



Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

3

utad
Pedro Melo-Pinto 2022

Um algoritmo de

ORDENAÇÃO

é um algoritmo que organiza elementos de uma lista (conjunto sequencial de dados) com uma determinada ordem (linear e normalmente numérica ou lexicográfica).

Este problema é fundamental e ubíquo em engenharia e nas suas aplicações.

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

4

utad
Pedro Melo-Pinto 2022

RELEMBRAR



O Problema da Ordenação de Dados :

Entrada: Sequência de n números inteiros $\langle a_1, a_2, \dots, a_n \rangle$.

Saída: Permutação ou reorganização da sequência de entrada $\langle a'_1, a'_2, \dots, a'_n \rangle$ de tal forma que $a'_1 \leq a'_2 \leq \dots \leq a'_n$.

| | | |
|-------------------------------------|-----------|---|
| Instância do problema do somatório: | (entrada) | Sequência de 6 números $\langle 31, 41, 59, 26, 41, 58 \rangle$. |
| Saída esperada dessa instância: | (saída) | Permutação da entrada $\langle 26, 31, 41, 41, 58, 59 \rangle$. |

Cormen T.H., Leiserson C.E., Rivest R.L. and Stein C., 2009
Introduction to Algorithms.

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

RELEMBRAR

a ordenação é uma operação fundamental nas ciências da computação
(muitos programas utilizam-na como passo intermédio)

existe um conjunto alargado de métodos à nossa disposição

o melhor algoritmo para determinada aplicação depende de

- número de elementos a ordenar
- a organização prévia dos dados
(quase ordenado, ordenado inversamente, aleatório, etc.)
- tipo e eventuais restrições aos valores dos elementos
- a arquitectura do computador
- o tipo de armazenamento usado
- e OUTROS...

Cormen T.H., Leiserson C.E., Rivest R.L. and Stein C., 2009
Introduction to Algorithms.

5

utad

Pedro Melo-Pinto 2022

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

características dos algoritmos de ordenação

De uma maneira básica poderemos dividir os algoritmos de ordenação em dois tipos :

ordenação interna,
em que tudo se passa na memória principal
(e onde podemos tirar partido do acesso aleatório deste tipo de memória)

ordenação externa,
quando o número e dimensão dos objectos a ordenar é tal que inviabiliza o uso da memória principal

6

utad

Pedro Melo-Pinto 2022

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

7 utad
Pedro Melo-Pinto 2022

características dos algoritmos de ordenação

Os algoritmos de ordenação podem ainda considerar-se :

métodos não estáveis,
a ordem relativa dos elementos com chaves iguais é alterada durante o processo de ordenação

métodos estáveis,
a ordem relativa de elementos com chaves iguais mantém-se inalterada durante o processo

exemplo

método estável método não estável

0 4 4 5 7 9 0 4 4 5 7 9

8 utad
Pedro Melo-Pinto 2022

eficácia dos algoritmos de ordenação

A eficácia de um algoritmo de ordenação pode ser avaliada (essencialmente) através de :

A) número de passos do algoritmo necessários à ordenação de n elementos
B) número de comparações entre chaves
(especialmente quando as chaves são strings extensas)
C) número de vezes que os elementos têm de ser movidos
(apropriado para elementos de grande dimensão)

Algoritmia (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

9

utad
Pedro Melo-Pinto 2022

revisão algoritmos básicos

- ➊ ordenação por inserção (*insertion sort*)
- ➋ ordenação por selecção (*selection sort*)
- ➌ ordenação por flutuação (*bubble sort*)

Algoritmia (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

10

utad
Pedro Melo-Pinto 2022

revisão algoritmos básicos

INSERTION SORT

Seja a lista a ordenar $\langle a_1, a_2, \dots, a_n \rangle$.

No início, e depois de cada iteração do algoritmo, a lista pode considerar-se dividida em duas partes:

- ➊ a primeira parte a_1, \dots, a_{i-1} que já se encontra ordenada,
- ➋ e uma segunda parte a_i, \dots, a_n ainda por ordenar ($i \in 1, \dots, n$).

Em cada passagem, o primeiro elemento da parte não ordenada da lista move-se para a sua posição na parte ordenada da lista (é inserido).

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

11
utad
Pedro Melo-Pinto 2022

revisão algoritmos básicos

INSERTION SORT

Seja a lista a ordenar $\langle a_1, a_2, \dots, a_n \rangle$.

No início, e depois de cada iteração do algoritmo, a lista pode considerar-se dividida em duas partes:

- a primeira parte a_1, \dots, a_{i-1} que já se encontra ordenada,
- e uma segunda parte a_i, \dots, a_n ainda por ordenar ($i \in 1, \dots, n$).

Em cada passagem, o primeiro elemento da parte não ordenada da lista move-se para a sua posição na parte ordenada da lista (é inserido).

exemplo: elemento 1 da sequência de valores $\{-2, 5, 6, 1, 8, 7, 2, 7\}$

The diagram shows a sequence of 8 boxes representing a list. In the first row, the boxes contain -2, 5, 6, 1, 8, 7, 2, 7. In the second row, the boxes are identical but the 1 is highlighted with a red arrow pointing to it. In the third row, the 1 has moved to the fourth position, with a red arrow pointing to it. In the fourth row, the 1 is now in its final sorted position at index 4.

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

12
utad
Pedro Melo-Pinto 2022

revisão algoritmos básicos

INSERTION SORT

A lista pode considerar-se dividida em duas partes:

- a primeira parte a_1, \dots, a_{i-1} que já se encontra ordenada,
- e uma segunda parte a_i, \dots, a_n ainda por ordenar ($i \in 1, \dots, n$).

Em cada passagem, o primeiro elemento da parte não ordenada da lista move-se para a sua posição na parte ordenada da lista (é inserido).

The diagram shows a sequence of 8 boxes representing a list. It is divided into two columns: a left column showing the state of the list after each pass (labeled 'passagem 01' through 'passagem 05') and a right column showing the state of the list after each pass (labeled 'passagem 01' through 'passagem 05'). A large orange arrow points from the left column to the right column. To the right of the arrows, there is explanatory text.

Em cada passagem, o primeiro elemento da parte não ordenada da lista “procura” a sua posição na parte ordenada (à sua esquerda) da lista, movendo-se sucessivamente os elementos de valor superior uma posição para a direita.

Sugestão - 1

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

13

utad
Pedro Melo-Pinto 2022

revisão algoritmos básicos

INSERTION SORT

A lista pode considerar-se dividida em duas partes:

- ① a primeira parte a_1, \dots, a_{i-1} que já se encontra ordenada,
- ② e uma segunda parte a_i, \dots, a_n ainda por ordenar ($i \in 1, \dots, n$).

Em cada passagem, o primeiro elemento da parte não ordenada da lista move-se para a sua posição na parte ordenada da lista (é inserido).

| | | |
|-----------------------|-------------|--------------|
| 5 3 1 8 7 2 | | |
| 5 3 1 8 7 2 | passagem 01 | inversões: 1 |
| 3 5 1 8 7 2 | | |
| 3 5 1 8 7 2 | passagem 02 | inversões: 2 |
| 1 3 5 8 7 2 | | |
| 1 3 5 8 7 2 | passagem 03 | inversões: 0 |
| 1 3 5 8 7 2 | | |
| 1 3 5 8 7 2 | passagem 04 | inversões: 1 |
| 1 3 5 7 8 2 | | |
| 1 3 5 7 8 2 | passagem 05 | inversões: 4 |

Definition: Let $\alpha = a_0, \dots, a_{n-1}$ be a finite sequence.
An inversion is a pair of index positions, where the elements of the sequence are out of order.
Formally, an inversion is a pair (i, j) , where $i < j$ and $a_i > a_j$.

O número de passagens é fixo ($n-1$).
O número de inversões em cada passagem é variável.

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

14

utad
Pedro Melo-Pinto 2022

revisão algoritmos básicos

INSERTION SORT

A lista pode considerar-se dividida em duas partes:

- ① a primeira parte a_1, \dots, a_{i-1} que já se encontra ordenada,
- ② e uma segunda parte a_i, \dots, a_n ainda por ordenar ($i \in 1, \dots, n$).

Em cada passagem, o primeiro elemento da parte não ordenada da lista move-se para a sua posição na parte ordenada da lista (é inserido).

Como vimos, o tempo de execução depende de:

1. tamanho da entrada - naturalmente aumenta com o tamanho da entrada
2. do tipo de ordenação dos valores de entrada

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

15

utad
Pedro Melo-Pinto 2022

revisão algoritmos básicos
INSERTION SORT

A lista pode considerar-se dividida em duas partes:

- ① a primeira parte a_1, \dots, a_{i-1} que já se encontra ordenada,
- ② e uma segunda parte a_i, \dots, a_n ainda por ordenar ($i \in 1, \dots, n$).

Em cada passagem, o primeiro elemento da parte não ordenada da lista move-se para a sua posição na parte ordenada da lista (é inserido).

De todas as instruções, as que mais penalizam o desempenho de um algoritmo de ordenação são as **comparações** e as **trocas** entre elementos da lista (uma inversão tem ambas)

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

16

utad
Pedro Melo-Pinto 2022

revisão algoritmos básicos
INSERTION SORT

A lista pode considerar-se dividida em duas partes:

- ① a primeira parte a_1, \dots, a_{i-1} que já se encontra ordenada,
- ② e uma segunda parte a_i, \dots, a_n ainda por ordenar ($i \in 1, \dots, n$).

Em cada passagem, o primeiro elemento da parte não ordenada da lista move-se para a sua posição na parte ordenada da lista (é inserido).

caso-pior

$$\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \Rightarrow O(n^2)$$

comparações: $\sim \frac{1}{2} n^2$
trocas: $\sim \frac{1}{2} n^2$

| | | | | | |
|---|---|---|---|---|---|
| 8 | 7 | 5 | 3 | 2 | 1 |
| 8 | 7 | 5 | 3 | 2 | 1 |
| 7 | 8 | 5 | 3 | 2 | 1 |
| 7 | 8 | 5 | 3 | 2 | 1 |
| 5 | 7 | 8 | 3 | 2 | 1 |
| 5 | 7 | 8 | 3 | 2 | 1 |
| 5 | 7 | 8 | 3 | 2 | 1 |
| 3 | 5 | 7 | 8 | 2 | 1 |
| 3 | 5 | 7 | 8 | 2 | 1 |
| 2 | 3 | 5 | 7 | 8 | 1 |
| 2 | 3 | 5 | 7 | 8 | 1 |
| 1 | 2 | 3 | 5 | 7 | 8 |

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

17
utad
Pedro Melo-Pinto 2022

revisão algoritmos básicos

INSERTION SORT

A lista pode considerar-se dividida em duas partes:

- ① a primeira parte a_1, \dots, a_{i-1} que já se encontra ordenada,
- ② e uma segunda parte a_i, \dots, a_n ainda por ordenar ($i \in 1, \dots, n$).

Em cada passagem, o primeiro elemento da parte não ordenada da lista move-se para a sua posição na parte ordenada da lista (é inserido).

caso-melhor

$$\sum_{i=1}^{n-1} 1 = n - 1 \Rightarrow O(n)$$

comparações: $n - 1$
trocas: 0

| | | | | | |
|---|---|---|---|---|---|
| 1 | 2 | 3 | 5 | 7 | 8 |
| 1 | 2 | 3 | 5 | 7 | 8 |
| 1 | 2 | 3 | 5 | 7 | 8 |
| 1 | 2 | 3 | 5 | 7 | 8 |
| 1 | 2 | 3 | 5 | 7 | 8 |
| 1 | 2 | 3 | 5 | 7 | 8 |
| 1 | 2 | 3 | 5 | 7 | 8 |
| 1 | 2 | 3 | 5 | 7 | 8 |
| 1 | 2 | 3 | 5 | 7 | 8 |
| 1 | 2 | 3 | 5 | 7 | 8 |
| 1 | 2 | 3 | 5 | 7 | 8 |
| 1 | 2 | 3 | 5 | 7 | 8 |

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

18
utad
Pedro Melo-Pinto 2022

revisão algoritmos básicos

INSERTION SORT

A lista pode considerar-se dividida em duas partes:

- ① a primeira parte a_1, \dots, a_{i-1} que já se encontra ordenada,
- ② e uma segunda parte a_i, \dots, a_n ainda por ordenar ($i \in 1, \dots, n$).

Em cada passagem, o primeiro elemento da parte não ordenada da lista move-se para a sua posição na parte ordenada da lista (é inserido).

caso-médio

$$\sum_{i=1}^{n-1} \frac{1}{2}i = \frac{1}{2} \frac{(n-1)n}{2} \Rightarrow O(n^2)$$

comparações: $\sim \frac{1}{4}n^2$
trocas: $\sim \frac{1}{4}n^2$

| | | | | | |
|----|---|---|---|----|---|
| 7 | 2 | 5 | 9 | -1 | 1 |
| 2 | 7 | 5 | 9 | -1 | 1 |
| 2 | 7 | 5 | 9 | -1 | 1 |
| 2 | 7 | 5 | 9 | -1 | 1 |
| 2 | 5 | 7 | 9 | -1 | 1 |
| 2 | 5 | 7 | 9 | -1 | 1 |
| 2 | 5 | 7 | 9 | -1 | 1 |
| 2 | 5 | 7 | 9 | -1 | 1 |
| -1 | 2 | 5 | 7 | 9 | 1 |
| -1 | 2 | 5 | 7 | 9 | 1 |
| -1 | 1 | 2 | 5 | 7 | 9 |

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

19

utad
Pedro Melo-Pinto 2022

revisão algoritmos básicos

INSERTION SORT

A lista pode considerar-se dividida em duas partes:

- ① a primeira parte a_1, \dots, a_{i-1} que já se encontra ordenada,
- ② e uma segunda parte a_i, \dots, a_n ainda por ordenar ($i \in 1, \dots, n$).

Em cada passagem, o primeiro elemento da parte não ordenada da lista move-se para a sua posição na parte ordenada da lista (é inserido).

Características

- Fácil implementação
- Eficaz em conjuntos de dados pequenos
- Eficaz em conjuntos que estejam substancialmente ordenados: $O(n + d)$, em que d é o nº de inversões
- Mais eficaz na prática do que os outros métodos básicos: $O(n^2)$ - tais como a ordenação por selecção ou *bubble sort* (tempo médio é $n^2/4$ e é linear no melhor dos casos)
- Estável (não altera a ordem relativa de elementos com chave igual)
- *In-place* (necessita de um espaço extra de memória constante: $O(1)$)

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

20

utad
Pedro Melo-Pinto 2022

revisão algoritmos básicos

SELECTION SORT

Seja a lista a ordenar (a_1, a_2, \dots, a_n) .

No início, e depois de cada iteração do algoritmo, a lista pode considerar-se dividida em duas partes:

- ① a primeira parte a_1, \dots, a_{i-1} que já se encontra ordenada,
- ② e uma segunda parte a_i, \dots, a_n ainda por ordenar ($i \in 1, \dots, n$).

Em cada passagem, o primeiro elemento da parte não ordenada da lista é trocado com o mínimo da parte não ordenada da lista.

O algoritmo funciona da seguinte maneira:

- ④ encontrar o valor mínimo da lista
- ④ trocar este elemento com o primeiro elemento da lista, dividindo assim a lista na sua parte ordenada e não ordenada (sendo a parte ordenada o primeiro elemento)
- ④ repetir os passos anteriores, actuando sobre a parte não ordenada da lista

Em cada passagem, a parte ordenada da lista aumenta um elemento

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

21 utad Pedro Melo-Pinto 2022

revisão algoritmos básicos SELECTION SORT

O algoritmo funciona da seguinte maneira:

- Ⓐ encontrar o valor mínimo da lista
- Ⓑ trocar este elemento com o primeiro elemento da lista, dividindo assim a lista na sua parte ordenada e não ordenada (sendo a parte ordenada o primeiro elemento)
- Ⓒ repetir os passos anteriores, actuando sobre a parte não ordenada da lista

Em cada passagem, a parte ordenada da lista aumenta um elemento

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

22 utad Pedro Melo-Pinto 2022

revisão algoritmos básicos SELECTION SORT

A lista pode considerar-se dividida em duas partes:

- Ⓐ a primeira parte a_1, \dots, a_{i-1} que já se encontra ordenada,
- Ⓑ e uma segunda parte a_i, \dots, a_n ainda por ordenar ($i \in 1, \dots, n$).

Em cada passagem, o primeiro elemento da parte não ordenada da lista é trocado com o mínimo da parte não ordenada da lista.

Determinar o mínimo em cada passagem implica uma série de comparações entre elementos.
 $n-1$ passagens

23

Algoritmia (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

utad
Pedro Melo-Pinto 2022

revisão algoritmos básicos SELECTION SORT

A lista pode considerar-se dividida em duas partes:

- ① a primeira parte a_1, \dots, a_{i-1} que já se encontra ordenada,
- ② e uma segunda parte a_i, \dots, a_n ainda por ordenar ($i \in 1, \dots, n$).

Em cada passagem, o primeiro elemento da parte não ordenada da lista é trocado com o mínimo da parte não ordenada da lista.

| | |
|--|--|
| | |
| | passagem 01 comparações: n-1 trocas: 1 |
| | passagem 02 comparações: n-2 trocas: 1 |
| | passagem 03 comparações: n-3 trocas: 1 |
| | passagem 04 comparações: 2 trocas: 1 |
| | passagem 05 comparações: 1 trocas: 1 |

O número de passagens é fixo ($n-1$).
O número de comparações em cada passagem é $n-p$, em que p é o número de passagens já realizadas.
O número de trocas por passagem é fixo (1).

24

Algoritmia (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

utad
Pedro Melo-Pinto 2022

revisão algoritmos básicos SELECTION SORT

A lista pode considerar-se dividida em duas partes:

- ① a primeira parte a_1, \dots, a_{i-1} que já se encontra ordenada,
- ② e uma segunda parte a_i, \dots, a_n ainda por ordenar ($i \in 1, \dots, n$).

Em cada passagem, o primeiro elemento da parte não ordenada da lista é trocado com o mínimo da parte não ordenada da lista.

qualquer caso

$$\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \Rightarrow O(n^2)$$

comparações: $\sim \frac{1}{2} n^2$
trocas: $n - 1$

| | |
|--|--|
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

25

utad
Pedro Melo-Pinto 2022

revisão algoritmos básicos

SELECTION SORT

A lista pode considerar-se dividida em duas partes:

- (i) a primeira parte a_1, \dots, a_{i-1} que já se encontra ordenada,
- (ii) é uma segunda parte a_i, \dots, a_n ainda por ordenar ($i \in 1, \dots, n$).

Em cada passagem, o primeiro elemento da parte não ordenada da lista é trocado com o mínimo da parte não ordenada da lista.

Características

- Fácil implementação
- Complexidade invariante: $O(n^2)$
- Não é geralmente estável, mas pode ser implementado como tal
- *In-place* (necessita de um espaço extra de memória constante: $O(1)$)

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

26

utad
Pedro Melo-Pinto 2022

revisão algoritmos básicos

SELECTION SORT

A lista pode considerar-se dividida em duas partes:

- (i) a primeira parte a_1, \dots, a_{i-1} que já se encontra ordenada,
- (ii) é uma segunda parte a_i, \dots, a_n ainda por ordenar ($i \in 1, \dots, n$).

Em cada passagem, o primeiro elemento da parte não ordenada da lista é trocado com o mínimo da parte não ordenada da lista.

