Projeto e Análise de Algoritmos
Aula 07 – Ordenação Linear

Prof. Napoleão Nepomuceno

FUNDAÇÃO EDSON QUEIROZ
UNIVERSIDADE DE FORTALEZA
ENSINANDO E APRENDENDO

Objetivos

- Definir ordenação por comparação;
- Provar um limite inferior $\Omega(n \lg n)$ para o tempo de execução de pior caso para ordenações por comparação;
- Apresentar dois algoritmos que fazem ordenação em tempo linear:
  - Ordenação por contagem;
  - Ordenação da base (radix sort).

Ordenação por Comparação

- Dados dois elementos a_i e a_j, algoritmos de ordenação por comparação se utilizam somente de testes $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$, $a_i > a_j$ para determinar a ordem relativa dos elementos.

```
INSERTION-SORT(A)
for j ← 2 to comprimento[A]
     do chave ← A[j]
        ▷ Inserir A[j] na seqüência ordenada A[1..j – 1].
        i ← j – 1
        while i > 0 e A[i] > chave
           do A[i + 1] ← A[i]
              i ← i – 1
        A[i + 1] ← chave
```

Ordenação por Comparação

- Os algoritmos que estudamos até o momento fazem ordenação por comparação;
- Alguns têm tempo de pior caso $O(n \lg n)$:
  - Mergesort;
  - Heapsort.
- Quicksort, no caso típico, também tem tempo de execução $O(n \lg n)$.

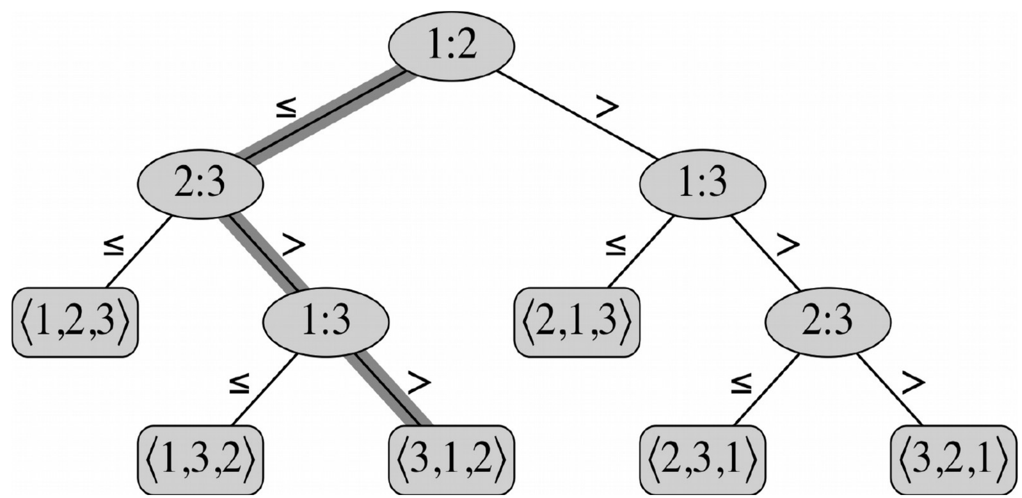
Ordenação por Comparação

- Construção de um limite inferior:
  - Vamos supor, sem perda de generalidade, que os elementos da entrada são diferentes;
  - Logo, testes da forma $a_i = a_j$ são inúteis;
  - Como os outros testes são equivalentes, podemos usar apenas comparações do tipo $a_i \leq a_j$.

