

# Análise de algoritmos

Ordenação em tempo linear

# Conteúdo

Limites inferiores para ordenação por comparações

Ordenação por contagem

Radix sort

Bucket sort

Exercícios

Referências

## Limites inferiores para ordenação por comparações

- ▶ Em um algoritmo de ordenação por comparação, a ordem dos elementos é determinada usando apenas comparações entre os elementos
- ▶ Supondo que não existem elementos iguais na entrada, todas as operações de comparação fornecem a mesma informação, e portanto podemos usar um único operador de comparação
- ▶ Vamos supor que todas as comparações têm a forma  $a_i \leq a_j$

# Limites inferiores para ordenação por comparações

- ▶ As ordenações por comparações podem ser vistas de modo abstrato em termos de **árvores de decisão**
- ▶ Uma árvore de decisão é uma árvore binária cheia que representa as comparações executadas por um algoritmo de ordenação por comparação quando ele opera em uma entrada de um tamanho dado.

# Limites inferiores para ordenação por comparações

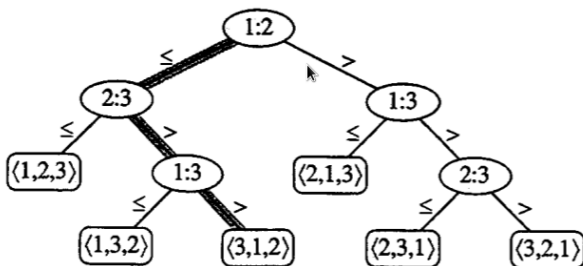
## Exemplo

- ▶  $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$

# Limites inferiores para ordenação por comparações

Exemplo

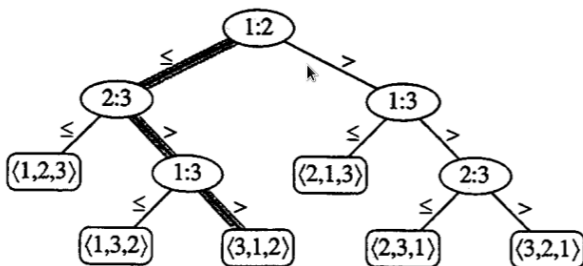
- $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$



# Limites inferiores para ordenação por comparações

## Exemplo

- ▶  $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$

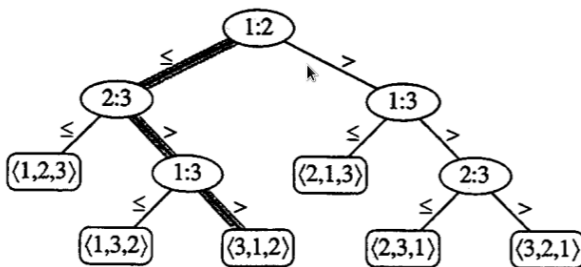


- ▶  $\langle 3, 1, 2 \rangle$  indica  $a_3 = 5 \leq a_1 = 6 \leq a_2 = 8$

# Limites inferiores para ordenação por comparações

## Exemplo

- ▶  $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$



- ▶  $\langle 3, 1, 2 \rangle$  indica  $a_3 = 5 \leq a_1 = 6 \leq a_2 = 8$

FIGURA 8.1 A árvore de decisão para ordenação por inserção, operando sobre três elementos. Um nó interno anotado por  $i:j$  indica uma comparação entre  $a_i$  e  $a_j$ . Uma folha anotada pela permutação  $\langle \pi(1), \pi(2), \dots, \pi(n) \rangle$  indica a ordenação  $a_{\pi(1)} \leq a_{\pi(2)} \leq \dots \leq a_{\pi(n)}$ . O caminho sombreado indica as decisões tomadas durante a ordenação da sequência de entrada  $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$ ; a permutação  $\langle 3, 1, 2 \rangle$  na folha indica que a sequência ordenada é  $a_3 = 5 \leq a_1 = 6 \leq a_2 = 8$ . Existem  $3! = 6$  permutações possíveis dos elementos de entrada; assim, a árvore de decisão deve ter no mínimo 6 folhas



## Limites inferiores para ordenação por comparações

- ▶ Cada nó é anotado por  $i : j$  para algum  $i$  e  $j$  no intervalo  $1 \leq i, j \leq n$
- ▶ Cada folha é anotada por uma permutação  $\langle \pi(1), \pi(2), \dots, \pi(3) \rangle$
- ▶ A execução de um algoritmo de ordenação corresponde a traçar um caminho desde a raiz até a folha
- ▶ Cada algoritmo de ordenação correto deve ser capaz de produzir cada permutação da entrada
- ▶ Cada uma das  $n!$  permutações sobre os  $n$  elementos deve aparecer como uma das folhas
- ▶ O tamanho do caminho mais longo da raiz a uma folha corresponde ao número de comparações do pior caso
- ▶ Um limite inferior sobre as alturas de todas as árvores de decisão é um limite inferior sobre o tempo de execução de qualquer algoritmo de ordenação por comparação

# Limites inferiores para ordenação por comparações

## Teorema 8.1

Qualquer algoritmo de ordenação por comparação exige  $\Omega(n \lg n)$  comparações no pior caso.