Variantes

- Variante bidireccional: *cocktail sort*
Neste caso, o algoritmo encontra o máximo e o mínimo em cada passagem.
Reduz o número de passagens por um factor de 2 (actuando nos ciclos), mas não diminui o número de comparações ou trocas.
- Variante estável: *stable sort*
Insere o mínimo na primeira posição (em vez de realizar a troca).

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

27

utad
Pedro Melo-Pinto 2022

revisão algoritmos básicos BUBBLE SORT

Seja a lista a ordenar (a_1, a_2, \dots, a_n) .

Utiliza como princípio básico a comparação em pares dos elementos da lista, trocando-os caso a sua ordem esteja errada.

Na 1^a passagem, o maior elemento “flutua” até à sua posição.

Na 2^a passagem, o segundo maior elemento “flutua” até à sua posição e assim sucessivamente.

O processo repete-se até não ocorrer qualquer troca, altura em que a lista está ordenada.

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

28

utad
Pedro Melo-Pinto 2022

revisão algoritmos básicos BUBBLE SORT

Seja a lista a ordenar (a_1, a_2, \dots, a_n) .

Percorre-se a lista da esquerda para a direita

Utiliza como princípio básico a comparação em pares dos elementos da lista, trocando-os caso a sua ordem esteja errada.

No fim da primeira passagem garante-se que o maior elemento “subiu” ou “flutuou” até ao fim da lista.

| | | |
|---------------------------|---------------------------|---------------------------|
| não ordenado | | não ordenado |
| | | operação |
| | | |
| [6, 2, 5, 1, 8, 7, -2, 7] | [2, 5, 1, 6, 7, 8, -2, 7] | [2, 5, 1, 6, 7, -2, 7, 8] |

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

29
utad
Pedro Melo-Pinto 2022

revisão algoritmos básicos BUBBLE SORT

Seja a lista a ordenar (a_1, a_2, \dots, a_n) .
Percorre-se a lista da esquerda para a direita.

No fim da primeira passagem garante-se que o maior elemento "subiu" ou "flutuou" até ao fim da lista.
O processo repete-se até não ocorrer qualquer troca, altura em que a lista está ordenada.

passagem 01 passagem 02 passagem 03 passagem 04 passagem 05 passagem 06 passagem 07

$\leq n-1$ passagens (dependendo da existência de trocas)

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

30
utad
Pedro Melo-Pinto 2022

revisão algoritmos básicos BUBBLE SORT

Seja a lista a ordenar (a_1, a_2, \dots, a_n) .
Percorre-se a lista da esquerda para a direita.

Utiliza como princípio básico a comparação em pares dos elementos da lista, trocando-os caso a sua ordem esteja errada.
No fim da primeira passagem garante-se que o maior elemento "subiu" ou "flutuou" até ao fim da lista.
O processo repete-se até não ocorrer qualquer troca, altura em que a lista está ordenada.

| | | |
|-------------------------|-------------|-------------------------------|
| [5 3 1 8 7 2] | | |
| [5 3 1 8 7 2] | passagem 01 | comparações: n-1 trocas: 4 |
| [3 1 5 7 2 8] | | |
| [3 1 5 7 2 8] | passagem 02 | comparações: n-2 trocas: 2 |
| [1 3 5 2 7 8] | | |
| [1 3 2 5 7 8] | passagem 03 | comparações: n-3 trocas: 1 |
| [1 3 2 5 7 8] | | |
| [1 2 3 5 7 8] | passagem 04 | comparações: 2 trocas: 1 |
| [1 2 3 5 7 8] | | |
| [1 2 3 5 7 8] | passagem 05 | comparações: 1 trocas: 0 |

O número de passagens é $\leq n-1$.

O número de comparações em cada passagem é $n-p$, em que p é o número de passagens já realizadas.

O número de trocas por passagem depende da ordenação relativa prévia.

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

31

utad
Pedro Melo-Pinto 2022

revisão algoritmos básicos
BUBBLE SORT

Seja a lista a ordenar (a_1, a_2, \dots, a_n) .
Percorre-se a lista da esquerda para a direita
Utiliza como princípio básico a comparação em pares dos elementos da lista, trocando-os caso a sua ordem esteja errada.
No fim da primeira passagem garante-se que o maior elemento "subiu" ou "flutuou" até ao fim da lista.
O processo repete-se até não ocorrer qualquer troca, altura em que a lista está ordenada.

qualquer caso ou caso-pior

$$\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \Rightarrow \mathbf{O(n^2)}$$

excepto quando a lista está ordenada ou quase ordenada

comparações: $\sim \frac{1}{2} n^2$
trocas: $\sim \frac{1}{2} n^2$

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

32

utad
Pedro Melo-Pinto 2022

revisão algoritmos básicos
BUBBLE SORT

Seja a lista a ordenar (a_1, a_2, \dots, a_n) .
Percorre-se a lista da esquerda para a direita
Utiliza como princípio básico a comparação em pares dos elementos da lista, trocando-os caso a sua ordem esteja errada.
No fim da primeira passagem garante-se que o maior elemento "subiu" ou "flutuou" até ao fim da lista.
O processo repete-se até não ocorrer qualquer troca, altura em que a lista está ordenada.

Características

- Dos três métodos é o único que consegue determinar quando parar se o conjunto estiver ordenado.
- A rapidez com que elementos muito desordenados atingem o seu lugar final depende do seu valor e posição:
Elementos grandes no inicio rapidamente "sobem" até ao seu lugar no final do conjunto ("rabbits")
Elementos pequenos no fim são extraordinariamente lentos a atingir a sua posição ("turtles")

The slide has an orange background. At the top left, it says "Algoritma (e Estruturas de Dados)" and "ALGORITMOS ORDENAÇÃO". At the top right, there's a logo for "utad" and the text "Pedro Melo-Pinto 2022". A small number "33" is in the top right corner. The main title "revisão algoritmos básicos" is centered, followed by "BUBBLE SORT". Below the title, the text describes the algorithm: "Seja a lista a ordenar (a_1, a_2, \dots, a_n) . Percorre-se a lista da esquerda para a direita. Utiliza como princípio básico a comparação em pares dos elementos da lista, trocando-os caso a sua ordem esteja errada. No fim da primeira passagem garante-se que o maior elemento "subiu" ou "flutuou" até ao fim da lista. O processo repete-se até não ocorrer qualquer troca, altura em que a lista está ordenada." A section titled "Características" lists the following points:

- Fácil implementação
- Complexidade $O(n^2)$, menos em listas (quase) ordenadas (cuja complexidade é melhor)
- Complexidade $O(n)$ no melhor caso, se implementado de modo a detectar que a lista se encontra ordenada (neste caso só perde para a *insertion sort* que apresenta um menor número de inversões)
- Estável (não altera a ordem relativa de elementos com chave igual)
- *In-place* (necessita de um espaço extra de memória constante: $O(1)$)

The slide has an orange background. At the top left, it says "Algoritma (e Estruturas de Dados)" and "ALGORITMOS ORDENAÇÃO". At the top right, there's a logo for "utad" and the text "Pedro Melo-Pinto 2022". A small number "34" is in the top right corner. The main title "revisão algoritmos básicos" is centered, followed by "BUBBLE SORT". Below the title, the text describes the algorithm: "Seja a lista a ordenar (a_1, a_2, \dots, a_n) . Percorre-se a lista da esquerda para a direita. Utiliza como princípio básico a comparação em pares dos elementos da lista, trocando-os caso a sua ordem esteja errada. No fim da primeira passagem garante-se que o maior elemento "subiu" ou "flutuou" até ao fim da lista. O processo repete-se até não ocorrer qualquer troca, altura em que a lista está ordenada." A section titled "Variantes" lists the following point:

- Variante bidireccional: *cocktail sort* ou *shaker sort*
O algoritmo da variante faz alternadamente uma passagem da esquerda para a direita (fazendo subir os elementos maiores) e uma passagem da direita para a esquerda (fazendo descer os elementos menores). É um pouco mais difícil do que a original, resolve o problema das "turtles", mas mantém complexidade $O(n^2)$ em geral.

35

utad
Pedro Melo-Pinto 2022

Algoritma (e Estruturas de Dados)

ALGORITMOS ORDENAÇÃO

Desempenho dos diferentes algoritmos

| Nome | Caso Médio | Caso Pior | Memória | Estável | Notas |
|---------------------|---|-----------------|-------------|---------|---|
| Bubble sort | — | $O(n^2)$ | $O(1)$ | Sim | |
| Cocktail sort | — | $O(n^2)$ | $O(1)$ | Sim | |
| Comb sort | $O(n \log n)$ | $O(n \log n)$ | $O(1)$ | Não | |
| Gnome sort | — | $O(n^2)$ | $O(1)$ | Sim | |
| Selection sort | $O(n^2)$ | $O(n^2)$ | $O(1)$ | Não | Pode ser implementado como estável |
| Insertion sort | $O(n + d)$ | $O(n^2)$ | $O(1)$ | Sim | d é o número de inversões, que é $O(n^2)$ |
| Shell sort | — | $O(n \log^2 n)$ | $O(1)$ | Não | |
| Binary tree sort | $O(n \log n)$ | $O(n \log n)$ | $O(n)$ | Sim | Quanto é utilizada uma árvore binária (auto)equilibrada |
| Library sort | $O(n \log n)$ | $O(n^2)$ | $O(n)$ | Sim | |
| Merge sort | $O(n \log n)$ | $O(n \log n)$ | $O(n)$ | Sim | |
| In-place merge sort | $O(n \log n)$ | $O(n \log n)$ | $O(1)$ | Não | |
| Heapsort | $O(n \log n)$ | $O(n \log n)$ | $O(1)$ | Não | |
| Smoothsort | — | $O(n \log n)$ | $O(1)$ | Não | |
| Quicksort | $O(n \log n)$ | $O(n^2)$ | $O(\log n)$ | Não | Variantes "ingênuas" utilizam espaço $O(n)$; Pode ter um pior caso $O(n \log n)$ se utilizada a mediana como pivot. |
| Introsort | $O(n \log n)$ | $O(n \log n)$ | $O(\log n)$ | Não | |
| Patience sorting | — | $O(n^2)$ | $O(n)$ | Não | |
| Strand sort | $O(n \log n)$ | $O(n^2)$ | $O(n)$ | Sim | |
| Radix sort | non-comparative integer sorting algorithm | | | | Ordena n inteiros na base k com, no máximo, d dígitos em $O(d(n + k))$. |

36

utad
Pedro Melo-Pinto 2022

Algoritma (e Estruturas de Dados)

ALGORITMOS ORDENAÇÃO

Desempenho dos diferentes algoritmos

| Número de Elementos (random) | Inteiros | | | Strings | | |
|---------------------------------|----------|----------|-----------|---------|----------|-----------|
| | Seleção | Inserção | Flutuação | Seleção | Inserção | Flutuação |
| 5000 | 64 | 40 | 394 | 179 | 83 | 554 |
| 10000 | 258 | 163 | 1591 | 782 | 365 | 2221 |
| 20000 | 1041 | 654 | 6359 | 3049 | 1543 | 10162 |
| | 1,0 | 1,0 | 1,0 | 1,0 | 1,0 | 1,0 |
| | 4,0 | 4,1 | 4,0 | 4,4 | 4,4 | 4,0 |
| | 16,3 | 16,4 | 16,1 | 17,0 | 18,6 | 18,3 |

| Número de Elementos (sorted) | Inteiros | | | Strings | | |
|---------------------------------|----------|----------|-----------|---------|----------|-----------|
| | Seleção | Inserção | Flutuação | Seleção | Inserção | Flutuação |
| 5000 | 66 | 0 | 0 | 273 | 0 | 0 |
| 10000 | 257 | 0 | 0 | 1090 | 0 | 0 |
| 20000 | 1050 | 0 | 0 | 4147 | 0 | 0 |
| 50000 | 0 | 0 | 0 | 3 | 2 | 2 |
| | 1,0 | | | 1,0 | | |
| | 3,9 | | | 4,0 | | |
| | 15,9 | | | 15,2 | | |

Algoritmia (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

37

algoritmos de ordenação

04 MERGE SORT 05 QUICK SORT 06 HEAP SORT 07 RADIX SORT

| Nome | Caso Médio | Caso Pior | Memória | Estável | Notas |
|----------------|---|-----------------|-------------|---------|---|
| Bubble sort | $O(n^2)$ | $O(n^2)$ | $O(1)$ | Sim | |
| Insertion sort | $O(n^2)$ | $O(n^2)$ | $O(1)$ | Não | |
| Selection sort | $O(n^2)$ | $O(n^2)$ | $O(1)$ | Não | |
| Shell sort | $O(n^2)$ | $O(n^2)$ | $O(1)$ | Não | |
| Deutsch sort | $O(n^2)$ | $O(n^2)$ | $O(1)$ | Não | Pode ser implementado como estável. |
| Implement sort | $O(n \cdot n!)$ | $O(n \cdot n!)$ | $O(1)$ | Sim | |
| Sort sort | $O(n \log n)$ | $O(n \log n)$ | $O(1)$ | Não | é o menor de inserções, cost é $O(n^2)$. |
| Bitonic sort | $O(n \log n)$ | $O(n \log n)$ | $O(1)$ | Não | Quando é utilizada para incrementar ou decrementar. |
| Radix sort | $O(1)$ | $O(1)$ | $O(1)$ | Não | |
| Merge sort | $O(n \log n)$ | $O(n \log n)$ | $O(n)$ | Sim | |
| Introsort | $O(n \log n)$ | $O(n \log n)$ | $O(1)$ | Não | |
| Heapsort | $O(n \log n)$ | $O(n \log n)$ | $O(1)$ | Não | |
| Quicksort | $O(n \log n)$ | $O(n^2)$ | $O(\log n)$ | Não | Variante "Inplace", consome espaço $O(1)$. Nota: O sorteio para quarto $O(n \log n)$ é feito utilizando a medida deslocada (pivot). |
| Introsort | $O(n \log n)$ | $O(n \log n)$ | $O(1)$ | Não | |
| Introsort | $O(n \log n)$ | $O(n \log n)$ | $O(1)$ | Não | |
| Introsort | $O(n \log n)$ | $O(n \log n)$ | $O(1)$ | Não | |
| Radix sort | non-comparative integer sorting algorithm | | | | Opera o intervalo numérico crescente, no máximo, 32 dígitos em $O(d \log n)$. |

Algoritmia (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

38

04 MERGE SORT
divisão-e-conquista

Utiliza o paradigma da divisão-e-conquista, fazendo a partição sucessiva em dois subconjuntos dos elementos a ordenar.

Seja a lista (a_1, a_2, \dots, a_n) .

Divide-se a lista desordenada em duas sublistas de tamanho igual.

Divide-se, recursivamente, cada uma das sublistas.

O processo deve cessar quando a sublista resultante tiver dimensão igual a um.

Unem-se sucessivamente cada duas sublistas numa lista ordenada.

O processo termina quando ficarmos com a lista completa ordenada.

Algoritmia (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

39

utad
Pedro Melo-Pinto 2022

04 MERGE SORT

divisão-e-conquista

Utiliza o paradigma da divisão-e-conquista, fazendo a partição sucessiva em dois subconjuntos dos elementos a ordenar, unindo sucessivamente pares de sublistas.

Em cada instante todas as sublistas devem estar ordenadas.

Este método baseia-se em dois princípios:

- Uma lista pequena leva menos passos até estar ordenada do que uma lista maior.
- A união de duas listas ordenadas na lista resultante ordenada leva menos passos do que ordenar esta lista.

Algoritmia (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

40

utad
Pedro Melo-Pinto 2022

04 MERGE SORT

divisão-e-conquista

Divide-se a lista desordenada em duas sublistas de tamanho igual

Divide-se, recursivamente, cada uma das sublistas.

O processo deve cessar quando a sublista resultante tiver dimensão igual a um.

exemplo: lista {6, -2, 5, 1, 8, 7, 2, 7}

| | | | | | | | |
|---|----|---|---|---|---|---|---|
| 6 | -2 | 5 | 1 | 8 | 7 | 2 | 7 |
| 6 | -2 | 5 | 1 | 8 | 7 | 2 | 7 |
| 6 | -2 | 5 | 1 | 8 | 7 | 2 | 7 |
| 6 | -2 | 5 | 1 | 8 | 7 | 2 | 7 |

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

04 MERGE SORT
divisão-e-conquista

41

utad
Pedro Melo-Pinto 2022

Divide-se a lista desordenada em duas sublistas de tamanho igual.
Divide-se, recursivamente, cada uma das sublistas.
O processo deve cessar quando a sublistas resultante tiver dimensão igual a um.
Unem-se sucessivamente cada duas sublistas numa lista ordenada.
O processo termina quando ficarmos com a lista completa ordenada.

conquista (*merging*)

exemplo: lista {6,-2,5,1,8,7,2,7}

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

04 MERGE SORT
divisão-e-conquista

42

utad
Pedro Melo-Pinto 2022

Faz a partição sucessiva em dois subconjuntos dos elementos a ordenar, unindo sucessivamente pares de sublistas.
Em cada instante todas as sublistas devem estar ordenadas.

Este método baseia-se em dois princípios:
Uma lista pequena leva menos passos até estar ordenada do que uma lista maior.
A união de duas listas ordenadas na lista resultante ordenada leva menos passos do que ordenar esta lista.

Algoritmo

| | |
|--|---|
| <pre>function mergesort pass in: ARRAY INT v, INT iLeft, iRight var: INT iMed</pre> | <pre>function merge pass in: ARRAY INT v, INT iLeft, iMed, iRight var: ARRAY INT aux, INT i, j, k</pre> |
| <pre>if iLeft < iRight iMed ← (iLeft + iRight) / 2 call: mergesort(v, iLeft, iMed) call: mergesort(v, iMed + 1, iRight) call: merge(v, iLeft, iMed, iRight) end if end function</pre> | <pre>iLeft repeat while i ≤ iMed aux[i] ← v[i] i ← i + 1 end repeat i ← i + 1 repeat while i ≤ iMed if v[i] ≤ v[i] v[i] ← aux[i] i ← i + 1 else v[i] ← aux[i] i ← i + 1 end if end repeat repeat while k < j v[k] ← aux[i] i ← i + 1 k ← k + 1 end repeat end function</pre> |
| <pre>merge(v, 0, 3, 7)</pre> | <pre>iMed -2 1 5 6 2 7 7 8</pre> |

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

04 MERGE SORT

divisão-e-conquista

Faz a partição sucessiva em dois subconjuntos dos elementos a ordenar, unindo sucessivamente pares de sublistas. Em cada instante todas as sublistas devem estar ordenadas.

Este método baseia-se em dois princípios:
 Uma lista pequena leva menos passos até estar ordenada do que uma lista maior.
 A união de duas listas ordenadas na lista resultante ordenada leva menos passos do que ordenar esta lista.

Algoritmo

```

function mergesort
    pass in: ARRAY INT v, INT iLeft, iRight
    var: INT iMed
    if iLeft < iRight
        iMed ← (iLeft + iRight) / 2
        call: mergesort(v, iLeft, iMed)
        call: mergesort(v, iMed + 1, iRight)
        call: merge(v, iLeft, iMed, iRight)
    end if
    end function
    merge(v, 0, 3, 7)

```

```

function merge
    pass in: ARRAY INT v, INT iLeft, iMed, iRight
    var: ARRAY INT aux, INT i, j, k
    i ← 0
    j ← iLeft
    repeat while i ≤ iMed
        aux[i] ← v[i]
        i ← i + 1
        j ← j + 1
    end repeat
    i ← iLeft
    repeat while k < j and j ≤ iRight
        if aux[i] ≤ aux[j]
            v[k] ← aux[i]
            i ← i + 1
        else
            v[k] ← aux[j]
            j ← j + 1
        end if
        k ← k + 1
    end repeat
    repeat while k < j
        v[k] ← aux[i]
        i ← i + 1
        k ← k + 1
    end repeat
    end function

```

copia a primeira sublistas (ordenada)

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

04 MERGE SORT

divisão-e-conquista

Faz a partição sucessiva em dois subconjuntos dos elementos a ordenar, unindo sucessivamente pares de sublistas. Em cada instante todas as sublistas devem estar ordenadas.