Ordenação por Comparação

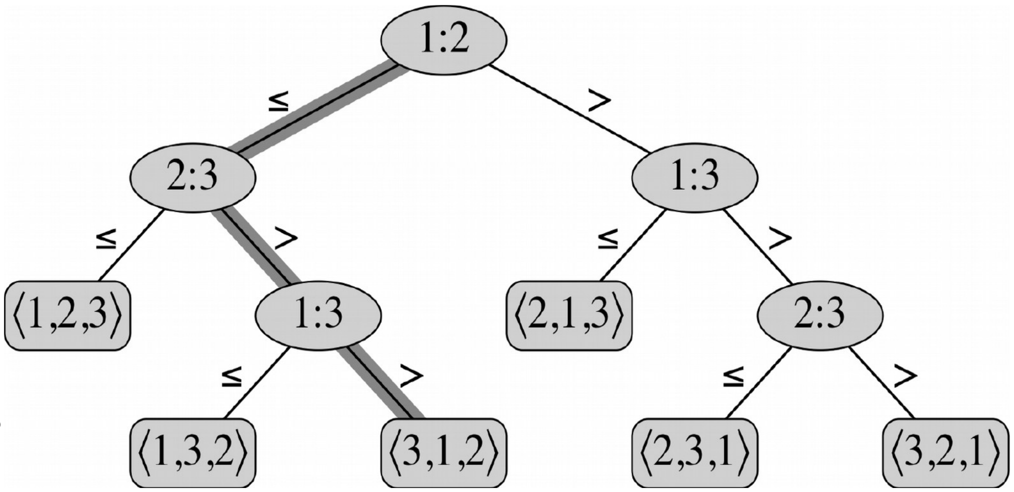
- Considere uma árvore de decisão que representa as comparações entre os elementos de uma entrada. Ex.: Insertion-Sort com 3 elementos:

```
INSERTION-SORT(A)
for j ← 2 to comprimento[A]
    do chave ← A[j]
       ▹ Inserir A[j] na seqüência ordenada A[1..j – 1].
       i ← j – 1
       while i > 0 e A[i] > chave
           do A[i + 1] ← A[i]
              i ← i – 1
       A[i + 1] ← chave
```



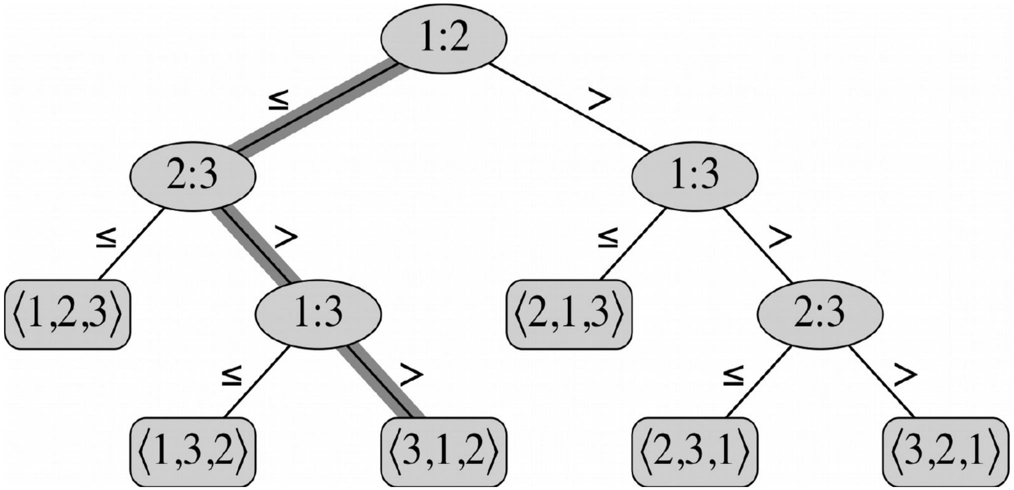
Ordenação por Comparação

- Algoritmo de ordenação correto deve ser capaz de produzir cada permutação da sua entrada;
- Ou seja:
  - Existem *n*! permutações de uma entrada de tamanho *n*;
  - Cada uma das *n*! permutações deve aparecer como uma das folhas da árvore de decisão;
  - Cada uma das folhas deve ser acessível a partir da raiz;
  - Caminho correspondente a uma execução do algoritmo.