# Limites inferiores para ordenação por comparações

## Teorema 8.1

Qualquer algoritmo de ordenação por comparação exige  $\Omega(n \lg n)$  comparações no pior caso.

## Prova

- ▶ Considere um árvore de decisão com altura  $h$  com  $l$  folhas
- ▶ Cada uma das  $n!$  permutações da entrada aparecem como alguma folha, portanto  $n! \leq l$

# Limites inferiores para ordenação por comparações

## Teorema 8.1

Qualquer algoritmo de ordenação por comparação exige  $\Omega(n \lg n)$  comparações no pior caso.

## Prova

- ▶ Considere um árvore de decisão com altura  $h$  com  $l$  folhas
- ▶ Cada uma das  $n!$  permutações da entrada aparecem como alguma folha, portanto  $n! \leq l$
- ▶ Como uma árvore binária de altura  $h$  não tem mais que  $2^h$  folhas, temos  $n! \leq l \leq 2^h$

# Limites inferiores para ordenação por comparações

## Teorema 8.1

Qualquer algoritmo de ordenação por comparação exige  $\Omega(n \lg n)$  comparações no pior caso.

## Prova

- ▶ Considere um árvore de decisão com altura  $h$  com  $l$  folhas
- ▶ Cada uma das  $n!$  permutações da entrada aparecem como alguma folha, portanto  $n! \leq l$
- ▶ Como uma árvore binária de altura  $h$  não tem mais que  $2^h$  folhas, temos  $n! \leq l \leq 2^h$
- ▶ Aplicando logaritmo, obtemos  $h \geq \lg(n!)$

# Limites inferiores para ordenação por comparações

## Teorema 8.1

Qualquer algoritmo de ordenação por comparação exige  $\Omega(n \lg n)$  comparações no pior caso.

## Prova

- ▶ Considere um árvore de decisão com altura  $h$  com  $l$  folhas
- ▶ Cada uma das  $n!$  permutações da entrada aparecem como alguma folha, portanto  $n! \leq l$
- ▶ Como uma árvore binária de altura  $h$  não tem mais que  $2^h$  folhas, temos  $n! \leq l \leq 2^h$
- ▶ Aplicando logaritmo, obtemos  $h \geq \lg(n!)$
- ▶ Pela aproximação de Stirling (equação 3.18),  $n! > (n/e)^n$ , e portanto  $h = \Omega(n \lg n)$

# Ordenação por contagem

- ▶ Cada elemento da entrada é um inteiro no intervalo de 0 a  $k$
- ▶ Quando  $k = O(n)$ , a ordenação é executada no tempo  $\Theta(n)$
- ▶ Ideia
  - ▶ Determinar, para cada elemento  $x$  da entrada, o número de elementos menores que  $x$
  - ▶ Esta informação é utilizada para inserir o elemento diretamente em sua posição no arranjo de saída
  - ▶ Por exemplo, existem 17 elementos menores que  $x$ , então  $x$  é colocado na posição de saída 18
- ▶ A entrada do algoritmo é um array  $A[1..n]$ , a saída é dada em um array  $B[1..n]$ , e um array  $C[0..k]$  é utilizado para armazenamento de trabalho

# Ordenação por contagem

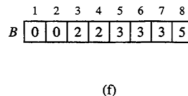
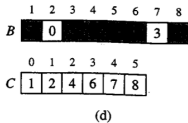
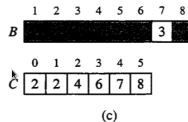
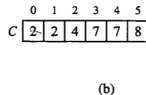
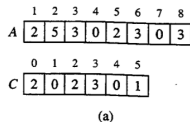


FIGURA 8.2 A operação de COUNTING-SORT sobre um arranjo de entrada  $A[1..8]$ , onde cada elemento de  $A$  é um inteiro não negativo não maior que  $k = 5$ . (a) O arranjo  $A$  e o arranjo auxiliar  $C$  após a linha 4. (b) O arranjo  $C$  após a linha 7. (c)–(e) O arranjo de saída  $B$  e o arranjo auxiliar  $C$  após uma, duas e três iterações do loop nas linhas 9 a 11, respectivamente. Apenas os elementos levemente sombreados do arranjo  $B$  foram preenchidos. (f) O arranjo de saída final ordenado  $B$