Este método baseia-se em dois princípios:
 Uma lista pequena leva menos passos até estar ordenada do que uma lista maior.
 A união de duas listas ordenadas na lista resultante ordenada leva menos passos do que ordenar esta lista.

Algoritmo

```

function mergesort
    pass in: ARRAY INT v, INT iLeft, iRight
    var: INT iMed
    if iLeft < iRight
        iMed ← (iLeft + iRight) / 2
        call: mergesort(v, iLeft, iMed)
        call: mergesort(v, iMed + 1, iRight)
        call: merge(v, iLeft, iMed, iRight)
    end if
    end function
    merge(v, 0, 3, 7)

```

```

function merge
    pass in: ARRAY INT v, INT iLeft, iMed, iRight
    var: ARRAY INT aux, INT i, j, k
    i ← 0
    j ← iLeft
    repeat while i ≤ iMed
        aux[i] ← v[i]
        i ← i + 1
        j ← j + 1
    end repeat
    i ← iLeft
    repeat while k < j and j ≤ iRight
        if v[i] ≤ aux[j]
            v[k] ← aux[i]
            i ← i + 1
        else
            v[k] ← aux[j]
            j ← j + 1
        end if
        k ← k + 1
    end repeat
    repeat while k < j
        v[k] ← aux[i]
        i ← i + 1
        k ← k + 1
    end repeat
    end function

```

reinicializa os índices de controlo

45

utad
Pedro Melo-Pinto 2022

04 MERGE SORT

divisão-e-conquista

Faz a partição sucessiva em dois subconjuntos dos elementos a ordenar, unindo sucessivamente pares de sublistas. Em cada instante todas as sublistas devem estar ordenadas.

Este método baseia-se em dois princípios:
 Uma lista pequena leva menos passos até estar ordenada do que uma lista maior.
 A união de duas listas ordenadas na lista resultante ordenada leva menos passos do que ordenar esta lista.

Algoritmo

```

function mergesort
    pass in: ARRAY INT v, INT iLeft, iRight
    var: INT iMed
    if iLeft < iRight
        iMed ← (iLeft + iRight) / 2
        call: mergesort(v, iLeft, iMed)
        call: mergesort(v, iMed + 1, iRight)
        call: merge(v, iLeft, iMed, iRight)
    end if
end function

function merge
    pass in: ARRAY INT v, INT iLeft, iMed, iRight
    var: ARRAY INT aux, INT i, j, k
    i ← 0
    j ← iLeft
    repeat while j ≤ iMed
        aux[i] ← v[j]
        i ← i + 1
        j ← j + 1
    end repeat
    i ← iLeft
    k ← iMed + 1
    repeat while k < iRight
        if aux[i] ≤ v[k]
            v[i] ← aux[i]
            i ← i + 1
        else
            v[i] ← v[k]
            i ← i + 1
            k ← k + 1
        end if
    end repeat
    repeat while k < iRight
        v[i] ← aux[i]
        i ← i + 1
        k ← k + 1
    end repeat
end function

```

actualiza o array v, de acordo com os valores de aux e v

iMed

-2 1 5 6 2 7 7 8

k j

-2 1 5 6 aux[]

i

46

utad
Pedro Melo-Pinto 2022

04 MERGE SORT

divisão-e-conquista

Faz a partição sucessiva em dois subconjuntos dos elementos a ordenar, unindo sucessivamente pares de sublistas. Em cada instante todas as sublistas devem estar ordenadas.

Este método baseia-se em dois princípios:
 Uma lista pequena leva menos passos até estar ordenada do que uma lista maior.
 A união de duas listas ordenadas na lista resultante ordenada leva menos passos do que ordenar esta lista.

Algoritmo

```

function mergesort
    pass in: ARRAY INT v, INT iLeft, iRight
    var: INT iMed
    if iLeft < iRight
        iMed ← (iLeft + iRight) / 2
        call: mergesort(v, iLeft, iMed)
        call: mergesort(v, iMed + 1, iRight)
        call: merge(v, iLeft, iMed, iRight)
    end if
end function

function merge
    pass in: ARRAY INT v, INT iLeft, iMed, iRight
    var: ARRAY INT aux, INT i, j, k
    i ← 0
    j ← iLeft
    repeat while j ≤ iMed
        aux[i] ← v[j]
        i ← i + 1
        j ← j + 1
    end repeat
    i ← iLeft
    k ← iMed + 1
    repeat while k < iRight
        if aux[i] ≤ v[k]
            v[i] ← aux[i]
            i ← i + 1
        else
            v[i] ← v[k]
            i ← i + 1
            k ← k + 1
        end if
    end repeat
    repeat while k < iRight
        v[i] ← aux[i]
        i ← i + 1
        k ← k + 1
    end repeat
end function

```

actualiza o array v, de acordo com os valores de aux e v

iMed

-2 1 2 5 6 7 7 8

k j

-2 1 5 6 aux[]

i

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

04 MERGE SORT

divisão-e-conquista

Faz a partição sucessiva em dois subconjuntos dos elementos a ordenar, unindo sucessivamente pares de sublistas. Em cada instante todas as sublistas devem estar ordenadas.

Este método baseia-se em dois princípios:
 Uma lista pequena leva menos passos até estar ordenada do que uma lista maior.
 A união de duas listas ordenadas na lista resultante ordenada leva menos passos do que ordenar esta lista.

Algoritmo

```

function mergesort
    pass in: ARRAY INT v, INT iLeft, iRight
    var: INT iMed
    if iLeft < iRight
        iMed ← (iLeft + iRight) / 2
        call: mergesort(v, iLeft, iMed)
        call: mergesort(v, iMed + 1, iRight)
        call: merge(v, iLeft, iMed, iRight)
    end if
    end function

    merge(v, 0, 3, 7)
        iMed
        -2 1 5 6 2 7 7 8

```

ordem bitônica

```

function merge
    pass in: ARRAY INT v, INT iLeft, iMed, iRight
    var: ARRAY INT aux, INT i, j, k
    i ← iLeft
    j ← iRight
    k ← iLeft
    repeat while i ≤ iMed
        aux[k] ← v[i]
        i ← i + 1
    end repeat
    repeat while j > iMed
        aux[k] ← v[j]
        j ← j - 1
    end repeat
    i ← iLeft
    j ← iRight
    k ← iLeft
    repeat while i ≤ j
        if aux[i] ≤ aux[j]
            v[k] ← aux[i]
            i ← i + 1
        else
            v[k] ← aux[j]
            j ← j - 1
        end if
        aux[k] ← v[k]
        k ← k + 1
    end repeat
    end function

```

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

04 MERGE SORT

divisão-e-conquista

Faz a partição sucessiva em dois subconjuntos dos elementos a ordenar, unindo sucessivamente pares de sublistas. Em cada instante todas as sublistas devem estar ordenadas.

Este método baseia-se em dois princípios:
 Uma lista pequena leva menos passos até estar ordenada do que uma lista maior.
 A união de duas listas ordenadas na lista resultante ordenada leva menos passos do que ordenar esta lista.

Algoritmo

```

function mergesort
    pass in: ARRAY INT v, INT iLeft, iRight
    var: INT iMed
    if iLeft < iRight
        iMed ← (iLeft + iRight) / 2
        call: mergesort(v, iLeft, iMed)
        call: mergesort(v, iMed + 1, iRight)
        call: merge(v, iLeft, iMed, iRight)
    end if
    end function

    merge(v, 0, 3, 7)
        iMed
        -2 1 5 6 2 7 7 8

```

ordem bitônica

```

function merge
    pass in: ARRAY INT v, INT iLeft, iMed, iRight
    var: ARRAY INT aux, INT i, j, k
    i ← iLeft
    j ← iRight
    k ← iLeft
    repeat while i ≤ iMed
        aux[k] ← v[i]
        i ← i + 1
    end repeat
    repeat while j > iMed
        aux[k] ← v[j]
        j ← j - 1
    end repeat
    i ← iLeft
    j ← iRight
    k ← iLeft
    repeat while i ≤ j
        if aux[i] ≤ aux[j]
            v[k] ← aux[i]
            i ← i + 1
        else
            v[k] ← aux[j]
            j ← j - 1
        end if
        aux[k] ← v[k]
        k ← k + 1
    end repeat
    end function

```

inicialização dos índices de controlo

aux[]

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

04 MERGE SORT

divisão-e-conquista

Faz a partição sucessiva em dois subconjuntos dos elementos a ordenar, unindo sucessivamente pares de sublistas. Em cada instante todas as sublistas devem estar ordenadas.

Este método baseia-se em dois princípios:
 Uma lista pequena leva menos passos até estar ordenada do que uma lista maior.
 A união de duas listas ordenadas na lista resultante ordenada leva menos passos do que ordenar esta lista.

Algoritmo

```

function mergesort
    pass in: ARRAY INT v, INT iLeft, iRight
    var: INT iMed
    if iLeft < iRight
        iMed ← (iLeft + iRight) / 2
        call: mergesort(v, iLeft, iMed)
        call: mergesort(v, iMed + 1, iRight)
    end if
    end function
    
```

merge(v, 0, 3, 7)

```

function merge
    pass in: ARRAY INT v, INT iLeft, iMed, iRight
    var: ARRAY INT aux, INT i, j, k
    i ← iLeft
    j ← iRight
    k ← iLeft
    repeat while i ≤ iMed
        aux[k] ← v[i]
        i ← i + 1
    end repeat
    repeat while j > iMed
        aux[k] ← v[j]
        j ← j - 1
    end repeat
    i ← iLeft
    j ← iRight
    k ← iLeft
    repeat while i ≤ j
        if aux[i] ≤ aux[j]
            v[k] ← aux[i]
            i ← i + 1
        else
            v[k] ← aux[j]
            j ← j - 1
        end if
        aux[k] ← v[i]
        i ← i + 1
        k ← k + 1
    end repeat
    
```

ordem bitônica B

copia a primeira sublista (ordenada)

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

04 MERGE SORT

divisão-e-conquista

Faz a partição sucessiva em dois subconjuntos dos elementos a ordenar, unindo sucessivamente pares de sublistas. Em cada instante todas as sublistas devem estar ordenadas.

Este método baseia-se em dois princípios:
 Uma lista pequena leva menos passos até estar ordenada do que uma lista maior.
 A união de duas listas ordenadas na lista resultante ordenada leva menos passos do que ordenar esta lista.

Algoritmo

```

function mergesort
    pass in: ARRAY INT v, INT iLeft, iRight
    var: INT iMed
    if iLeft < iRight
        iMed ← (iLeft + iRight) / 2
        call: mergesort(v, iLeft, iMed)
        call: mergesort(v, iMed + 1, iRight)
    end if
    end function
    
```

merge(v, 0, 3, 7)

```

function merge
    pass in: ARRAY INT v, INT iLeft, iMed, iRight
    var: ARRAY INT aux, INT i, j, k
    i ← iLeft
    j ← iRight
    k ← iLeft
    repeat while i ≤ iMed
        aux[k] ← v[i]
        i ← i + 1
    end repeat
    repeat while j > iMed
        aux[k] ← v[j]
        j ← j - 1
    end repeat
    i ← iLeft
    j ← iRight
    k ← iLeft
    repeat while i ≤ j
        if aux[i] ≤ aux[j]
            v[k] ← aux[i]
            i ← i + 1
        else
            v[k] ← aux[j]
            j ← j - 1
        end if
        aux[k] ← v[i]
        i ← i + 1
        k ← k + 1
    end repeat
    
```

ordem bitônica B

copia a segunda sublista (ordenada) pela ordem inversa

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

04 MERGE SORT

divisão-e-conquista

Faz a partição sucessiva em dois subconjuntos dos elementos a ordenar, unindo sucessivamente pares de sublistas. Em cada instante todas as sublistas devem estar ordenadas.

Este método baseia-se em dois princípios:
 Uma lista pequena leva menos passos até estar ordenada do que uma lista maior.
 A união de duas listas ordenadas na lista resultante ordenada leva menos passos do que ordenar esta lista.

Algoritmo

```

function mergesort
    pass in: ARRAY INT v, INT iLeft, iRight
    var: INT iMed
    if iLeft < iRight
        iMed ← (iLeft + iRight) / 2
        call: mergesort(v, iLeft, iMed)
        call: mergesort(v, iMed + 1, iRight)
        call: merge(v, iLeft, iMed, iRight)
    end if
end function

merge(v, 0, 3, 7)
    ordem bitônica
    function merge
        pass in: ARRAY INT v, INT iLeft, iMed, iRight
        var: ARRAY INT aux, INT i, j, k
        i ← iLeft
        j ← iRight
        k ← iLeft
        repeat while i ≤ iMed
            aux[k] ← v[i]
            i ← i + 1
        end repeat
        repeat while j > iMed
            aux[k] ← v[j]
            j ← j - 1
        end repeat
        i ← iLeft
        j ← iRight
        k ← iLeft
        repeat while i ≤ j
            if aux[i] ≤ aux[j]
                v[i] ← aux[i]
                i ← i + 1
            else
                v[i] ← aux[j]
                j ← j - 1
            end if
            aux[k] ← aux[i]
            k ← k + 1
        end repeat
end function

```

reinicialização dos índices de controlo

iMed

aux[]

ordem bitônica

reinicialização dos índices de controlo

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

04 MERGE SORT

divisão-e-conquista

Faz a partição sucessiva em dois subconjuntos dos elementos a ordenar, unindo sucessivamente pares de sublistas. Em cada instante todas as sublistas devem estar ordenadas.

Este método baseia-se em dois princípios:
 Uma lista pequena leva menos passos até estar ordenada do que uma lista maior.
 A união de duas listas ordenadas na lista resultante ordenada leva menos passos do que ordenar esta lista.

Algoritmo

```

function mergesort
    pass in: ARRAY INT v, INT iLeft, iRight
    var: INT iMed
    if iLeft < iRight
        iMed ← (iLeft + iRight) / 2
        call: mergesort(v, iLeft, iMed)
        call: mergesort(v, iMed + 1, iRight)
        call: merge(v, iLeft, iMed, iRight)
    end if
end function

merge(v, 0, 3, 7)
    ordem bitônica
    function merge
        pass in: ARRAY INT v, INT iLeft, iMed, iRight
        var: ARRAY INT aux, INT i, j, k
        i ← iLeft
        j ← iRight
        k ← iLeft
        repeat while i ≤ iMed
            aux[k] ← v[i]
            i ← i + 1
        end repeat
        repeat while j > iMed
            aux[k] ← v[j]
            j ← j - 1
        end repeat
        i ← iLeft
        j ← iRight
        k ← iLeft
        repeat while i ≤ j
            if aux[i] ≤ aux[j]
                v[i] ← aux[i]
                i ← i + 1
            else
                v[i] ← aux[j]
                j ← j - 1
            end if
            aux[k] ← aux[i]
            k ← k + 1
        end repeat
end function

```

actualiza o array v, de acordo com os valores de aux e v

iMed

aux[]

ordem bitônica

actualiza o array v, de acordo com os valores de aux e v

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

04 MERGE SORT

divisão-e-conquista

Faz a partição sucessiva em dois subconjuntos dos elementos a ordenar, unindo sucessivamente pares de sublistas. Em cada instante todas as sublistas devem estar ordenadas.

Este método baseia-se em dois princípios:
Uma lista pequena leva menos passos até estar ordenada do que uma lista maior.
A união de duas listas ordenadas na lista resultante ordenada leva menos passos do que ordenar esta lista.

Algoritmo

```

function mergesort
    pass in: ARRAY INT v, INT iLeft, iRight
    var: INT iMed
    if iLeft < iRight
        iMed  $\leftarrow$  (iLeft + iRight) / 2
        call: mergesort(v, iLeft, iMed)
        call: mergesort(v, iMed + 1, iRight)
    end if
    end function

    merge(v, 0, 3, 7)
        if iLeft < iRight
            iMed  $\leftarrow$  (iLeft + iRight) / 2
            call: mergesort(v, iLeft, iMed)
            call: mergesort(v, iMed + 1, iRight)
        end if
        end function

```

ordem bitônica B

```

function merge
    pass in: ARRAY INT v, INT iLeft, iMed, iRight
    var: ARRAY INT aux, INT i, j, k
    i  $\leftarrow$  iLeft
    j  $\leftarrow$  iRight
    k  $\leftarrow$  iLeft
    repeat while i  $\leq$  iMed
        aux[k]  $\leftarrow$  v[i]
        i  $\leftarrow$  i + 1
    end repeat
    repeat while j  $>$  iMed
        aux[k]  $\leftarrow$  v[j]
        j  $\leftarrow$  j - 1
    end repeat
    i  $\leftarrow$  iLeft
    j  $\leftarrow$  iRight
    k  $\leftarrow$  iLeft
    repeat while i  $\leq$  j
        if aux[i]  $\leq$  aux[j]
            v[i]  $\leftarrow$  aux[i]
            i  $\leftarrow$  i + 1
        else
            v[k]  $\leftarrow$  aux[j]
            j  $\leftarrow$  j - 1
        end if
        k  $\leftarrow$  k + 1
    end repeat
end function

```

actualiza o array v,
de acordo com os valores de aux e v

-2 1 2 5 6 7 7 8 aux[]

iMed
-2 1 2 5 6 7 7 8
k
i

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

04 MERGE SORT

divisão-e-conquista

Faz a partição sucessiva em dois subconjuntos dos elementos a ordenar, unindo sucessivamente pares de sublistas. Em cada instante todas as sublistas devem estar ordenadas.

Este método baseia-se em dois princípios:
Uma lista pequena leva menos passos até estar ordenada do que uma lista maior.
A união de duas listas ordenadas na lista resultante ordenada leva menos passos do que ordenar esta lista.

Análise Geral

Baseia-se no paradigma de divisão-e-conquista.

O processo de divisão-e-conquista tem, habitualmente, três passos. Neste caso:

- ➊ Divisão
Divide a lista L[p..r] de n elementos em duas sublistas de $n/2$ elementos, L[p..m] e L[m+1..r].
O cálculo do índice m faz parte desta parte do processo.
- ➋ Conquista
Ordena as duas sublistas L[p..m] e L[m+1..r] através de chamadas recursivas da função que implementa o método (mergesort).
- ➌ Reunião
Reúne (merge) as duas sublistas ordenadas.
A lista L[p..r] está ordenada.

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

55

utad
Pedro Melo-Pinto 2022

04 MERGE SORT

divisão-e-conquista

Faz a partição sucessiva em dois subconjuntos dos elementos a ordenar, unindo sucessivamente pares de sublistas.
Em cada instante todas as sublistas devem estar ordenadas.

Este método baseia-se em dois princípios:
Uma lista pequena leva menos passos até estar ordenada do que uma lista maior.
A união de duas listas ordenadas na lista resultante ordenada leva menos passos do que ordenar esta lista.

Análise Geral

Eficaz em geral.
Tempo esperado de execução : $\Theta(n \lg n)$.
Não efectua a ordenação *in place*: $O(n)$.
Estável.

Adequado para ordenação externa.
Não necessita de ter acesso aleatório.

