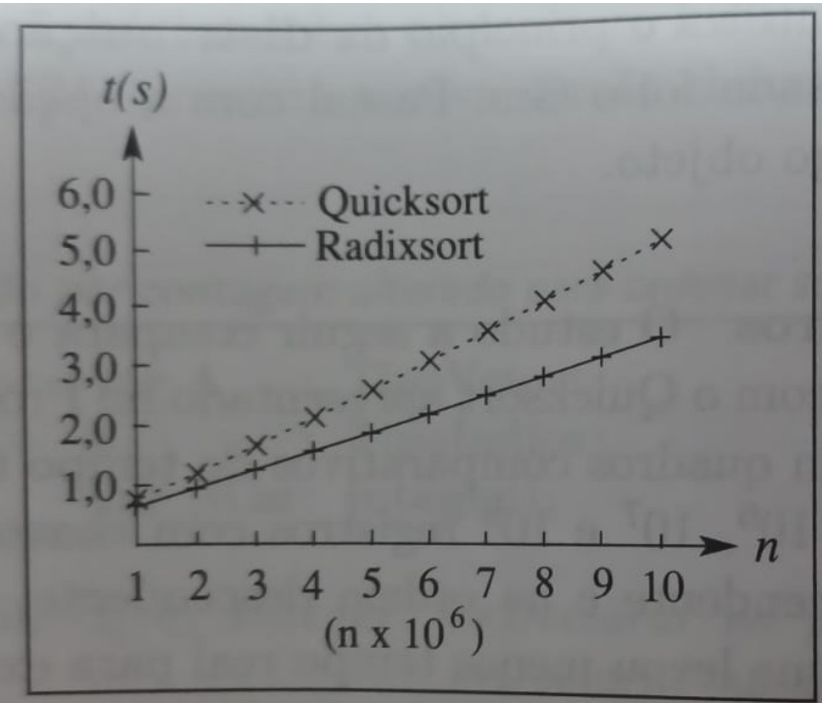


Figura 4.13 *Quicksort versus Radixsort.*

Referências da segunda parte da aula - ordenação em tempo linear (com exercícios!)

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein. Algoritmos - 3a. ed. Edição Americana. Editora Campus, 2002. Cap 8
- Nívio Ziviani. Projeto de Algoritmos com implementações em C e Pascal. Editora Thomson, 3a. Edição, 2004. Cap 4.1.7