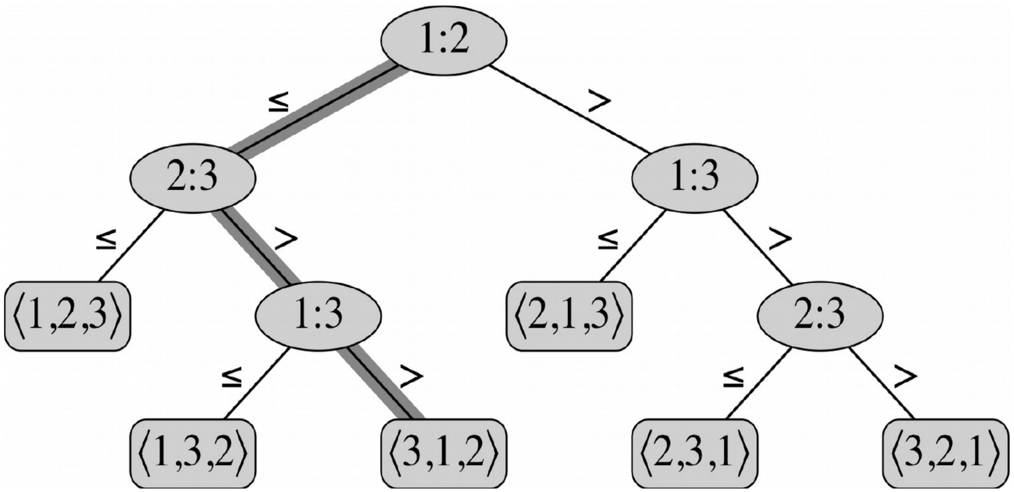
Ordenação por Comparação

- Pior caso de um algoritmo de ordenação:
  - Comprimento do caminho mais longo da raiz de uma árvore de decisão até qualquer de suas folhas;
  - O número de comparações é igual à altura da árvore de decisão do algoritmo.
- Podemos encontrar um limite inferior para a altura das árvores de decisão?



Ordenação por Comparação

- Um limite inferior para a "altura de todas as árvores de decisão nas quais cada permutação aparece como uma folha alcançável" é um limite inferior para o tempo de execução de qualquer algoritmo de ordenação por comparação.



Ordenação por Comparação

- Teorema 8.1:
  - Qualquer algoritmo de ordenação por comparação exige $\Omega\ (n \lg n)$ comparações no pior caso.
- Prova:
  - Basta determinar a altura de uma árvore de decisão em que cada permutação aparece como uma folha;
  - Dada uma árvore de altura h e l folhas, o número de permutações pode ser descrito como
    - $n! \leq l \leq 2^h$, usando logaritmos temos
    - $h \geq \lg(n!)$
    - $h = \Omega\ (n \lg n)$, (pela equação (3.18): $\lg(n!) = \Theta(n \lg n)$).

Ordenação por Comparação

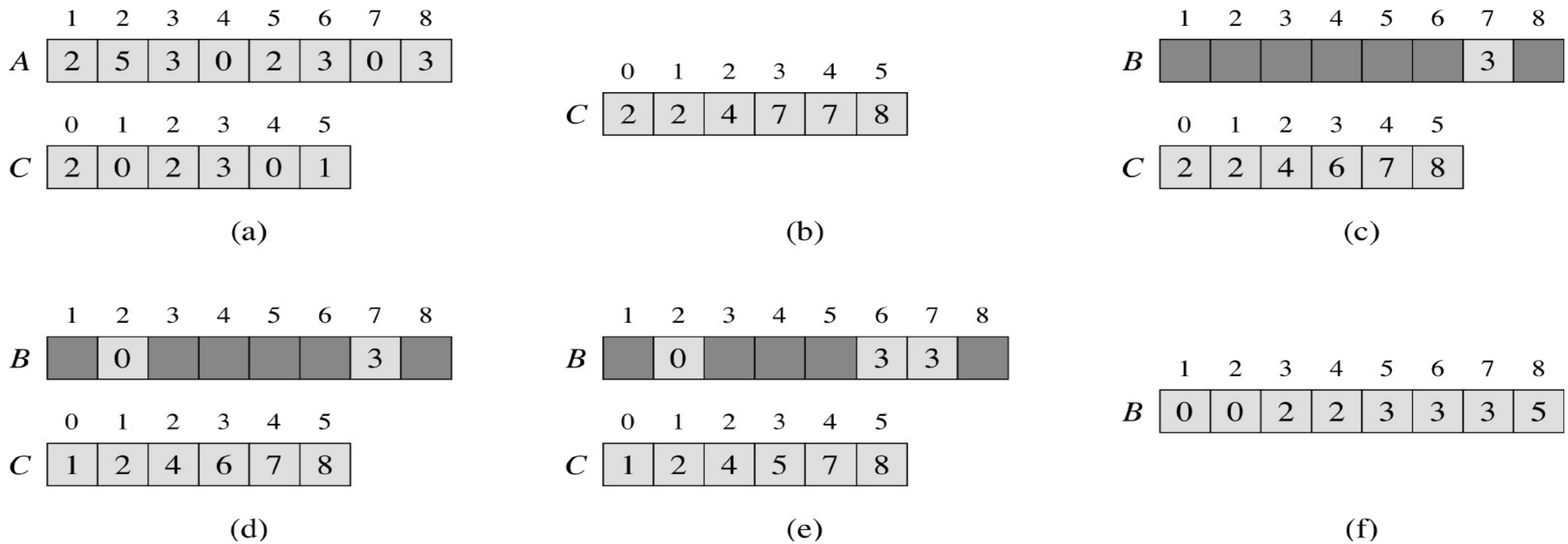
- Corolário 8.2:
  - O Heapsort e o Mergesort são algoritmos de ordenações por comparação assintoticamente ótimos.