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

56

utad
Pedro Melo-Pinto 2022

04 MERGE SORT

divisão-e-conquista

Faz a partição sucessiva em dois subconjuntos dos elementos a ordenar, unindo sucessivamente pares de sublistas.
Em cada instante todas as sublistas devem estar ordenadas.

Este método baseia-se em dois princípios:
Uma lista pequena leva menos passos até estar ordenada do que uma lista maior.
A união de duas listas ordenadas na lista resultante ordenada leva menos passos do que ordenar esta lista.

Análise da união de duas sublistas ordenadas (*merging*)

As duas sublistas têm sempre tamanho aproximadamente igual: $n/2$

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

04 MERGE SORT
divisão-e-conquista

Faz a partição sucessiva em dois subconjuntos dos elementos a ordenar, unindo sucessivamente pares de sublistas.
Em cada instante todas as sublistas devem estar ordenadas.

Este método baseia-se em dois princípios:
Uma lista pequena leva menos passos até estar ordenada do que uma lista maior.
A união de duas listas ordenadas na lista resultante ordenada leva menos passos do que ordenar esta lista.

Análise da união de duas sublistas ordenadas (merging)

A O caso-melhor, quanto às comparações, ocorre quando o último elemento da sub-lista da esquerda é menor ou igual que o 1º elemento da sublista da direita

B É efectuada uma comparação para cada um dos elementos da sublista da esquerda

```

function merge
    passim: ARRAY INT v, INT iLeft, iMed, iRight
    var:   ARRAY INT aux, INT i, j, k
    i < 0
    j < iLeft
    repeat while j < iMed
        aux[i] = v[i]
        i = i + 1
        j = j + 1
    endrepeat
    k = iLeft
    repeat while k < j and j < iRight
        if aux[k] <= v[j]
            v[k] = aux[i]
            i = i + 1
        else
            v[k] = aux[j]
            j = j + 1
        endif
        k = k + 1
    endrepeat
    repeat while k < j
        v[k] = aux[i]
        i = i + 1
        k = k + 1
    endrepeat
end function

```

comparações: $\sim n/2$

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

04 MERGE SORT
divisão-e-conquista

Faz a partição sucessiva em dois subconjuntos dos elementos a ordenar, unindo sucessivamente pares de sublistas.
Em cada instante todas as sublistas devem estar ordenadas.

Este método baseia-se em dois princípios:
Uma lista pequena leva menos passos até estar ordenada do que uma lista maior.
A união de duas listas ordenadas na lista resultante ordenada leva menos passos do que ordenar esta lista.

Análise da união de duas sublistas ordenadas (merging)

A O caso-pior, quanto às comparações, ocorre quando o 1º elemento da sub-lista da esquerda é maior do que o último elemento da sublista da direita

B É efectuada uma comparação para cada um dos elementos da sublista da esquerda

C É efectuada uma comparação para cada um dos elementos da lista auxiliar

```

function merge
    passim: ARRAY INT v, INT iLeft, iMed, iRight
    var:   ARRAY INT aux, INT i, j, k
    i < 0
    j < iLeft
    repeat while j < iMed
        aux[i] = v[i]
        i = i + 1
        j = j + 1
    endrepeat
    i < 0
    k < iLeft
    repeat while k < j and j < iRight
        if aux[k] <= v[j]
            v[k] = aux[i]
            i = i + 1
        else
            v[k] = aux[j]
            j = j + 1
        endif
        k = k + 1
    endrepeat
    repeat while k < j
        v[k] = aux[i]
        i = i + 1
        k = k + 1
    endrepeat
end function

```

comparações: $\sim n$

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

04 MERGE SORT
divisão-e-conquista

Faz a partição sucessiva em dois subconjuntos dos elementos a ordenar, unindo sucessivamente pares de sublistas. Em cada instante todas as sublistas devem estar ordenadas.

Este método baseia-se em dois princípios:
Uma lista pequena leva menos passos até estar ordenada do que uma lista maior.
A união de duas listas ordenadas na lista resultante ordenada leva menos passos do que ordenar esta lista.

Análise da união de duas sublistas ordenadas (merging)

A No caso geral (expectável), os elementos estão distribuídos aleatoriamente
B Será razoável admitir que o 2º ciclo while será executado menos que n vezes, ou que o último ciclo while será executado menos que $n/2$ vezes

```

function merge
    passim: ARRAY INT v, INT iLeft, iMed, iRight
    var:   ARRAY INT aux, INT i, j, k
    i < 0
    j < iLeft
    repeat while j <= iMed
        aux[i] = v[i]
        i = i + 1
        j = j + 1
    endrepeat
    k = iLeft
    repeat while k < i and j < iRight
        if aux[k] < v[j]
            v[k] = aux[i]
            i = i + 1
        else
            v[k] = aux[j]
            j = j + 1
        endif
        k = k + 1
    endrepeat
    repeat while k < i
        v[k] = aux[i]
        i = i + 1
        k = k + 1
    endrepeat
end function

```

comparações: $< n$

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

04 MERGE SORT
divisão-e-conquista

Faz a partição sucessiva em dois subconjuntos dos elementos a ordenar, unindo sucessivamente pares de sublistas. Em cada instante todas as sublistas devem estar ordenadas.

Este método baseia-se em dois princípios:
Uma lista pequena leva menos passos até estar ordenada do que uma lista maior.
A união de duas listas ordenadas na lista resultante ordenada leva menos passos do que ordenar esta lista.

Análise da união de duas sublistas ordenadas (merging)

A O caso-melhor, quanto aos movimentos, ocorre quando o último elemento da sublista da esquerda é menor ou igual que o 1º elemento da sublista da direita
B Na criação da lista auxiliar, é efectuado um movimento por cada um dos elementos da sublista da esquerda
C Na reorganização da lista, é efectuado um movimento por cada um dos elementos da sublista da esquerda

```

function merge
    passim: ARRAY INT v, INT iLeft, iMed, iRight
    var:   ARRAY INT aux, INT i, j, k
    i < 0
    j < iLeft
    repeat while j <= iMed
        aux[i] = v[i]
        i = i + 1
        j = j + 1
    endrepeat
    i = 0
    k = iLeft
    repeat while k < i and j < iRight
        if aux[i] <= v[j]
            v[k] = aux[i]
            i = i + 1
        else
            v[k] = aux[j]
            j = j + 1
        endif
        k = k + 1
    endrepeat
    repeat while k < i
        v[k] = aux[i]
        i = i + 1
        k = k + 1
    endrepeat
end function

```

comparações: $\sim n/2$
movimentos: $\sim 2 \times n/2$

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

04 MERGE SORT

divisão-e-conquista

Faz a partição sucessiva em dois subconjuntos dos elementos a ordenar, unindo sucessivamente pares de sublistas.
Em cada instante todas as sublistas devem estar ordenadas.

Este método baseia-se em dois princípios:
Uma lista pequena leva menos passos até estar ordenada do que uma lista maior.
A união de duas listas ordenadas na lista resultante ordenada leva menos passos do que ordenar esta lista.

Análise da união de duas sublistas ordenadas (merging)

A O caso-pior, quanto aos movimentos, ocorre quando o 1º elemento da sublista da esquerda é maior do que o último elemento da sublista da direita
B Na criação da lista auxiliar, é efectuado um movimento por cada um dos elementos da sublista da esquerda
Na reorganização da lista,
C É efectuado um movimento por cada um dos elementos da sublista da esquerda
D É efectuado um movimento por cada um dos elementos da lista auxiliar

```

function merge
    passim: ARRAY INT v, INT iLeft, iMed, iRight
    var:   ARRAY INT aux, INT i, k
    i < 0
    j < iLeft
    repeat while j < iMed
        aux[i] = v[i]
        i = i + 1
        j = j + 1
    endrepeat
    i = iLeft
    repeat while k < j and j < iRight
        if aux[i] > v[j]
            v[k] = aux[i]
            i = i + 1
        else
            v[k] = aux[j]
            j = j + 1
        endif
        k = k + 1
    endrepeat
    repeat while k < j
        v[k] = aux[i]
        i = i + 1
        k = k + 1
    endrepeat
end function

```

comparações: $\sim n$
movimentos: $\sim n/2 + n$

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

04 MERGE SORT

divisão-e-conquista

Faz a partição sucessiva em dois subconjuntos dos elementos a ordenar, unindo sucessivamente pares de sublistas.
Em cada instante todas as sublistas devem estar ordenadas.

Este método baseia-se em dois princípios:
Uma lista pequena leva menos passos até estar ordenada do que uma lista maior.
A união de duas listas ordenadas na lista resultante ordenada leva menos passos do que ordenar esta lista.

Análise da união de duas sublistas ordenadas (merging)

A No caso geral (expectável), os elementos estão distribuídos aleatoriamente
B Na criação da lista auxiliar, é efectuado um movimento por cada um dos elementos da sublista da esquerda
C Será razoável admitir que o 2º ciclo while será executado menos que n vezes, ou o último ciclo while será executado menos que $n/2$ vezes

```

function merge
    passim: ARRAY INT v, INT iLeft, iMed, iRight
    var:   ARRAY INT aux, INT i, k
    i < 0
    j < iLeft
    repeat while j < iMed
        aux[i] = v[i]
        i = i + 1
        j = j + 1
    endrepeat
    i = iLeft
    repeat while k < j and j < iRight
        if aux[i] > v[j]
            v[k] = aux[i]
            i = i + 1
        else
            v[k] = aux[j]
            j = j + 1
        endif
        k = k + 1
    endrepeat
    repeat while k < j
        v[k] = aux[i]
        i = i + 1
        k = k + 1
    endrepeat
end function

```

comparações: $< n$
movimentos: $< n/2 + n$

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

04 MERGE SORT
divisão-e-conquista

Análise das comparações na união de duas sublistas ordenadas (merging)
As duas sublistas têm tamanho aproximadamente igual

| algoritmo de merging | variante com dois arrays auxiliares | variante - ordem bitônica |
|---|---|---|
| <pre> function merge pass in ARRAY INT v, INT iLeft, iMed, iRight var: ARRAY INT aux, INT i,j,k i = 0 j = iLeft repeat while i <= iMed aux[i] = v[i] i = i + 1 end repeat i = 0 k = iLeft repeat while k < i and j < iRight if aux[k] <= v[j] v[k] = aux[i] i = i + 1 else v[k] = aux[j] j = j + 1 end if k = k + 1 end repeat repeat while k < i and j < iRight if aux[k] <= v[j] v[k] = aux[i] i = i + 1 else v[k] = aux[j] j = j + 1 end if k = k + 1 end repeat end function </pre> | <pre> function merge pass in ARRAY INT v, INT iLeft, iMed, iRight var: ARRAY INT aux1, aux2, INT i,j,k i = 0 j = iLeft repeat while i <= iMed aux1[i] = v[i] i = i + 1 end repeat i = 0 k = iLeft repeat while k < i and j < iRight if aux1[k] <= v[j] aux2[i] = aux1[k] i = i + 1 else aux2[i] = v[j] j = j + 1 end if k = k + 1 end repeat repeat while k < i and j < iRight if aux1[k] <= v[j] aux2[i] = aux1[k] i = i + 1 else aux2[i] = v[j] j = j + 1 end if k = k + 1 end repeat end function </pre> | <pre> function merge pass in ARRAY INT v, INT iLeft, iMed, iRight var: ARRAY INT aux, INT i,j,k i = iLeft j = iLeft repeat while i <= iMed aux[i] = v[i] i = i + 1 end repeat k = iLeft i = i + 1 repeat while i <= iRight aux[i] = v[i] i = i + 1 end repeat i = iLeft j = iLeft repeat while i > iMed aux[i] = v[i] i = i + 1 end repeat k = iLeft i = i + 1 repeat while j < iRight - iMed aux[i] = v[j] j = j + 1 i = i + 1 end repeat repeat while i <= iRight - iMed aux[i] = v[j] j = j + 1 i = i + 1 end repeat end function </pre> |
| caso-pior - comparações: $\sim n$ caso esperado - comparações: $< n$ caso-pior - movimentos: $\sim n/2 + n$ caso esperado - movimentos: $< n/2 + n$ | caso-pior - comparações: $n - 1$ caso esperado - comparações: $\sim n - 1$ caso-pior - movimentos: $2 \times n$ caso esperado - movimentos: $2 \times n$ | caso geral - comparações: n caso geral - movimentos: $2 \times n$ |
| Complexidade algorítmica: $\Theta(n)$ | | |

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

04 MERGE SORT
divisão-e-conquista

Faz a partição sucessiva em dois subconjuntos dos elementos a ordenar, unindo sucessivamente pares de sublistas. Em cada instante todas as sublistas devem estar ordenadas.

Este método baseia-se em dois princípios:
 Uma lista pequena leva menos passos até estar ordenada do que uma lista maior.
 A união de duas listas ordenadas na lista resultante ordenada leva menos passos do que ordenar esta lista.

Análise geral do desempenho

Baseia-se no paradigma de divisão-e-conquista.

O processo de divisão-e-conquista tem, habitualmente, três passos. Neste caso:

- ➊ **Divisão**
Este passo apenas calcula o meio da lista para proceder à divisão. O seu tempo de execução é constante: $D(n) = \Theta(1)$.
- ➋ **Conquista**
Resolve recursivamente dois subproblemas, cada um de tamanho $n/2$. $T(n) = 2T(n/2)$.
- ➌ **Reunião**
Como vimos, o processo de reunião executa em $R(n) = \Theta(n)$.

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

04 MERGE SORT

divisão-e-conquista

Faz a partição sucessiva em dois subconjuntos dos elementos a ordenar, unindo sucessivamente pares de sublistas.
Em cada instante todas as sublistas devem estar ordenadas.

Este método baseia-se em dois princípios:
Uma lista pequena leva menos passos até estar ordenada do que uma lista maior.
A união de duas listas ordenadas na lista resultante ordenada leva menos passos do que ordenar esta lista.

Análise do desempenho

O tempo total de execução do algoritmo vai ser a soma do tempo de duas chamadas recursivas dele próprio (para cada iteração/partição) mais o tempo de execução da reunião (*merging*).
A comparação e o cálculo do índice central levam tempo constante e são ignorados.

```

function mergesort
    pass in: ARRAY INT v, INT iLeft, iRight
    var: INT iMed
    if iLeft < iRight
        iMed = (iLeft + iRight) / 2
        call: mergesort(v, iLeft, iMed)
        call: mergesort(v, iMed + 1, iRight)
        call: merge(v, iLeft, iMed, iRight)
    end if
end function

```

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

04 MERGE SORT

divisão-e-conquista

Faz a partição sucessiva em dois subconjuntos dos elementos a ordenar, unindo sucessivamente pares de sublistas.
Em cada instante todas as sublistas devem estar ordenadas.

Este método baseia-se em dois princípios:
Uma lista pequena leva menos passos até estar ordenada do que uma lista maior.
A união de duas listas ordenadas na lista resultante ordenada leva menos passos do que ordenar esta lista.

Análise do desempenho

O tempo total de execução do algoritmo vai ser a soma do tempo de duas chamadas recursivas dele próprio (para cada iteração/partição) mais o tempo de execução da reunião (*merging*).
A comparação e o cálculo do índice central levam tempo constante e são ignorados.

Como vimos, a reunião tem um desempenho linear ($\Theta(n)$).

Todas as restantes instruções são executadas em tempo constante ($\Theta(1)$).

O desempenho final vai então depender do número de partições.

```

function merge
    pass in: ARRAY INT v, INT iLeft, iMed, iRight
    var: ARRAY INT aux, INT i, j, k
    i < iLeft
    j < iLeft
    repeat while j <= iMed
        aux[i] = v[j]
        i = i + 1
        j = j + 1
    end repeat
    k < iLeft
    repeat while k < i and j <= iRight
        if aux[k] < v[j]
            v[k] = aux[i]
            i = i + 1
        else
            v[k] = v[j]
            j = j + 1
        end if
        k = k + 1
    end repeat
    repeat while k < i and j <= iRight
        v[k] = aux[i]
        i = i + 1
        k = k + 1
    end repeat
    repeat while k < i
        v[k] = aux[i]
        i = i + 1
        k = k + 1
    end repeat
end function

```

Algoritmia (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

67 utad Pedro Melo-Pinto 2022

04 MERGE SORT

divisão-e-conquista

Faz a partição sucessiva em dois subconjuntos dos elementos a ordenar, unindo sucessivamente pares de sublistas.
Em cada instante todas as sublistas devem estar ordenadas.

Este método baseia-se em dois princípios:
Uma lista pequena leva menos passos até estar ordenada do que uma lista maior.
A união de duas listas ordenadas na lista resultante ordenada leva menos passos do que ordenar esta lista.

Análise do desempenho

O tempo de execução em cada iteração vai ser a soma do tempo das duas chamadas recursivas mais o tempo da reunião.
A reunião tem um desempenho linear, $\Theta(n)$.
O desempenho final vai depender do número de partições.

caso geral

www.tutorialspoint.org

Algoritmia (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

68 utad Pedro Melo-Pinto 2022

04 MERGE SORT

divisão-e-conquista

Faz a partição sucessiva em dois subconjuntos dos elementos a ordenar, unindo sucessivamente pares de sublistas.
Em cada instante todas as sublistas devem estar ordenadas.

Este método baseia-se em dois princípios:
Uma lista pequena leva menos passos até estar ordenada do que uma lista maior.
A união de duas listas ordenadas na lista resultante ordenada leva menos passos do que ordenar esta lista.

Análise do desempenho

O tempo de execução em cada iteração vai ser a soma do tempo das duas chamadas recursivas mais o tempo da reunião.
A reunião tem um desempenho linear, $\Theta(n)$.
O desempenho final vai depender do número de partições.

caso geral

As partições são sempre equilibradas: $n = 2^k$ ou seja, $k = \log_2 n$.

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= 2(2T(n/4) + \Theta(n/2)) + \Theta(n) = 2^2 T(n/4) + 2\Theta(n) \\ &\dots \\ &= 2^k T(n/2^k) + k\Theta(n) \end{aligned}$$

como $n = 2^k \rightarrow k = \log_2 n$

$$\begin{aligned} &= n \times T(1) + \Theta(n) \log_2 n \\ &= \Theta(n \log_2 n) \end{aligned}$$

```

function merge
    pass in: ARRAY INT v[], INT iLeft, INT iMed, INT iRight
    var: ARRAY INT aux, INT i, j, k
    i ← 0
    j ← iLeft
    repeat while j ≤ iMed
        aux[i] ← v[j]
        i ← i + 1
        j ← j + 1
    end repeat
    i ← 0
    k ← iLeft
    repeat while k < i and j ≤ iRight
        if v[i] ≤ v[j]
            v[k] ← aux[i]
            i ← i + 1
        else
            v[k] ← aux[j]
            j ← j + 1
        end if
        k ← k + 1
    end repeat
    repeat while k < j
        v[k] ← aux[i]
        i ← i + 1
        k ← k + 1
    end repeat
end function

```

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

04 MERGE SORT

divisão-e-conquista

Faz a partição sucessiva em dois subconjuntos dos elementos a ordenar, unindo sucessivamente pares de sublistas.
Em cada instante todas as sublistas devem estar ordenadas.

Este método baseia-se em dois princípios:
Uma lista pequena leva menos passos até estar ordenada do que uma lista maior.
A união de duas listas ordenadas na lista resultante ordenada leva menos passos do que ordenar esta lista.