## Ordenação por contagem

```
counting-sort(A, B, k)
    n = A.comprimento
    1 for i = 0 to k
    2   C[i] = 0
    3 for j = 1 to n
    4   C[A[j]] = C[A[j]] + 1
    5 // agora C[i] contém o número de elementos
      iguais a i
    6 for i = 1 to k
    7   C[i] = C[i] + C[i - 1]
    8 // agora C[i] contém o número de elementos
      menores que o iguais a i
    9 for j = n downto 1
    10  B[C[A[j]]] = A[j]
    11  C[A[j]] = C[A[j]] - 1
```

# Ordenação por contagem

- ▶ Quanto tempo a ordenação por contagem exige?

# Ordenação por contagem

- ▶ Quanto tempo a ordenação por contagem exige?
  - ▶ O loop das linhas 1 e 2 demora o tempo  $\Theta(k)$
  - ▶ O loop das linhas 3 e 3 demora o tempo  $\Theta(n)$
  - ▶ O loop das linhas 6 e 7 demora o tempo  $\Theta(k)$
  - ▶ O loop das linhas 9 e 11 demora o tempo  $\Theta(n)$

# Ordenação por contagem

- ▶ Quanto tempo a ordenação por contagem exige?
  - ▶ O loop das linhas 1 e 2 demora o tempo  $\Theta(k)$
  - ▶ O loop das linhas 3 e 3 demora o tempo  $\Theta(n)$
  - ▶ O loop das linhas 6 e 7 demora o tempo  $\Theta(k)$
  - ▶ O loop das linhas 9 e 11 demora o tempo  $\Theta(n)$
  - ▶ Portanto, o tempo total é  $\Theta(k + n)$

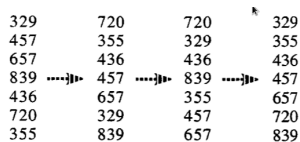
# Ordenação por contagem

- ▶ Quanto tempo a ordenação por contagem exige?
  - ▶ O loop das linhas 1 e 2 demora o tempo  $\Theta(k)$
  - ▶ O loop das linhas 3 e 3 demora o tempo  $\Theta(n)$
  - ▶ O loop das linhas 6 e 7 demora o tempo  $\Theta(k)$
  - ▶ O loop das linhas 9 e 11 demora o tempo  $\Theta(n)$
  - ▶ Portanto, o tempo total é  $\Theta(k + n)$
  - ▶ Quando  $k = O(n)$ , o tempo de execução é  $\Theta(n)$

# Radix sort

- ▶ Algoritmo usado pelas máquinas de ordenação de cartões
- ▶ Ideia
  - ▶ Ordenar as chaves pelos dígitos, começando com o menos significativo
- ▶ É essencial que o algoritmo de ordenação dos dígitos seja estável
- ▶ É utilizado para ordenar registros cuja a chave é constituída de vários campos

# Radix sort



The diagram illustrates the Radix Sort process with four columns of numbers. The first column contains the initial list: 329, 457, 657, 839, 436, 720, 355. The second column shows the list after sorting by the least significant digit (ones place), with arrows indicating the movement of elements. The third column shows the list after sorting by the middle digit (tens place). The fourth column shows the final sorted list after sorting by the most significant digit (hundreds place). A mouse cursor is visible near the top right of the table.

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

FIGURA 8.3 A operação de radix sort sobre uma lista de sete números de 3 dígitos. A primeira coluna é a entrada. As colunas restantes mostram a lista após ordenações sucessivas sobre posições de dígitos cada vez mais significativas. As setas verticais indicam a posição do dígito sobre o qual é feita a ordenação para produzir cada lista a partir da anterior

# Radix sort

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

The diagram illustrates the radix sort process with four columns of numbers. Arrows indicate the sequence of sorting steps: from the first column to the second, then to the third, and finally to the fourth. The numbers in each column represent the state of the list after sorting by a specific digit.