- Prova:
  - Os dois algoritmos têm tempo de pior caso limitado superiormente por $O(n \lg n)$, o qual corresponde ao limite inferior $\Omega(n \lg n)$ do pior caso do Teorema 8.1.

Ordenação por Contagem

- Algoritmo que não trabalha com comparações;
- Assume que os números a serem ordenados são inteiros entre 0 e k, para algum inteiro k;
- Ideia básica:
  - Determinar para cada elemento x o número de elementos menores do que ele;
  - Depois basta inserir x na sua posição final.

Ordenação por Contagem




```
COUNTING-SORT(A, B, k)
let C[0..k] be a new array
for i = 0 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i - 1]
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

Ordenação por Contagem

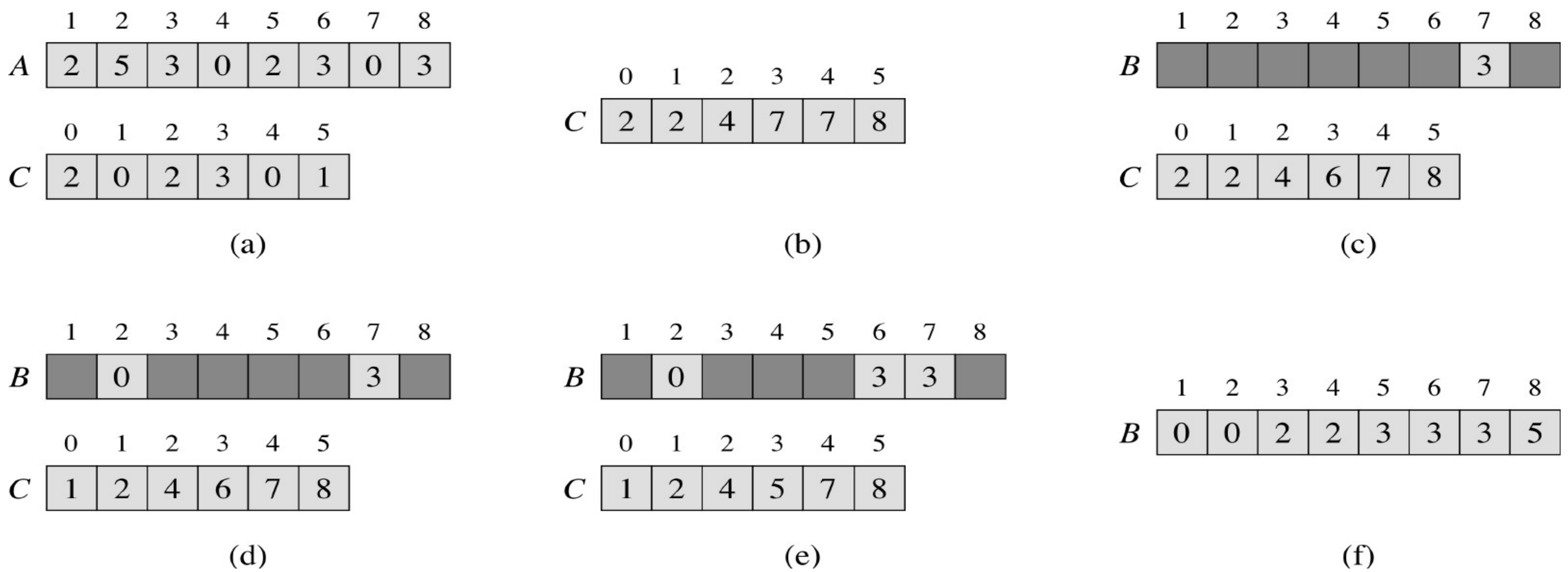
- Tempo de execução do algoritmo COUNTING-SORT?