Análise do desempenho

O tempo de execução em cada iteração vai ser a soma do tempo das duas chamadas recursivas mais o tempo da reunião.
A reunião tem um desempenho linear: $\Theta(n)$.
O desempenho final vai depender do número de partições.

caso geral (análise das comparações)

$$C_{\max}(n) = n-1 + C(\lceil n/2 \rceil) + C(\lfloor n/2 \rfloor), C(1) = 0$$

Se n é uma potência de 2, como as partições são equilibradas: $n = 2^k$ ou seja, $k = \log_2 n$, e podemos dividir sempre a lista em duas partes iguais:

$$\begin{aligned} C_{\max}(n) &= n - 1 + 2C(n/2) \\ &= n - 1 + 2(n/2 - 1 + 2C(n/2^2)) = 2n - (1 + 2) + 2^2 C(n/2^2) \\ &\dots \\ &= kn - (1 + 2 + 2^2 + \dots + 2^{k-1}) + 2^k C(n/2^k) \\ &= n \log_2 n - n + 1 \end{aligned}$$

```

function merge
pass in: ARRAY INT v, INT iLeft, iMed, iRight
var:   ARRAY INT aux, INT i, j, k
i < 0
j < iLeft
repeat while j <= iMed
    aux[j] = v[j]
    i < i+1
    j < j+1
end repeat
i < 0
k < iLeft
repeat while k < j and j < iRight
    if aux[i] <= v[j]
        v[i] = aux[i]
        i < i+1
    else
        v[k] = aux[j]
        j < j+1
    end if
    k < k+1
end repeat
repeat while k < j
    v[k] = aux[i]
    i < i+1
    k < k+1
end repeat
end function

```

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

04 MERGE SORT

divisão-e-conquista

Faz a partição sucessiva em dois subconjuntos dos elementos a ordenar, unindo sucessivamente pares de sublistas.
Em cada instante todas as sublistas devem estar ordenadas.

Este método baseia-se em dois princípios:
Uma lista pequena leva menos passos até estar ordenada do que uma lista maior.
A união de duas listas ordenadas na lista resultante ordenada leva menos passos do que ordenar esta lista.

Análise do desempenho

O tempo de execução em cada iteração vai ser a soma do tempo das duas chamadas recursivas mais o tempo da reunião.
A reunião tem um desempenho linear: $\Theta(n)$.
O desempenho final vai depender do número de partições.

caso geral (análise das comparações)

Se n não for uma potência de 2, a certa altura não poderemos dividir a lista exactamente ao meio.
Neste caso, $C(n)$ será ligeiramente maior.

$$C_{\max}(n) = n-1 + C(\lceil n/2 \rceil) + C(\lfloor n/2 \rfloor), C(1) = 0$$

A aproximação feita anteriormente (substituição de $\lceil n/2 \rceil$ e $\lfloor n/2 \rfloor$ por $n/2$) é acertada nos exemplos:

| | | |
|----------------------------|-----------------------------|---------------------------------|
| $n = 3$ (pior caso) | $n = 975$ | $n = 8329$ |
| $C(n) = 3$ e | $C(n) = 8727$ e | $C(n) = 100223$ e |
| $n \log_2 n - n + 1 = 2.8$ | $n \log_2 n - n + 1 = 8707$ | $n \log_2 n - n + 1 = 100148.3$ |

Logo, podemos dizer que $C_{\max}(n) \approx n \log_2 n - n + 1$

```

function merge
pass in: ARRAY INT v, INT iLeft, iMed, iRight
var:   ARRAY INT aux, INT i, j, k
i < 0
j < iLeft
repeat while j <= iMed
    aux[j] = v[j]
    i < i+1
    j < j+1
end repeat
i < 0
k < iLeft
repeat while k < j and j < iRight
    if aux[i] <= v[j]
        v[i] = aux[i]
        i < i+1
    else
        v[k] = aux[j]
        j < j+1
    end if
    k < k+1
end repeat
repeat while k < j
    v[k] = aux[i]
    i < i+1
    k < k+1
end repeat
end function

```

71

utad
Pedro Melo-Pinto 2022

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

04 MERGE SORT

divisão-e-conquista

Faz a partição sucessiva em dois subconjuntos dos elementos a ordenar, unindo sucessivamente pares de sublistas.
Em cada instante todas as sublistas devem estar ordenadas.

Este método baseia-se em dois princípios:
Uma lista pequena leva menos passos até estar ordenada do que uma lista maior.
A união de duas listas ordenadas na lista resultante ordenada leva menos passos do que ordenar esta lista.

Análise do desempenho

O tempo de execução em cada iteração vai ser a soma da tempo das duas chamadas recursivas mais o tempo da reunião.
A reunião tem um desempenho linear: $\Theta(n)$.
O desempenho final vai depender do número de partições.

caso geral (análise das comparações)

$C_{\max}(n) \sim n \log_2 n - n + 1$

$C_{\text{med}}(n)$ é ligeiramente inferior a $C_{\max}(n)$

```

function merge
    pass in: ARRAY INT v, INT iLeft, iMed, iRight
    var:   ARRAY INT aux, INT i, j, k
    i <= 0
    j <= iLeft
    repeat while i <= iMed
        aux[i] = v[j]
        i <= i + 1
        j <= j + 1
    end repeat
    i <= 0
    k <= iLeft
    repeat while k < j and j < iRight
        if aux[i] > v[j]
            i <= i + 1
        else
            j <= j + 1
        end if
        k <= k + 1
    end repeat
    repeat while k < j
        v[k] = aux[i]
        i <= i + 1
        k <= k + 1
    end repeat
end function

```

72

utad
Pedro Melo-Pinto 2022

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

04 MERGE SORT

divisão-e-conquista

| Número de Elementos (random) | Inteiros | | | Strings | | |
|---------------------------------|-------------|----------|-------|-------------|----------|-------|
| | Sel. & Ord. | Inserção | Fusão | Sel. & Ord. | Inserção | Fusão |
| 1000 | 64 | 4.0 | 354 | 179 | 8.3 | 854 |
| 10000 | 256 | 16.3 | 1591 | 762 | 36.5 | 2224 |
| 100000 | 1024 | 65.6 | 6359 | 3049 | 154.9 | 10162 |
| 1000000 | 4096 | 1.1 | 1.1 | 1.1 | 1.1 | 1.1 |
| 10000000 | 16384 | 4.1 | 4.0 | 4.4 | 4.4 | 4.0 |
| 100000000 | 65536 | 16.4 | 16.1 | 17.0 | 18.6 | 18.3 |

| Número de Elementos (sorted) | Inteiros | | | Strings | | |
|---------------------------------|-------------|----------|-------|-------------|----------|-------|
| | Sel. & Ord. | Inserção | Fusão | Sel. & Ord. | Inserção | Fusão |
| 1000 | 66 | 0 | 0 | 273 | 0 | 0 |
| 10000 | 257 | 0 | 0 | 1090 | 0 | 0 |
| 100000 | 1050 | 0 | 0 | 4147 | 0 | 0 |
| 1000000 | 0 | 0 | 0 | 0 | 3 | 2 |
| 10000000 | 1.0 | — | — | 1.0 | — | — |
| 100000000 | 15.9 | — | — | 15.2 | — | — |

| Número de Elementos (random) | Inteiros | | | | | | Strings | | | | | |
|----------------------------------|----------|-------|------|-------|-------|------|---------|-------|------|-------|-------|------|
| | Quick | Merge | Heap | Quick | Merge | Heap | Quick | Merge | Heap | Quick | Merge | Heap |
| 20000 | 8 | | | | | | 18 | | | | | |
| 50000 | 21 | | | | | | 57 | | | | | |
| 100000 | 45 | | | | | | 137 | | | | | |
| evolução de $n \times \log_2(n)$ | 2,73 | 2,6 | | | | | 3,2 | | | | | |
| | 5,81 | 5,6 | | | | | 7,6 | | | | | |

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

73

utad
Pedro Melo-Pinto 2022

05 QUICK SORT

divisão-e-conquista

Utiliza o paradigma da divisão-e-conquista, fazendo a partição sucessiva em dois sub-conjuntos dos elementos a ordenar.

Seja a lista (a_1, a_2, \dots, a_n) .

Escolher um qualquer elemento da lista designado por *pivot*.

Posicionar todos os elementos da lista menores do que o *pivot* antes dele.

Posicionar todos os elementos da lista maiores do que o *pivot* depois dele.

Aplicar recursivamente o método às sublistas dos lados esquerdo e direito do *pivot*.

O processo termina quando a sublista resultante tiver dimensão menor ou igual a um.

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

74

utad
Pedro Melo-Pinto 2022

05 QUICK SORT

divisão-e-conquista

Utiliza como princípio a partição sucessiva em dois subconjuntos.

exemplo: lista {6, -2, 5, 1, 8, 7, 2, 7}

The diagram shows the step-by-step execution of the Quick Sort algorithm on the array [6, -2, 5, 1, 8, 7, 2, 7]. The pivot element is 6, which is highlighted in red. The array is partitioned into two halves: elements less than 6 on the left and elements greater than or equal to 6 on the right. The process is shown step-by-step, with arrows indicating the movement of elements during partitioning.

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

05 QUICK SORT
divisão-e-conquista

Utiliza como princípio a partição sucessiva em dois subconjuntos:
 Escolher um qualquer elemento da lista designado por *pivot*.
 Posicionar todos os elementos da lista menores do que o *pivot* antes dele.
 Posicionar todos os elementos da lista maiores do que o *pivot* depois dele.
 Aplicar recursivamente o método às sublistas dos lados esquerdo e direito do *pivot*.

Algoritmo

```

function quicksort
    pass in: ARRAY INT v, INT iLeft, iRight
    var: INT iPivot
    if iLeft < iRight
        iPivot ← call: partition(v, iLeft, iRight)
        call: quicksort(v, iLeft, iPivot - 1)
        call: quicksort(v, iPivot + 1, iRight)
    end if
    end function

```

```

function partition
    pass in: ARRAY INT v, INT iLeft, iRight
    var: INT pivot, l, r
    l ← iLeft + 1
    r ← iRight
    pivot ← v[iLeft]
    repeat while l < r
        repeat while v[l] ≤ pivot and l < r
            l ← l + 1
        end repeat
        repeat while v[r] > pivot
            r ← r - 1
        end repeat
        if l < r
            call: swap(v[l], v[r])
        end if
    end repeat
    v[iLeft] ← v[r]
    v[r] ← pivot
    pass out: r
    end function

```

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

05 QUICK SORT
divisão-e-conquista

Utiliza como princípio a partição sucessiva em dois subconjuntos:
 Escolher um qualquer elemento da lista designado por *pivot*.
 Posicionar todos os elementos da lista menores do que o *pivot* antes dele.
 Posicionar todos os elementos da lista maiores do que o *pivot* depois dele.
 Aplicar recursivamente o método às sublistas dos lados esquerdo e direito do *pivot*.

Algoritmo

```

function quicksort
    pass in: ARRAY INT v, INT iLeft, iRight
    var: INT iPivot
    if iLeft < iRight
        iPivot ← call: partition(v, iLeft, iRight)
        call: quicksort(v, iLeft, iPivot - 1)
        call: quicksort(v, iPivot + 1, iRight)
    end if
    end function

```

partition(v, 0, 7)

partition(v, 0, 7)

pivot

| | | | | | | | |
|---|----|---|---|---|---|---|---|
| 6 | -2 | 5 | 1 | 8 | 7 | 2 | 7 |
| | | | ↑ | | | | ↑ |
| | | | l | | | | r |

function partition

```

    pass in: ARRAY INT v, INT iLeft, iRight
    var: INT pivot, l, r
    l ← iLeft + 1
    r ← iRight
    pivot ← v[iLeft]
    repeat while l < r
        repeat while v[l] ≤ pivot and l < r
            l ← l + 1
        end repeat
        repeat while v[r] > pivot
            r ← r - 1
        end repeat
        if l < r
            call: swap(v[l], v[r])
        end if
    end repeat
    v[iLeft] ← v[r]
    v[r] ← pivot
    pass out: r
    end function

```

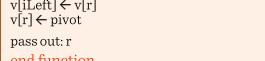
inicialização

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

05 QUICK SORT
divisão-e-conquista

Utiliza como princípio a partição sucessiva em dois subconjuntos:
 Escolher um qualquer elemento da lista designado por *pivot*.
 Posicionar todos os elementos da lista menores do que o *pivot* antes dele.
 Posicionar todos os elementos da lista maiores do que o *pivot* depois dele.
 Aplicar recursivamente o método às sublistas dos lados esquerdo e direito do *pivot*.

Algoritmo

| | |
|--|---|
| <pre>function quicksort pass in: ARRAY INT v, INT iLeft, iRight var: INT iPivot if iLeft < iRight iPivot ← call: partition(v, iLeft, iRight) call: quicksort(v, iLeft, iPivot - 1) call: quicksort(v, iPivot + 1, iRight) end if end function</pre> | <pre>function partition pass in: ARRAY INT v, INT iLeft, iRight var: INT pivot, l, r l ← iLeft + 1 r ← iRight pivot ← v[iLeft] repeat while l < r repeat while v[l] ≤ pivot and l < r l ← l + 1 end repeat repeat while v[r] > pivot r ← r - 1 end repeat if l < r call: swap(v[l], v[r]) end if end repeat v[iLeft] ← v[r] v[r] ← pivot pass out: r end function</pre> |
| partition(v, 0, 7) | partindo da esquerda (<i>l</i>), encontra o primeiro elemento > pivot |
|  |  |

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

05 QUICK SORT
divisão-e-conquista

Utiliza como princípio a partição sucessiva em dois subconjuntos:
 Escolher um qualquer elemento da lista designado por *pivot*.
 Posicionar todos os elementos da lista menores do que o *pivot* antes dele.
 Posicionar todos os elementos da lista maiores do que o *pivot* depois dele.
 Aplicar recursivamente o método às sublistas dos lados esquerdo e direito do *pivot*.

Algoritmo

| | |
|--|---|
| <pre>function quicksort pass in: ARRAY INT v, INT iLeft, iRight var: INT iPivot if iLeft < iRight iPivot ← call: partition(v, iLeft, iRight) call: quicksort(v, iLeft, iPivot - 1) call: quicksort(v, iPivot + 1, iRight) end if end function</pre> | <pre>function partition pass in: ARRAY INT v, INT iLeft, iRight var: INT pivot, l, r l ← iLeft + 1 r ← iRight pivot ← v[iLeft] repeat while l < r repeat while v[l] ≤ pivot and l < r l ← l + 1 end repeat repeat while v[r] > pivot r ← r - 1 end repeat if l < r call: swap(v[l], v[r]) end if end repeat v[iLeft] ← v[r] v[r] ← pivot pass out: r end function</pre> |
| partition(v, 0, 7) | partindo da direita (<i>r</i>), encontra o primeiro elemento < pivot |
|  |  |

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

05 QUICK SORT
divisão-e-conquista

Utiliza como princípio a partição sucessiva em dois subconjuntos:
 Escolher um qualquer elemento da lista designado por *pivot*.
 Posicionar todos os elementos da lista menores do que o *pivot* antes dele.
 Posicionar todos os elementos da lista maiores do que o *pivot* depois dele.
 Aplicar recursivamente o método às sublistas dos lados esquerdo e direito do *pivot*.

Algoritmo

| | |
|--|---|
| <pre>function quicksort pass in: ARRAY INT v, INT iLeft, iRight var: INT iPivot if iLeft < iRight iPivot ← call: partition(v, iLeft, iRight) call: quicksort(v, iLeft, iPivot - 1) call: quicksort(v, iPivot + 1, iRight) end if end function</pre> | <pre>function partition pass in: ARRAY INT v, INT iLeft, iRight var: INT pivot, l, r l ← iLeft + 1 r ← iRight pivot ← v[iLeft] repeat while l < r repeat while v[l] ≤ pivot and l < r l ← l + 1 end repeat repeat while v[r] > pivot r ← r - 1 end repeat if l < r call: swap(v[l], v[r]) end if end repeat v[iLeft] ← v[r] v[r] ← pivot pass out: r end function</pre> |
| partition(v, 0, 7) | <i>troca: estes elementos estão fora do seu lado</i> |
| pivot  | |

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

05 QUICK SORT
divisão-e-conquista

Utiliza como princípio a partição sucessiva em dois subconjuntos:
 Escolher um qualquer elemento da lista designado por *pivot*.
 Posicionar todos os elementos da lista menores do que o *pivot* antes dele.
 Posicionar todos os elementos da lista maiores do que o *pivot* depois dele.
 Aplicar recursivamente o método às sublistas dos lados esquerdo e direito do *pivot*.

Algoritmo

| | |
|--|---|
| <pre>function quicksort pass in: ARRAY INT v, INT iLeft, iRight var: INT iPivot if iLeft < iRight iPivot ← call: partition(v, iLeft, iRight) call: quicksort(v, iLeft, iPivot - 1) call: quicksort(v, iPivot + 1, iRight) end if end function</pre> | <pre>function partition pass in: ARRAY INT v, INT iLeft, iRight var: INT pivot, l, r l ← iLeft + 1 r ← iRight pivot ← v[iLeft] repeat while l < r repeat while v[l] ≤ pivot and l < r l ← l + 1 end repeat repeat while v[r] > pivot r ← r - 1 end repeat if l < r call: swap(v[l], v[r]) end if end repeat v[iLeft] ← v[r] v[r] ← pivot pass out: r end function</pre> |
| partition(v, 0, 7) | <i>troca: estes elementos estão fora do seu lado</i> |
| pivot  | |

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

81
utad
Pedro Melo-Pinto 2022

05 QUICK SORT

divisão-e-conquista

Utiliza como princípio a partição sucessiva em dois subconjuntos:

Escolher um qualquer elemento da lista designado por *pivot*.
 Posicionar todos os elementos da lista menores do que o *pivot* antes dele.
 Posicionar todos os elementos da lista maiores do que o *pivot* depois dele.
 Aplicar recursivamente o método às sublistas dos lados esquerdo e direito do *pivot*.

Algoritmo

```

function quicksort
  pass in: ARRAY INT v, INT iLeft, iRight
  var: INT iPivot
  if iLeft < iRight
    iPivot ← call: partition(v, iLeft, iRight)
    call: quicksort(v, iLeft, iPivot - 1)
    call: quicksort(v, iPivot + 1, iRight)
  end if
end function

partition(v, 0, 7)

pivot
[ 6 | -2 | 5 | 1 | 2 | 7 | 8 | 7 ]
  r | l

```

```

function partition
  pass in: ARRAY INT v, INT iLeft, iRight
  var: INT pivot, l, r
  l ← iLeft + 1
  r ← iRight
  pivot ← v[iLeft]
  repeat while l < r
    repeat while v[l] ≤ pivot and l < r
      l ← l + 1
    end repeat
    repeat while v[r] > pivot
      r ← r - 1
    end repeat
    if l < r
      call: swap(v[l], v[r])
    end if
  end repeat
  v[iLeft] ← v[r]
  v[r] ← pivot
  pass out: r
end function

```

fim do ciclo

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

82
utad
Pedro Melo-Pinto 2022

05 QUICK SORT

divisão-e-conquista

Utiliza como princípio a partição sucessiva em dois subconjuntos:

Escolher um qualquer elemento da lista designado por *pivot*.
 Posicionar todos os elementos da lista menores do que o *pivot* antes dele.
 Posicionar todos os elementos da lista maiores do que o *pivot* depois dele.
 Aplicar recursivamente o método às sublistas dos lados esquerdo e direito do *pivot*.