FIGURA 8.3 A operação de radix sort sobre uma lista de sete números de 3 dígitos. A primeira coluna é a entrada. As colunas restantes mostram a lista após ordenações sucessivas sobre posições de dígitos cada vez mais significativas. As setas verticais indicam a posição do dígito sobre o qual é feita a ordenação para produzir cada lista a partir da anterior

```
radix-sort(A, d)
```

```
1 for i = 1 to d
```

```
2   usar uma ordenação estável para ordenar  
   o arranjo A sobre o dígito i
```



# Radix sort

- ▶ Análise

- ▶ Suponha que a ordenação por contagem é utilizada como ordenação intermediária
- ▶  $\Theta(n + k)$  por passagem (dígitos estão no intervalo  $0, \dots, k$ )
- ▶  $d$  passagens
- ▶ Total de  $\Theta(d(n + k))$
- ▶ Se  $k = O(n)$  e  $d$  constante, obtemos  $\Theta(n)$

# Bucket sort

- ▶ A entrada é gerada por um processo aleatório que distribui os elementos uniformemente sobre o intervalo  $[0, 1)$
- ▶ Ideia
  - ▶ Dividir o intervalo  $[0, 1)$  em  $n$  baldes do mesmo tamanho
  - ▶ Distribuir os  $n$  valores de entrada nos baldes
  - ▶ Ordenar cada balde
  - ▶ Juntar os elementos de todos os baldes

# Bucket sort

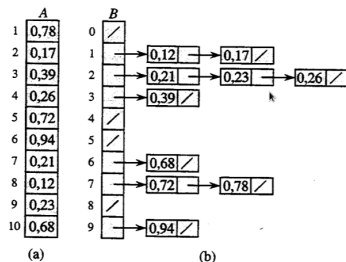


FIGURA 8.4 A operação de BUCKET-SORT. (a) O arranjo de entrada  $A[1 \dots 10]$ . (b) O arranjo  $B[0 \dots 9]$  de listas ordenadas (baldes) depois da linha 5 do algoritmo. O balde  $i$  contém valores no intervalo  $[i/10, (i + 1)/10)$ . A saída ordenada consiste em uma concatenação em ordem das listas  $B[0], B[1], \dots, B[9]$

# Bucket sort

- ▶ Entrada:  $A[1..n]$ , onde  $0 \leq A[i] < 1$  para todo  $i$
- ▶ Auxiliar: array  $B[0..n - 1]$  de listas ligadas, cada lista começa vazia

```
bucket-sort(A)
1 n = A.comprimento
2 for i = 1 to n
3   inserir A[i] na lista B[floor(n * A[i])]
4 for i = 1 to n - 1
5   insertion-sort(B[i])
6 concatenar as listas B[0], B[1], ..., B[n - 1]
   em ordem
```

# Bucket sort

## ► Análise

- Cada balde não deve ter muitos valores
- Todas as linhas do algoritmo, exceto a do ordenação por inserção, demoram  $\Theta(n)$
- Intuitivamente, cada balde terá um número constante, já que a média é um elemento por balde
- Portanto, cada balde é ordenado em  $O(1)$
- O tempo para ordenar todos os baldes é  $O(n)$
- O tempo esperado de execução do algoritmo é  $\Theta(n)$

# Exercícios

- 8.1-1 Qual é a menor profundidade possível de uma folha em uma árvore de decisão para uma ordenação por comparação?
- 8.2-1 Usando a figura 8.2 como modelo, ilustre a operação de counting-sort sobre o array  $A = \langle 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2 \rangle$
- 8.2-4 Descreva um algoritmo que, dados  $n$  inteiros no intervalo de 0 a  $k$ , realiza o pré-processamento de sua entrada e depois responde a qualquer consulta sobre quantos dos  $n$  inteiros recaem em um intervalo  $[a..b]$  no tempo  $O(1)$ . Seu algoritmo deve utilizar o tempo de pré-processamento  $\Theta(n + k)$ .
- 8.3-1 Usando a figura 8.3 como modelo, ilustre a operação de radix-sort sobre a seguinte lista de palavras em inglês: COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOE, FOX.
- 8.3-2 Quais dos seguintes algoritmos de ordenação são estáveis: ordenação por inserção, ordenação por intercalação, heapsort e quicksort? Forneça um esquema simples que torne estável qualquer algoritmo de ordenação. Quanto tempo e espaço adicional seu esquema requer?
- 8.4-1 Usando a figura 8.4 como modelo, ilustre a operação de bucket-sort no arranjo  $A = \langle 0, 79; 0, 16; 0, 64; 0, 39; 0, 20; 0, 89; 0, 53; 0, 71; 0, 42 \rangle$
- 8.4-2 Qual é o tempo de execução do pior caso para o algoritmo de bucket sort? Que alteração simples no algoritmo preserva seu tempo de execução esperado linear e torna seu tempo de execução no pior caso igual a  $O(n \lg n)$ .

# Referências

- ▶ Thomas H. Cormen et al. Introdução a Algoritmos. 2ª edição em português. Capítulo 8.