  - *for* das linhas 1 e 2 demora $\Theta(k)$;
  - *for* das linhas 3 e 4 demora $\Theta(n)$;
  - *for* das linhas 6 e 7 demora $\Theta(k)$;
  - *for* das linhas 9 e 11 demora $\Theta(n)$;

- Tempo total é $\Theta(n + k)$.

- Quando $k = O(n)$, o tempo de execução do COUNTING-SORT é $\Theta(n)$.

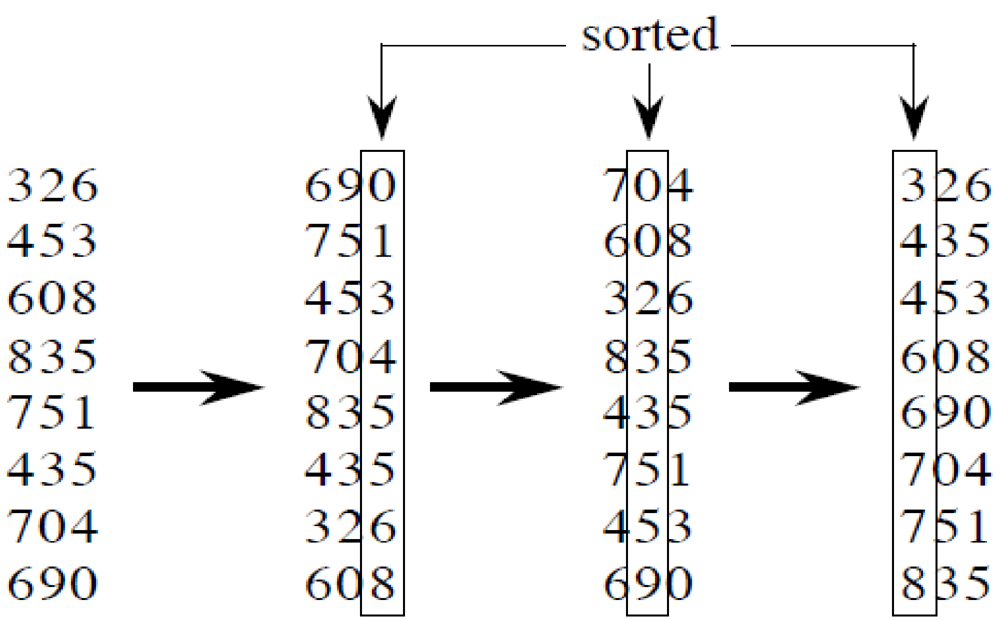
Ordenação por Contagem

- A ordenação por contagem é estável:
  - Mantém a ordem de entrada dos valores repetidos;
  - Propriedade importante para o algoritmo Radix-Sort;



Radix Sort

- Ordena do dígito menos significativo para o mais significativo, aplicando a ordenação por contagem.



Tempo de execução:

$\Theta\ (d(n + k))$

```
RADIX-SORT(A, d)
for i ← 1 to d
     do usar uma ordenação estável para ordenar o arranjo A sobre o dígito i
```