Algoritmo

```

function quicksort
  pass in: ARRAY INT v, INT iLeft, iRight
  var: INT iPivot
  if iLeft < iRight
    iPivot ← call: partition(v, iLeft, iRight)
    call: quicksort(v, iLeft, iPivot - 1)
    call: quicksort(v, iPivot + 1, iRight)
  end if
end function

partition(v, 0, 7)

pivot
[ 2 | -2 | 5 | 1 | 6 | 7 | 8 | 7 ]
  r | l

```

```

function partition
  pass in: ARRAY INT v, INT iLeft, iRight
  var: INT pivot, l, r
  l ← iLeft + 1
  r ← iRight
  pivot ← v[iLeft]
  repeat while l < r
    repeat while v[l] ≤ pivot and l < r
      l ← l + 1
    end repeat
    repeat while v[r] > pivot
      r ← r - 1
    end repeat
    if l < r
      call: swap(v[l], v[r])
    end if
  end repeat
  v[iLeft] ← v[r]
  v[r] ← pivot
  pass out: r
end function

```

repositionamento do pivot
por troca com o elemento na posição r

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

83

05 QUICK SORT
divisão-e-conquista

Utiliza como princípio a partição sucessiva em dois subconjuntos:
 Escolher um qualquer elemento da lista designado por *pivot*.
 Posicionar todos os elementos da lista menores do que o *pivot* antes dele.
 Posicionar todos os elementos da lista maiores do que o *pivot* depois dele.
 Aplicar recursivamente o método às sublistas dos lados esquerdo e direito do *pivot*.

Análise Geral

Baseia-se no paradigma de divisão-e-conquista.
 O processo de divisão-e-conquista tem, habitualmente, três passos. Neste caso:

- ➊ Divisão
Divide a lista $L[p..r]$ em duas sublistas $L[p..k-1]$ e $L[k+1..r]$ (vazias ou não), em que cada elemento de $L[p..k-1]$ é menor ou igual a $L[k]$ e cada elemento de $L[k+1..r]$ é maior que $L[k]$. O cálculo do índice k faz parte desta parte do processo.
- ➋ Conquista
Ordena as duas sublistas $L[p..k-1]$ e $L[k+1..r]$ através de chamadas recursivas da função que implementa o método (*quicksort*).
- ➌ Reunião
Uma vez que as duas sublistas são ordenadas *in place* não é necessária efectuar a sua reunião. A lista $L[p..r]$ está ordenada.

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

84

05 QUICK SORT
divisão-e-conquista

Utiliza como princípio a partição sucessiva em dois subconjuntos:
 Escolher um qualquer elemento da lista designado por *pivot*.
 Posicionar todos os elementos da lista menores do que o *pivot* antes dele.
 Posicionar todos os elementos da lista maiores do que o *pivot* depois dele.
 Aplicar recursivamente o método às sublistas dos lados esquerdo e direito do *pivot*.

Análise Geral

Extraordinariamente eficaz em geral.
 Caso-pior bastante desfavorável.
 Tempo esperado de execução : $\Theta(n \log_2 n)$.
 Constantes (escondidas quando se utiliza a notação Θ) pequenas.
 Efectua a ordenação *in place*: $O(1)$.
 Não é estável.

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

85 utad Pedro Melo-Pinto 2022

05 QUICK SORT

divisão-e-conquista

Utiliza como princípio a partição sucessiva em dois subconjuntos:

- A Escolher um qualquer elemento da lista designado por *pivot*.
- B Posicionar todos os elementos da lista menores do que o *pivot* antes dele.
- C Posicionar todos os elementos da lista maiores do que o *pivot* depois dele.
- D Aplicar recursivamente o método às sublistas dos lados esquerdo e direito do *pivot*.

Análise de uma partição

A As comparações começam no segundo elemento.

B É efectuada uma comparação para cada um destes elementos, excepto

C Nos elementos em que os índices l e r ficam no final ($l = r + 1$) são efectuadas duas comparações

comparações: $n + 1$

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

86 utad Pedro Melo-Pinto 2022

05 QUICK SORT

divisão-e-conquista

Utiliza como princípio a partição sucessiva em dois subconjuntos:

- A Escolher um qualquer elemento da lista designado por *pivot*.
- B Posicionar todos os elementos da lista menores do que o *pivot* antes dele.
- C Posicionar todos os elementos da lista maiores do que o *pivot* depois dele.
- D Aplicar recursivamente o método às sublistas dos lados esquerdo e direito do *pivot*.

Análise de uma partição

Número esperado de trocas para um array cujos elementos estejam ordenados aleatoriamente:

O *pivot* é escolhido aleatoriamente, digamos que fica na posição final k

Cada troca entre $v[l]$ e $v[r]$ representa uma mudança de um elemento originalmente à direita da posição k , para uma posição à esquerda de k

Dos elementos menores que o pivot ($k-1$), espera-se que $\frac{n-k}{n-1} \times (k-1)$ estejam originalmente à direita da posição k

O número esperado de trocas é $\frac{n-k}{n-1} \times (k-1) \sim k \frac{n-k}{n}$

Como todos os valores de k são igualmente prováveis, o número médio de trocas é

$$T_{\text{med}}(n) \sim \frac{1}{n^2} \left(\frac{n^3}{2} - \frac{n^3}{3} \right) \sim \frac{n}{6}$$

comparações: $n + 1$
trocas: $\sim n/6$

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

05 QUICK SORT

divisão-e-conquista

Utiliza como princípio a partição sucessiva em dois subconjuntos:

- Escolher um qualquer elemento da lista designado por *pivot*.
- Posicionar todos os elementos da lista menores do que o *pivot* antes dele.
- Posicionar todos os elementos da lista maiores do que o *pivot* depois dele.
- Aplicar recursivamente o método às sublistas dos lados esquerdo e direito do *pivot*.

Análise de uma partição

Em cada partição existe em média uma troca por cada 6 comparações.

O processo de partição da lista é bastante bom na minimização dos movimentos dos elementos.

Esta é provavelmente a razão pela qual o método *quick sort* tende a ser mais rápido, para o caso geral (esperado), do que outros métodos equivalentes, como o *merge sort*, mesmo fazendo mais comparações do que este.

comparações: $n + 1$
trocas: $\sim n/6$

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

05 QUICK SORT

divisão-e-conquista

Utiliza como princípio a partição sucessiva em dois subconjuntos:

- Escolher um qualquer elemento da lista designado por *pivot*.
- Posicionar todos os elementos da lista menores do que o *pivot* antes dele.
- Posicionar todos os elementos da lista maiores do que o *pivot* depois dele.
- Aplicar recursivamente o método às sublistas dos lados esquerdo e direito do *pivot*.

Análise de uma partição

```
algoritmo para partição em duas sublistas
function partition
    pass in: ARRAY INT v, INT iLeft, iRight
    var: INT pivot, l, r
    l = iLeft + 1
    r = iRight
    pivot = v[iLeft]
    repeat while l < r
        repeat while v[l] ≤ pivot and l < r
            l = l + 1
        end repeat
        repeat while v[r] > pivot
            r = r - 1
        end repeat
        if l < r
            call: swap(v[l], v[r])
        end if
    end repeat
    v[iLeft] = v[r]
    v[r] = pivot
    pass out: r
end function
```

comparações: $n + 1$
trocas: $\sim n/6$

Complexidade algorítmica: $\Theta(n)$

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

05 QUICK SORT
divisão-e-conquista

Utiliza como princípio a partição sucessiva em dois subconjuntos:
 Escolher um qualquer elemento da lista designado por *pivot*.
 Posicionar todos os elementos da lista menores do que o *pivot* antes dele.
 Posicionar todos os elementos da lista maiores do que o *pivot* depois dele.
 Aplicar recursivamente o método às sublistas dos lados esquerdo e direito do *pivot*.

Análise geral do desempenho

Baseia-se no paradigma de divisão-e-conquista.
 O processo de divisão-e-conquista tem, habitualmente, três passos. Neste caso:

- ➊ Divisão
O seu tempo de execução da determinação de *k* é constante.
A partição em duas sublistas tem um tempo de execução
 $P(n) = \Theta(n)$.
- ➋ Conquista
Ordena recursivamente as duas sublistas - $L[p..k-1]$ e $L[k+1..r]$.
 $T(n) = T(k-1) + T(n-k) + cn$.
- ➌ Reunião
As duas sublistas são ordenadas *in place*.
não é necessária efectuar a sua reunião.

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

05 QUICK SORT
divisão-e-conquista

Utiliza como princípio a partição sucessiva em dois subconjuntos:
 Escolher um qualquer elemento da lista designado por *pivot*.
 Posicionar todos os elementos da lista menores do que o *pivot* antes dele.
 Posicionar todos os elementos da lista maiores do que o *pivot* depois dele.
 Aplicar recursivamente o método às sublistas dos lados esquerdo e direito do *pivot*.

Análise do desempenho

O tempo total de execução do algoritmo vai ser a soma do tempo da partição mais duas chamadas recursivas dele próprio (para cada iteração/partição).
 A comparação leva tempo constante e é ignorada.

```

function quicksort
    pass in: ARRAY INT v, INT iLeft,
              iRight
    var: INT iPivot
    if iLeft < iRight
        iPivot ← callpartition(v,iLeft,iRight)
        call:quicksort(v,iLeft,iPivot - 1)
        call:quicksort(v,iPivot + 1,iRight)
    end if
end function
    
```

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

05 QUICK SORT
divisão-e-conquista

Utiliza como princípio a partição sucessiva em dois subconjuntos:
 Escolher um qualquer elemento da lista designado por *pivot*.
 Posicionar todos os elementos da lista menores do que o *pivot* antes dele.
 Posicionar todos os elementos da lista maiores do que o *pivot* depois dele.
 Aplicar recursivamente o método às sublistas dos lados esquerdo e direito do *pivot*.

Análise do desempenho

O tempo total de execução do algoritmo vai ser a soma do tempo da partição mais duas chamadas recursivas dele próprio (para cada iteração/partição).
 A comparação leva tempo constante e é ignorada.

A partição tem um desempenho claramente linear ($\Theta(n)$) - os elementos são percorridos uma só vez no ciclo *repeat-while*.
 Todas as restantes instruções são executadas em tempo constante ($O(1)$).
 O desempenho final vai então depender do número de partições.

```
function partition
    pass in ARRAY[INT] v, INT iLeft, iRight
    var INT pivot, l, r
    l < iLeft + 1
    r < iRight
    pivot <- v[iLeft]
    repeat while l < r
        repeat while v[l] ≤ pivot and l < r
            l +> 1
        end repeat
        repeat while v[r] > pivot
            r -> 1
        end repeat
        if l < r
            call swap(v[l], v[r])
        end if
    end repeat
    v[iLeft] <- v[r]
    v[r] <- pivot
    pass out: r
end function
```

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

05 QUICK SORT
divisão-e-conquista

Utiliza como princípio a partição sucessiva em dois subconjuntos:
 Escolher um qualquer elemento da lista designado por *pivot*.
 Posicionar todos os elementos da lista menores do que o *pivot* antes dele.
 Posicionar todos os elementos da lista maiores do que o *pivot* depois dele.
 Aplicar recursivamente o método às sublistas dos lados esquerdo e direito do *pivot*.

Análise do desempenho

O tempo de execução em cada iteração vai ser a soma do tempo da partição mais duas chamadas recursivas dele próprio.
 A partição tem um desempenho linear: $\Theta(n)$.
 O desempenho final vai depender do número de partições.

The diagram illustrates two extreme cases of partitioning in Quick Sort:

- Caso-pior:** This tree shows an unbalanced partition where the pivot always selects the smallest element (0). The root node splits into two children labeled 0 and n-1. The child 0 splits into 0 and n-2, which further splits into 0 and n-3, and so on. The child n-1 has a single child labeled 1. This results in n partitions.
- Caso-melhor:** This tree shows an balanced partition where the pivot always selects the median element. The root node splits into two children labeled $n/2$. Each $n/2$ child splits into four children labeled $n/4$, and so on. The bottom level of the tree has n leaf nodes, each labeled 1. This results in $\log_2 n$ partitions.

www.dreamalgorithms.org

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

05 QUICK SORT

divisão-e-conquista

Utiliza como princípio a partição sucessiva em dois subconjuntos:

- Escolher um qualquer elemento da lista designado por *pivot*.
- Posicionar todos os elementos da lista menores do que o *pivot* antes dele.
- Posicionar todos os elementos da lista maiores do que o *pivot* depois dele.
- Aplicar recursivamente o método às sublistas dos lados esquerdo e direito do *pivot*.

Análise do desempenho

O tempo de execução em cada iteração vai ser a soma do tempo da partição mais duas chamadas recursivas dele próprio.
A partição tem um desempenho linear: $\Theta(n)$.
O desempenho final vai depender do número de partições.

caso-pior

As partições são sempre o mais desequilibradas possível, ou seja, $k = 1$ ou $k = n$.

$$\begin{aligned} T(n) &= T(n-1) + \Theta(n) \\ &= T(n-2) + 2 \times \Theta(n) \\ &\dots \\ &= 1 + n \times \Theta(n) \\ &= \Theta(n^2) \end{aligned}$$

caso-menor

n partições

```
function partition
    pass in ARRAY V[INT] iLeft, iRight
    var: INT pivot, l, r
    l ← iLeft + 1
    r ← iRight
    pivot ← v[iLeft]
    repeat while l < r
        repeat while v[l] ≤ pivot and l < r
            l ← l + 1
        end repeat
        repeat while v[r] > pivot
            r ← r - 1
        end repeat
        if l < r
            call: swap(v[l], v[r])
        end if
    end repeat
    v[iLeft] ← v[r]
    v[r] ← pivot
    pass out: r
end function
```

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

05 QUICK SORT

divisão-e-conquista

Utiliza como princípio a partição sucessiva em dois subconjuntos:

- Escolher um qualquer elemento da lista designado por *pivot*.
- Posicionar todos os elementos da lista menores do que o *pivot* antes dele.
- Posicionar todos os elementos da lista maiores do que o *pivot* depois dele.
- Aplicar recursivamente o método às sublistas dos lados esquerdo e direito do *pivot*.

Análise do desempenho

O tempo de execução em cada iteração vai ser a soma do tempo da partição mais duas chamadas recursivas dele próprio.
A partição tem um desempenho linear: $\Theta(n)$.
O desempenho final vai depender do número de partições.

caso-pior (análise das comparações)

As partições são sempre o mais desequilibradas possível, ou seja, $k = 1$ ou $k = n$.

$$\begin{aligned} C(n) &= n + C(n-1), C(0) = C(1) = 0 \\ &= n + n-1 + C(n-2) \\ &= n + n-1 + n-2 + C(n-3) \\ &= \frac{n(n+1)}{2} - 1 \\ &\sim \frac{n^2}{2} \end{aligned}$$

caso-menor

n partições

```
function partition
    pass in ARRAY V[INT] iLeft, iRight
    var: INT pivot, l, r
    l ← iLeft + 1
    r ← iRight
    pivot ← v[iLeft]
    repeat while l < r
        repeat while v[l] ≤ pivot and l < r
            l ← l + 1
        end repeat
        repeat while v[r] > pivot
            r ← r - 1
        end repeat
        if l < r
            call: swap(v[l], v[r])
        end if
    end repeat
    v[iLeft] ← v[r]
    v[r] ← pivot
    pass out: r
end function
```

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

05 QUICK SORT

divisão-e-conquista

Utiliza como princípio a partição sucessiva em dois subconjuntos:
 Escolher um qualquer elemento da lista designado por *pivot*.
 Posicionar todos os elementos da lista menores do que o *pivot* antes dele.
 Posicionar todos os elementos da lista maiores do que o *pivot* depois dele.
 Aplicar recursivamente o método às sublistas dos lados esquerdo e direito do *pivot*.

Análise do desempenho

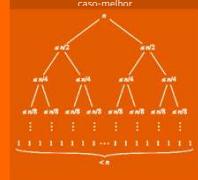
O tempo de execução em cada iteração vai ser a soma do tempo da partição mais duas chamadas recursivas dele próprio.
 A partição tem um desempenho linear: $\Theta(n)$.
 O desempenho final vai depender do número de partições.

caso-melhor

As partições são sempre equilibradas - nenhuma das sublistas resultantes tem mais de $k = n/2$ elementos.

$$\begin{aligned} T(n) &= 2T(n/2) + cn \\ &= 2(2T(n/4) + cn/2) + cn = 2^2 T(n/4) + 2cn \\ &\dots \\ &= 2^k T(n/2^k) + kcn \end{aligned}$$

como $n = 2^k \rightarrow k = \log_2 n$ (caso ideal, senão $n/2^k < 1$, mas podemos manter $k = \log n$)

$$\begin{aligned} &= n \times T(1) + cn \log_2 n \\ &= O(n \log n) \end{aligned}$$


function partition

```
pass int ARRAY[INT] v, INT iLeft, iRight
var int pivot, l, r
l ← iLeft + 1
r ← iRight
pivot ← v[iLeft]
repeat while l < r
    repeat while v[l] ≤ pivot and l < r
        l ← l + 1
    end repeat
    repeat while v[r] > pivot
        r ← r - 1
    end repeat
    if l < r
        call swap(v[l], v[r])
    end if
end repeat
v[iLeft] ← v[r]
v[r] ← pivot
pass out: r
end function
```

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

05 QUICK SORT

divisão-e-conquista

Utiliza como princípio a partição sucessiva em dois subconjuntos:
 Escolher um qualquer elemento da lista designado por *pivot*.
 Posicionar todos os elementos da lista menores do que o *pivot* antes dele.
 Posicionar todos os elementos da lista maiores do que o *pivot* depois dele.
 Aplicar recursivamente o método às sublistas dos lados esquerdo e direito do *pivot*.

Análise do desempenho

O tempo de execução em cada iteração vai ser a soma do tempo da partição mais duas chamadas recursivas dele próprio.
 A partição tem um desempenho linear: $\Theta(n)$.
 O desempenho final vai depender do número de partições.

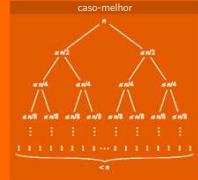
caso-melhor (análise das comparações)

As partições são sempre equilibradas - nenhuma das sublistas resultantes tem mais de $k = n/2$ elementos.

vamos admitir que ambas as partições têm dimensão $(n-1)/2$

$$\begin{aligned} C(n) &= n + 2C((n-1)/2), C(0) = C(1) = 0 \\ &\sim n + 2C(n/2), C(1) = 0 \quad \text{sem perda de generalização} \end{aligned}$$

podemos assumir que $n = 2^k \rightarrow k = \log_2 n$

$$\begin{aligned} &= n + 2(n/2 + 2C(n/4)) = 2n + 4C(n/4) \\ &= 2n + 4(n/4 + 2C(n/8)) = 3n + 8C(n/8) \\ &\dots \\ &= kn + 2^k C(n/2^k) = n \log_2 n + nC(1) = n \log_2 n \end{aligned}$$


function partition

```
pass int ARRAY[INT] v, INT iLeft, iRight
var int pivot, l, r
l ← iLeft + 1
r ← iRight
pivot ← v[iLeft]
repeat while l < r
    repeat while v[l] ≤ pivot and l < r
        l ← l + 1
    end repeat
    repeat while v[r] > pivot
        r ← r - 1
    end repeat
    if l < r
        call swap(v[l], v[r])
    end if
end repeat
v[iLeft] ← v[r]
v[r] ← pivot
pass out: r
end function
```

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

05 QUICK SORT
divisão-e-conquista

Utiliza como princípio a partição sucessiva em dois subconjuntos:
 Escolher um qualquer elemento da lista designado por *pivot*.
 Posicionar todos os elementos da lista menores do que o *pivot* antes dele.
 Posicionar todos os elementos da lista maiores do que o *pivot* depois dele.
 Aplicar recursivamente o método às sublistas dos lados esquerdo e direito do *pivot*.

Análise do desempenho

O tempo de execução em cada iteração vai ser a soma do tempo da partição mais duas chamadas recursivas dele próprio.
 A partição tem um desempenho linear: $\Theta(n)$.
 O desempenho final vai depender do número de partições.

caso geral

Uma partição tem $k-1$ (em que k é a posição final do *pivot*) elementos e outra tem $n-k$ elementos.
 $T(n) = T(k-1) + T(n-k) + cn$

ou, se analisarmos somente as comparações (visto que são dominantes em relação às trocas)
 $C(n) = n + C(k-1) + C(n-k)$
 em que $C(0) = C(1) = 0$

```

function quicksort
    pass in: ARRAY INT v, INT iLeft,
    iRight
    var: INT iPivot
    if iLeft < iRight
        iPivot ← call partition(v,iLeft,iRight)
        call:quicksort(v,iLeft,iPivot - 1)
        call:quicksort(v,iPivot + 1,iRight)
    end if
end function

function partition
    pass in: ARRAY INT v, INT iLeft, iRight
    var: INT pivot
    l ← iLeft + 1
    r ← iRight
    pivot ← v[iLeft]
    repeat while l < r
        repeat while v[l] ≤ pivot and l < r
            l ← l + 1
        end repeat
        repeat while v[r] > pivot
            r ← r - 1
        end repeat
        if l < r
            call: swap(v[l],v[r])
        end if
    end repeat
    v[iLeft] ← v[r]
    v[r] ← pivot
    pass out: r
end function

```

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

05 QUICK SORT
divisão-e-conquista

Utiliza como princípio a partição sucessiva em dois subconjuntos:
 Escolher um qualquer elemento da lista designado por *pivot*.
 Posicionar todos os elementos da lista menores do que o *pivot* antes dele.
 Posicionar todos os elementos da lista maiores do que o *pivot* depois dele.
 Aplicar recursivamente o método às sublistas dos lados esquerdo e direito do *pivot*.

Análise do desempenho

O tempo de execução em cada iteração vai ser a soma do tempo da partição mais duas chamadas recursivas dele próprio.
 A partição tem um desempenho linear: $\Theta(n)$.
 O desempenho final vai depender do número de partições.

caso geral (análise das comparações)

Uma partição tem $k-1$ (em que k é a posição final do *pivot*) elementos e outra tem $n-k$ elementos.
 $C(n) = n + C(k-1) + C(n-k)$, $C(0) = C(1) = 0$
 como todos os valores de k têm a mesma probabilidade de ocorrer, vamos calcular o valor médio
 $C(n) = (1/n) \times \sum_{k=1}^n (n + C(k-1) + C(n-k))$, $C(0) = C(1) = 0$
 $= n + (2/n) \times \sum_{i=0}^{n-1} C(i)$

$nC(n) = n^2 + 2 \times \sum_{i=0}^{n-1} C(i)$

```

function quicksort
    pass in: ARRAY INT v, INT iLeft,
    iRight
    var: INT iPivot
    if iLeft < iRight
        iPivot ← call partition(v,iLeft,iRight)
        call:quicksort(v,iLeft,iPivot - 1)
        call:quicksort(v,iPivot + 1,iRight)
    end if
end function

function partition
    pass in: ARRAY INT v, INT iLeft, iRight
    var: INT pivot
    l ← iLeft + 1
    r ← iRight
    pivot ← v[iLeft]
    repeat while l < r
        repeat while v[l] ≤ pivot and l < r
            l ← l + 1
        end repeat
        repeat while v[r] > pivot
            r ← r - 1
        end repeat
        if l < r
            call: swap(v[l],v[r])
        end if
    end repeat
    v[iLeft] ← v[r]
    v[r] ← pivot
    pass out: r
end function

```

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

05 QUICK SORT
divisão-e-conquista

Utiliza como princípio a partição sucessiva em dois subconjuntos:
 Escolher um qualquer elemento da lista designado por *pivot*.
 Posicionar todos os elementos da lista menores do que o *pivot* antes dele.
 Posicionar todos os elementos da lista maiores do que o *pivot* depois dele.
 Aplicar recursivamente o método às sublistas dos lados esquerdo e direito do *pivot*.

Análise do desempenho

O tempo de execução em cada iteração vai ser a soma do tempo da partição mais duas chamadas recursivas dele próprio.
 A partição tem um desempenho linear: $\Theta(n)$.
 O desempenho final vai depender do número de partições.

caso geral (análise das comparações)

Uma partição tem $k-1$ (em que k é a posição final do *pivot*) elementos e outra tem $n-k$ elementos.
 \dots
 $nC(n) = n^2 + 2 \times \sum_{i=0}^{n-1} C(i)$

se considerarmos $C(n-1)$ e subtrairmos de $C(n)$ fica

$$nC(n) = (n+1)C(n-1) + 2n - 1$$

dividindo por $n(n+1)$

$$C(n)/(n+1) \sim C(n-1)/n + 2/n$$

se considerarmos $D(n) = C(n)/(n+1)$ a recorrência fica:

$$D(n) = D(n-1) + 2/n, D(1) = 0$$

```

function quicksort
    pass in: ARRAY INT v, INT iLeft,
    iRight
    var: INT iPivot
    if iLeft < iRight
        iPivot ← call partition(v,iLeft,iRight)
        call: quicksort(v,iLeft,iPivot - 1)
        call: quicksort(v,iPivot + 1,iRight)
    end if
end function

function partition
    pass in: ARRAY INT v, INT iLeft, iRight
    var: INT pivot
    l ← iLeft + 1
    r ← iRight
    pivot ← v[iLeft]
    repeat while l < r
        repeat while v[l] ≤ pivot and l < r
            l ← l + 1
        end repeat
        repeat while v[r] > pivot
            r ← r - 1
        end repeat
        if l < r
            call: swap(v[l],v[r])
        end if
    end repeat
    v[iLeft] ← v[r]
    v[r] ← pivot
    pass out: r
end function

```

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

05 QUICK SORT
divisão-e-conquista

Utiliza como princípio a partição sucessiva em dois subconjuntos:
 Escolher um qualquer elemento da lista designado por *pivot*.
 Posicionar todos os elementos da lista menores do que o *pivot* antes dele.
 Posicionar todos os elementos da lista maiores do que o *pivot* depois dele.
 Aplicar recursivamente o método às sublistas dos lados esquerdo e direito do *pivot*.

Análise do desempenho

O tempo de execução em cada iteração vai ser a soma do tempo da partição mais duas chamadas recursivas dele próprio.
 A partição tem um desempenho linear: $\Theta(n)$.
 O desempenho final vai depender do número de partições.

caso geral (análise das comparações)

Uma partição tem $k-1$ (em que k é a posição final do *pivot*) elementos e outra tem $n-k$ elementos.
 \dots
 $D(n) = D(n-1) + 2/n, D(1) = 0$
 $= D(n-2) + 2/(n-1) + 2/n = D(n-3) + 2/(n-2) + 2/(n-1) + 2/n$
 \dots
 $= 2 \times \ln(n) - 2 \sim 2 \times \ln(n) = 2 \times \ln(2) \times \log_2(n) \sim 1.39 \log_2(n)$

como $C(n) = D(n)(n+1)$
 $C(n) \sim 1.39(n+1) \log_2(n) \sim 1.39 n \log_2(n)$

```

function quicksort
    pass in: ARRAY INT v, INT iLeft,
    iRight
    var: INT iPivot
    if iLeft < iRight
        iPivot ← call partition(v,iLeft,iRight)
        call: quicksort(v,iLeft,iPivot - 1)
        call: quicksort(v,iPivot + 1,iRight)
    end if
end function

function partition
    pass in: ARRAY INT v, INT iLeft, iRight
    var: INT pivot
    l ← iLeft + 1
    r ← iRight
    pivot ← v[iLeft]
    repeat while l < r
        repeat while v[l] ≤ pivot and l < r
            l ← l + 1
        end repeat
        repeat while v[r] > pivot
            r ← r - 1
        end repeat
        if l < r
            call: swap(v[l],v[r])
        end if
    end repeat
    v[iLeft] ← v[r]
    v[r] ← pivot
    pass out: r
end function

```

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

05 QUICK SORT

divisão-e-conquista

Utiliza como princípio a partição sucessiva em dois subconjuntos:
 Escolher um qualquer elemento da lista designado por *pivot*.
 Posicionar todos os elementos da lista menores do que o *pivot* antes dele.
 Posicionar todos os elementos da lista maiores do que o *pivot* depois dele.
 Aplicar recursivamente o método às sublistas dos lados esquerdo e direito do *pivot*.

Análise do desempenho

O tempo de execução em cada iteração vai ser a soma do tempo da partição mais duas chamadas recursivas dele próprio.
 A partição tem um desempenho linear $\Theta(n)$.
 O desempenho final vai depender do número de partições.

caso geral (análise das comparações)

Uma partição tem $k-1$ (em que k é a posição final do *pivot*) elementos e outra tem $n-k$ elementos.
 $C(n) \sim 1.39 n \log_2(n)$

O caso geral (esperado) do tempo de execução do método quick sort é próximo do seu caso-melhor (cerca de 39% comparações a mais) e afastado do seu caso-pior

```

function quicksort
    pass in: ARRAY INT v, INT iLeft,
    iRight
    var: INT iPivot
    if iLeft < iRight
        iPivot ← call partition(v,iLeft,iRight)
        call quicksort(v,iLeft,iPivot - 1)
        call quicksort(v,iPivot + 1,iRight)
    end if
end function

function partition
    pass in: ARRAY INT v, INT iLeft, iRight
    var: INT pivot, l, r
    l ← iLeft + 1
    r ← iRight
    pivot ← v[iLeft]
    repeat while l < r
        repeat while v[l] ≤ pivot and l < r
            l ← l + 1
        end repeat
        repeat while v[r] > pivot
            r ← r - 1
        end repeat
        if l < r
            call swap(v[l],v[r])
        end if
    end repeat
    v[iLeft] ← v[r]
    v[r] ← pivot
    pass out: r
end function

```

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

05 QUICK SORT

divisão-e-conquista

Utiliza como princípio a partição sucessiva em dois subconjuntos:
 Escolher um qualquer elemento da lista designado por *pivot*.
 Posicionar todos os elementos da lista menores do que o *pivot* antes dele.
 Posicionar todos os elementos da lista maiores do que o *pivot* depois dele.
 Aplicar recursivamente o método às sublistas dos lados esquerdo e direito do *pivot*.

Análise Geral

O tempo de execução depende da divisão (estabelecimento das partições).

A divisão depende do elemento (pivot) utilizado para estabelecer a partição, e pode ser mais ou menos equilibrada.

Se a divisão é equilibrada o tempo de execução do algoritmo é (assimptoticamente) tão rápido como outros algoritmos de ordenação (como é o caso do *merge sort*): $\Theta(n \lg n)$

Se a divisão é desequilibrada, pode ser tão lento quanto o algoritmo de ordenação por inserção: $O(n^2)$

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

103

05 QUICK SORT
divisão-e-conquista

Utiliza como princípio a partição sucessiva em dois subconjuntos:
 Escolher um qualquer elemento da lista designado por *pivot*.
 Posicionar todos os elementos da lista menores do que o *pivot* antes dele.
 Posicionar todos os elementos da lista maiores do que o *pivot* depois dele.
 Aplicar recursivamente o método às sublistas dos lados esquerdo e direito do *pivot*.

Análise Geral

O tempo de execução depende da divisão (estabelecimento das partições).

Na pior situação, faz cerca de $n^2/2$ comparações.

Em média (esperado), faz cerca de $2n \log_2 n$ comparações.
 (No melhor caso, faz cerca de $n \log_2 n$ comparações).

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

104

05 QUICK SORT
divisão-e-conquista

Utiliza como princípio a partição sucessiva em dois subconjuntos:
 Escolher um qualquer elemento da lista designado por *pivot*.
 Posicionar todos os elementos da lista menores do que o *pivot* antes dele.
 Posicionar todos os elementos da lista maiores do que o *pivot* depois dele.
 Aplicar recursivamente o método às sublistas dos lados esquerdo e direito do *pivot*.

Análise Geral

O tempo de execução depende da divisão (estabelecimento das partições).

Estabelecimento da divisão com uma escolha adequada do pivot:
 (métodos habituais)
 escolher aleatoriamente na lista (*randomized quicksort*)
 escolher o elemento a meio da lista
 escolher a mediana-de-três (primeiro, último e do meio)

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

105

05 QUICK SORT
divisão-e-conquista

Utiliza como princípio a partição sucessiva em dois subconjuntos:
 Escolher um qualquer elemento da lista designado por *pivot*.
 Posicionar todos os elementos da lista menores do que o *pivot* antes dele.
 Posicionar todos os elementos da lista maiores do que o *pivot* depois dele.
 Aplicar recursivamente o método às sublistas dos lados esquerdo e direito do *pivot*.

Análise do desempenho

O tempo de execução em cada iteração vai ser a soma do tempo da partição mais duas chamadas recursivas dele próprio.
 A partição tem um desempenho linear: $\Theta(n)$.
 O desempenho final vai depender do número de partições.

caso geral

Vamos admitir que as partições são menos equilibradas, mas que temos sempre pelo menos uma razão de 3 para 1 na divisão dos elementos.
 Ou seja uma das sublistas tem $n/4$ elementos e a outra $3n/4$.
 (Não consideraremos o *pivot* para manter a matemática mais simples, sem perda de generalidade)

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

106

05 QUICK SORT
divisão-e-conquista

Utiliza como princípio a partição sucessiva em dois subconjuntos:
 Escolher um qualquer elemento da lista designado por *pivot*.
 Posicionar todos os elementos da lista menores do que o *pivot* antes dele.
 Posicionar todos os elementos da lista maiores do que o *pivot* depois dele.
 Aplicar recursivamente o método às sublistas dos lados esquerdo e direito do *pivot*.

Análise do desempenho

O tempo de execução em cada iteração vai ser a soma do tempo da partição mais duas chamadas recursivas dele próprio.
 A partição tem um desempenho linear: $\Theta(n)$.
 O desempenho final vai depender do número de partições.

caso geral

Chegamos mais rápido a partições de dimensão unitária do lado da partição de $n/4$:
 $\log_4 n$ partições
 No lado da partição de $3n/4$:
 $\log_{4/3} n$ partições

Nos primeiros $\log_4 n$ níveis existem n elementos e o tempo de partição é cn .
 No níveis seguintes o número de elementos em análise diminui, e esse tempo é $< cn$.
 Como $\log_{4/3} n = \log_2 n / \log_2 (4/3)$ e como o número máximo de partições é $\log_{4/3} n$

$T(n) = O(n \log_2 n)$

caso com partições cuja razão é de 3 para 1

www.brunoculpo.org

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

107

05 QUICK SORT
divisão-e-conquista

Utiliza como princípio a partição sucessiva em dois subconjuntos:
 Escolher um qualquer elemento da lista designado por *pivot*.
 Posicionar todos os elementos da lista menores do que o *pivot* antes dele.
 Posicionar todos os elementos da lista maiores do que o *pivot* depois dele.
 Aplicar recursivamente o método às sublistas dos lados esquerdo e direito do *pivot*.

Análise do desempenho

O tempo de execução em cada iteração vai ser a soma do tempo da partição mais duas chamadas recursivas dele próprio.
 A partição tem um desempenho linear $\Theta(n)$.
 O desempenho final vai depender do número de partições.

caso geral

Para que haja uma razão de 3 para 1 (ou melhor) na divisão dos elementos o pivot tem de estar na "metade do meio" dos elementos ordenados:



Se a probabilidade do pivot acabar na posição k , for igual para todos os k , existe uma probabilidade de 50% de obtermos uma divisão de 3 para 1 (ou melhor).

www.tilmanoefen.org

caso com partições cuja razão é de 3 para 1



Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

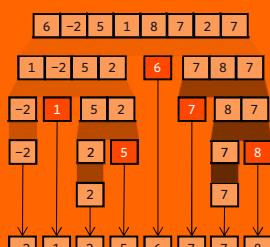
108

05 QUICK SORT
divisão-e-conquista

Utiliza como princípio a partição sucessiva em dois subconjuntos:
 Escolher um qualquer elemento da lista designado por *pivot*.
 Posicionar todos os elementos da lista menores do que o *pivot* antes dele.
 Posicionar todos os elementos da lista maiores do que o *pivot* depois dele.
 Aplicar recursivamente o método às sublistas dos lados esquerdo e direito do *pivot*.

Análise Geral (melhorias)

Estabelecer um valor m (designado normalmente por valor de corte - *threshold*) para o tamanho das sublistas, a partir do qual o método utilizado muda para um cujas constantes sejam menos pesadas em listas pequenas (por exemplo ordenação por inserção).



Algoritma (e Estruturas de Dados)

ALGORITMOS ORDENAÇÃO

05 QUICK SORT

divisão-e-conquista

109

utad

Pedro Melo-Pinto 2022

| Número de Elementos (random) | Inteiros | | | Strings | | |
|---------------------------------|----------|----------|-------|---------|----------|-------|
| | Seleção | Inserção | Fusão | Seleção | Inserção | Fusão |
| 1000 | 64 | 6,0 | 394 | 1,9 | 0,3 | 354 |
| 2000 | 128 | 12,0 | 761 | 2,0 | 0,4 | 741 |
| 5000 | 1061 | 6,6 | 6359 | 2669 | 1543 | 10162 |
| 10000 | 1,0 | 1,0 | 1,0 | 1,0 | 1,0 | 1,0 |
| 20000 | 4,0 | 4,1 | 4,0 | 4,4 | 4,4 | 4,0 |
| 50000 | 16,3 | 16,4 | 16,1 | 17,0 | 18,6 | 18,3 |

| Número de Elementos (random) | Inteiros | | | Strings | | |
|---------------------------------|----------|----------|-------|---------|----------|-------|
| | Seleção | Inserção | Fusão | Seleção | Inserção | Fusão |
| 1000 | 66 | 0 | 0 | 273 | 0 | 0 |
| 2000 | 287 | 0 | 0 | 1990 | 0 | 0 |
| 5000 | 1050 | 0 | 0 | 4147 | 0 | 0 |
| 10000 | 0 | 0 | 0 | 0 | 0 | 0 |

| evolução de $n \times \log_2(n)$ | Inteiros | | | Strings | | |
|----------------------------------|----------|-------|------|---------|-------|------|
| | Quick | Merge | Heap | Quick | Merge | Heap |
| 2,73 | 2,4 | 2,6 | | 3,4 | 3,2 | |
| 5,81 | 5,6 | 5,6 | | 7,8 | 7,6 | |

Algoritma (e Estruturas de Dados)

ALGORITMOS ORDENAÇÃO

05 QUICK SORT

divisão-e-conquista

110

utad

Pedro Melo-Pinto 2022

Utiliza como princípio a partição sucessiva em dois subconjuntos:
Escolher um qualquer elemento da lista designado por *pivot*.
Posicionar todos os elementos da lista menores do que o *pivot* antes dele.
Posicionar todos os elementos da lista maiores do que o *pivot* depois dele.
Aplicar recursivamente o método às sublistas dos lados esquerdo e direito do *pivot*.

O que acontece quando todos os elementos a ordenar são iguais?

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

111

06 HEAP SORT
transformação-e-conquista

Utiliza o paradigma da transformação-e-conquista, utilizando uma diferente representação (através de *heaps*) do mesmo problema.

Pode dizer-se que se comporta como o algoritmo de ordenação por selecção, em que a selecção do mínimo em cada iteração é feita de modo eficiente (através de uma *heap* de máximo).

Seja a lista (a_1, a_2, \dots, a_n) .

- 1 Constrói-se uma heap de máximo a partir da lista.
- 2 Troca-se o primeiro (raiz - valor máximo dos elementos da lista) com o último elemento da *heap/lista*.
- 3 Retira-se o último elemento da *heap*.
- 4 Reorganiza-se a *heap* (*max-heapification*).
- 5 Repete-se 2-4 até não haver elementos na *heap*.

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

112

06 HEAP SORT
transformação-e-conquista

Utiliza o paradigma da transformação-e-conquista, utilizando uma diferente representação do mesmo problema.

Constrói-se uma heap de máximo a partir da lista.
 Troca-se o primeiro com o último elemento da *heap*.
 Retira-se o último elemento da *heap*.
 Reorganiza-se a *heap* (*max-heapification*).
 Repete-se até não haver elementos na *heap*.

Algoritmo

```

function heapsort
    pass in: ARRAY INT v
    var: INT n,i
    call: buildmaxheap(v)
    n ← call:length(v)
    repeat for i ← n downto 2
        swap(v[1],v[i])
        call: maxheapify(v, 1, i-1)
    end repeat
end function

```

exemplo: lista {4,1,3,7,2}

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

113
utad
Pedro Melo-Pinto 2022

06 HEAP SORT

transformação-e-conquista

Utiliza o paradigma da transformação-e-conquista, utilizando uma diferente representação do mesmo problema.

Constrói-se uma heap de máximo a partir da lista.
Trocá-se o primeiro com o último elemento da *heap*.
Retira-se o último elemento da *heap*.
Reorganiza-se a *heap* (*max-heapification*).
Repete-se até não haver elementos na *heap*.

Algoritmo

```

function heapsort
    pass in: ARRAY INT v
    var: INT n, i
    call: buildmaxheap(v)
    n ← call: length(v)
    repeat for i ← n down to 2
        swap(v[1], v[i])
        call: maxheapify(v, 1, i-1)
    end repeat
end function

function buildmaxheap
    pass in: ARRAY INT v
    var: INT n, i
    n ← call: length(v)
    repeat for i ← floor(n/2) down to 1
        call: maxheapify(v, i, n)
    end repeat
end function

```

```

function maxheapify
    pass in: ARRAY INT v, INT i, n
    var: INT iLeft, iRight, iMax
    iLeft ← 2 × i
    iRight ← 2 × i + 1
    if iLeft ≤ n and v[iLeft] > v[i]
        iMax ← iLeft
    else
        iMax ← i
    end if
    if iRight ≤ n and v[iRight] > v[iMax]
        iMax ← iRight
    end if
    if iMax ≠ i
        swap(v[i], v[iMax])
        call: maxheapify(v, iMax, n)
    end if
end function

```

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

114
utad
Pedro Melo-Pinto 2022

06 HEAP SORT

transformação-e-conquista

Utiliza o paradigma da transformação-e-conquista, utilizando uma diferente representação do mesmo problema.

Constrói-se uma heap de máximo a partir da lista.
Trocá-se o primeiro com o último elemento da *heap*.
Retira-se o último elemento da *heap*.
Reorganiza-se a *heap* (*max-heapification*).
Repete-se até não haver elementos na *heap*.

Algoritmo

```

function heapsort
    pass in: ARRAY INT v
    var: INT n, i
    call: buildmaxheap(v)
    n ← call: length(v)
    repeat for i ← n down to 2
        swap(v[1], v[i])
        call: maxheapify(v, 1, i-1)
    end repeat
end function

function buildmaxheap
    pass in: ARRAY INT v
    var: INT n, i
    n ← call: length(v)
    repeat for i ← floor(n/2) down to 1
        call: maxheapify(v, i, n)
    end repeat
end function

```

```

function maxheapify
    pass in: ARRAY INT v, INT i, n
    var: INT iLeft, iRight, iMax
    iLeft ← 2 × i
    iRight ← 2 × i + 1
    if iLeft ≤ n and v[iLeft] > v[i]
        iMax ← iLeft
    else
        iMax ← i
    end if
    if iRight ≤ n and v[iRight] > v[iMax]
        iMax ← iRight
    end if
    if iMax ≠ i
        swap(v[i], v[iMax])
        call: maxheapify(v, iMax, n)
    end if
end function

```

exemplo: lista (4,1,3,7,2)

heap

```

graph TD
    4((4)) --> 1((1))
    4((4)) --> 3((3))
    1((1)) --> 7((7))
    1((1)) --> 2((2))

```

array correspondente

| | | | | |
|---|---|---|---|---|
| 4 | 1 | 3 | 7 | 2 |
| 1 | 2 | 3 | 4 | 5 |

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

06 HEAP SORT
transformação-e-conquista

Utiliza o paradigma da transformação-e-conquista, utilizando uma diferente representação do mesmo problema.

Constrói-se uma heap de máximo a partir da lista.
Trocá-se o primeiro com o último elemento da *heap*.
Retira-se o último elemento da *heap*.
Reorganiza-se a *heap* (*max-heapification*).
Repete-se até não haver elementos na *heap*.

Algoritmo

```

function heapsort
    pass in: ARRAY INT v
    var: INT n, i
    call: buildmaxheap(v)
    n ← call: length(v)
    repeat for i ← n downto 2
        swap(v[1], v[i])
        call: maxheapify(v, 1, i-1)
    end repeat
    end function

```

**troca a raiz da heap
(valor máximo) com o
último elemento válido**

```

function maxheapify
    pass in: ARRAY INT v, INT i
    var: INT iLeft, iRight, iMax
    iLeft ← 2 × i
    iRight ← 2 × i + 1
    if iLeft ≤ n and v[iLeft] > v[i]
        iMax ← iLeft
    else
        iMax ← i
    end if
    if iRight ≤ n and v[iRight] > v[iMax]
        iMax ← iRight
    end if
    if iMax ≠ i
        swap(v[i], v[iMax])
        call: maxheapify(v, iMax, n)
    end if
end function

```

exemplo: lista {4,1,3,7,2}

heap

array correspondente

| | | | | |
|---|---|---|---|---|
| 2 | 4 | 3 | 1 | 7 |
| 1 | 2 | 3 | 4 | 5 |

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

06 HEAP SORT
transformação-e-conquista

Utiliza o paradigma da transformação-e-conquista, utilizando uma diferente representação do mesmo problema.

Constrói-se uma heap de máximo a partir da lista.
Trocá-se o primeiro com o último elemento da *heap*.
Retira-se o último elemento da *heap*.
Reorganiza-se a *heap* (*max-heapification*).
Repete-se até não haver elementos na *heap*.

Algoritmo

```

function heapsort
    pass in: ARRAY INT v
    var: INT n, i
    call: buildmaxheap(v)
    n ← call: length(v)
    repeat for i ← n downto 2
        swap(v[1], v[i])
        call: maxheapify(v, 1, i-1)
    end repeat
    end function

```

**retira-se o último
elemento da heap e
reorganiza-se esta**

```

function maxheapify
    pass in: ARRAY INT v, INT i
    var: INT iLeft, iRight, iMax
    iLeft ← 2 × i
    iRight ← 2 × i + 1
    if iLeft ≤ n and v[iLeft] > v[i]
        iMax ← iLeft
    else
        iMax ← i
    end if
    if iRight ≤ n and v[iRight] > v[iMax]
        iMax ← iRight
    end if
    if iMax ≠ i
        swap(v[i], v[iMax])
        call: maxheapify(v, iMax, n)
    end if
end function

```

exemplo: lista {4,1,3,7,2}

heap

array correspondente

| | | | | |
|---|---|---|---|---|
| 2 | 4 | 3 | 1 | 7 |
| 1 | 2 | 3 | 4 | 5 |

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

06 HEAP SORT
transformação-e-conquista

Utiliza o paradigma da transformação-e-conquista, utilizando uma diferente representação do mesmo problema.

Constrói-se uma heap de máximo a partir da lista.
Trocá-se o primeiro com o último elemento da *heap*.
Retira-se o último elemento da *heap*.
Reorganiza-se a *heap* (*max-heapification*).
Repete-se até não haver elementos na *heap*.

Algoritmo

```

function heapsort
    pass in: ARRAY INT v
    var: INT n, i
    call: buildmaxheap(v)
    n ← call: length(v)
    repeat for i ← n downto 2
        swap(v[1], v[i])
        call: maxheapify(v, 1, i-1)
    end repeat
    end function

```

retira-se o último elemento da *heap* e reorganiza-se esta

```

function maxheapify
    pass in: ARRAY INT v, INT i
    var: INT iLeft, iRight, iMax
    iLeft ← 2 × i
    iRight ← 2 × i + 1
    if iLeft ≤ n and v[iLeft] > v[i]
        iMax ← iLeft
    else
        iMax ← i
    end if
    if iRight ≤ n and v[iRight] > v[iMax]
        iMax ← iRight
    end if
    if iMax ≠ i
        swap(v[i], v[iMax])
        call: maxheapify(v, iMax, n)
    end if
end function

```

exemplo: lista {4,1,3,7,2}

heap

```

graph TD
    4((4)) --- 2((2))
    4 --- 3((3))
    2 --- 1((1))

```

array correspondente

| | | | | |
|---|---|---|---|---|
| 4 | 2 | 3 | 1 | 7 |
| 1 | 2 | 3 | 4 | 5 |

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

06 HEAP SORT
transformação-e-conquista

Utiliza o paradigma da transformação-e-conquista, utilizando uma diferente representação do mesmo problema.

Constrói-se uma heap de máximo a partir da lista.
Trocá-se o primeiro com o último elemento da *heap*.
Retira-se o último elemento da *heap*.
Reorganiza-se a *heap* (*max-heapification*).
Repete-se até não haver elementos na *heap*.

Algoritmo

```

function heapsort
    pass in: ARRAY INT v
    var: INT n, i
    call: buildmaxheap(v)
    n ← call: length(v)
    repeat for i ← n downto 2
        swap(v[1], v[i])
        call: maxheapify(v, 1, i-1)
    end repeat
    end function

```

troca a raiz da *heap* (valor máximo) com o último elemento válido

```

function maxheapify
    pass in: ARRAY INT v, INT i
    var: INT iLeft, iRight, iMax
    iLeft ← 2 × i
    iRight ← 2 × i + 1
    if iLeft ≤ n and v[iLeft] > v[i]
        iMax ← iLeft
    else
        iMax ← i
    end if
    if iRight ≤ n and v[iRight] > v[iMax]
        iMax ← iRight
    end if
    if iMax ≠ i
        swap(v[i], v[iMax])
        call: maxheapify(v, iMax, n)
    end if
end function

```

exemplo: lista {4,1,3,7,2}

heap

```

graph TD
    1((1)) --- 2((2))
    1 --- 3((3))
    2 --- 4((4))

```

array correspondente

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 7 |
| 1 | 2 | 3 | 4 | 5 |

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

06 HEAP SORT
transformação-e-conquista

Utiliza o paradigma da transformação-e-conquista, utilizando uma diferente representação do mesmo problema.

Constrói-se uma heap de máximo a partir da lista.
Trocá-se o primeiro com o último elemento da *heap*.
Retira-se o último elemento da *heap*.
Reorganiza-se a *heap* (*max-heapification*).
Repete-se até não haver elementos na *heap*.

Algoritmo

```

function heapsort
    pass in: ARRAY INT v
    var: INT n, i
    call: buildmaxheap(v)
    n ← call: length(v)
    repeat for i ← n downto 2
        swap(v[1], v[i])
        call: maxheapify(v, 1, i-1)
    end repeat
    end function

```

retira-se o último elemento da *heap* e reorganiza-se esta

```

function maxheapify
    pass in: ARRAY INT v, INT i
    var: INT iLeft, iRight, iMax
    iLeft ← 2 × i
    iRight ← 2 × i + 1
    if iLeft ≤ n and v[iLeft] > v[i]
        iMax ← iLeft
    else
        iMax ← i
    end if
    if iRight ≤ n and v[iRight] > v[iMax]
        iMax ← iRight
    end if
    if iMax ≠ i
        swap(v[i], v[iMax])
        call: maxheapify(v, iMax, n)
    end if
end function

```

exemplo: lista {4,1,3,7,2}

heap

array correspondente

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 7 |
| 1 | 2 | 3 | 4 | 5 |

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

06 HEAP SORT
transformação-e-conquista

Utiliza o paradigma da transformação-e-conquista, utilizando uma diferente representação do mesmo problema.

Constrói-se uma heap de máximo a partir da lista.
Trocá-se o primeiro com o último elemento da *heap*.
Retira-se o último elemento da *heap*.
Reorganiza-se a *heap* (*max-heapification*).
Repete-se até não haver elementos na *heap*.

Algoritmo

```

function heapsort
    pass in: ARRAY INT v
    var: INT n, i
    call: buildmaxheap(v)
    n ← call: length(v)
    repeat for i ← n downto 2
        swap(v[1], v[i])
        call: maxheapify(v, 1, i-1)
    end repeat
    end function

```

retira-se o último elemento da *heap* e reorganiza-se esta

```

function maxheapify
    pass in: ARRAY INT v, INT i
    var: INT iLeft, iRight, iMax
    iLeft ← 2 × i
    iRight ← 2 × i + 1
    if iLeft ≤ n and v[iLeft] > v[i]
        iMax ← iLeft
    else
        iMax ← i
    end if
    if iRight ≤ n and v[iRight] > v[iMax]
        iMax ← iRight
    end if
    if iMax ≠ i
        swap(v[i], v[iMax])
        call: maxheapify(v, iMax, n)
    end if
end function

```

exemplo: lista {4,1,3,7,2}

heap

array correspondente

| | | | | |
|---|---|---|---|---|
| 3 | 2 | 1 | 4 | 7 |
| 1 | 2 | 3 | 4 | 5 |

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

121 utad Pedro Melo-Pinto 2022

06 HEAP SORT

transformação-e-conquista

Utiliza o paradigma da transformação-e-conquista, utilizando uma diferente representação do mesmo problema.

Constrói-se uma heap de máximo a partir da lista.
Trocá-se o primeiro com o último elemento da heap.
Retira-se o último elemento da heap.
Reorganiza-se a heap (*max-heapification*).
Repete-se até não haver elementos na heap.

Algoritmo

```

function heapsort
    pass in: ARRAY INT v
    var: INT n, i
    call: buildmaxheap(v)
    n ← call: length(v)
    repeat for i ← n downto 2
        swap(v[1], v[i])
        call: maxheapify(v, 1, i-1)
    end repeat
    end function

```

troca a raiz da heap
(valor máximo) com o
último elemento válido

```

function maxheapify
    pass in: ARRAY INT v, INT i
    var: INT iLeft, iRight, iMax
    iLeft ← 2 × i
    iRight ← 2 × i + 1
    if iLeft ≤ n and v[iLeft] > v[i]
        iMax ← iLeft
    else
        iMax ← i
    end if
    if iRight ≤ n and v[iRight] > v[iMax]
        iMax ← iRight
    end if
    if iMax ≠ i
        swap(v[i], v[iMax])
        call: maxheapify(v, iMax, n)
    end if
end function

```

exemplo: lista {4,1,3,7,2}

heap

array correspondente

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 7 |
| 1 | 2 | 3 | 4 | 5 |

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

122 utad Pedro Melo-Pinto 2022

06 HEAP SORT

transformação-e-conquista

Utiliza o paradigma da transformação-e-conquista, utilizando uma diferente representação do mesmo problema.

Constrói-se uma heap de máximo a partir da lista.
Trocá-se o primeiro com o último elemento da heap.
Retira-se o último elemento da heap.
Reorganiza-se a heap (*max-heapification*).
Repete-se até não haver elementos na heap.

Algoritmo

```

function heapsort
    pass in: ARRAY INT v
    var: INT n, i
    call: buildmaxheap(v)
    n ← call: length(v)
    repeat for i ← n downto 2
        swap(v[1], v[i])
        call: maxheapify(v, 1, i-1)
    end repeat
    end function

```

retira-se o último
elemento da heap e
reorganiza-se esta

```

function maxheapify
    pass in: ARRAY INT v, INT i
    var: INT iLeft, iRight, iMax
    iLeft ← 2 × i
    iRight ← 2 × i + 1
    if iLeft ≤ n and v[iLeft] > v[i]
        iMax ← iLeft
    else
        iMax ← i
    end if
    if iRight ≤ n and v[iRight] > v[iMax]
        iMax ← iRight
    end if
    if iMax ≠ i
        swap(v[i], v[iMax])
        call: maxheapify(v, iMax, n)
    end if
end function

```

exemplo: lista {4,1,3,7,2}

heap

array correspondente

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 7 |
| 1 | 2 | 3 | 4 | 5 |

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

06 HEAP SORT

transformação-e-conquista

Utiliza o paradigma da transformação-e-conquista, utilizando uma diferente representação do mesmo problema.

Constrói-se uma heap de máximo a partir da lista.
Trocá-se o primeiro com o último elemento da *heap*.
Retira-se o último elemento da *heap*.
Reorganiza-se a *heap* (*max-heapification*).
Repete-se até não haver elementos na *heap*.

Algoritmo

```

function heapsort
    pass in: ARRAY INT v
    var: INT n, i
    call: buildmaxheap(v)
    n ← call: length(v)
    repeat for i ← n downto 2
        swap(v[1], v[i])
        call: maxheapify(v, 1, i-1)
    end repeat
    end function

```

troca a raiz da *heap*
(valor máximo) com o
último elemento válido

```

function maxheapify
    pass in: ARRAY INT v, INT i
    var: INT iLeft, iRight, iMax
    iLeft ← 2 × i
    iRight ← 2 × i + 1
    if iLeft ≤ n and v[iLeft] > v[i]
        iMax ← iLeft
    else
        iMax ← i
    end if
    if iRight ≤ n and v[iRight] > v[iMax]
        iMax ← iRight
    end if
    if iMax ≠ i
        swap(v[i], v[iMax])
        call: maxheapify(v, iMax, n)
    end if
end function

```

exemplo: lista {4,1,3,7,2}

heap

array correspondente

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 7 |
| 1 | 2 | 3 | 4 | 5 |

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

06 HEAP SORT

transformação-e-conquista

Utiliza o paradigma da transformação-e-conquista, utilizando uma diferente representação do mesmo problema.

Constrói-se uma heap de máximo a partir da lista.
Trocá-se o primeiro com o último elemento da *heap*.
Retira-se o último elemento da *heap*.
Reorganiza-se a *heap* (*max-heapification*).
Repete-se até não haver elementos na *heap*.

Algoritmo

```

function heapsort
    pass in: ARRAY INT v
    var: INT n, i
    call: buildmaxheap(v)
    n ← call: length(v)
    repeat for i ← n downto 2
        swap(v[1], v[i])
        call: maxheapify(v, 1, i-1)
    end repeat
    end function

```

retira-se o último
elemento da *heap* e
reorganiza-se esta

```

function maxheapify
    pass in: ARRAY INT v, INT i
    var: INT iLeft, iRight, iMax
    iLeft ← 2 × i
    iRight ← 2 × i + 1
    if iLeft ≤ n and v[iLeft] > v[i]
        iMax ← iLeft
    else
        iMax ← i
    end if
    if iRight ≤ n and v[iRight] > v[iMax]
        iMax ← iRight
    end if
    if iMax ≠ i
        swap(v[i], v[iMax])
        call: maxheapify(v, iMax, n)
    end if
end function

```

exemplo: lista {4,1,3,7,2}

heap

array correspondente

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 7 |
| 1 | 2 | 3 | 4 | 5 |

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

06 HEAP SORT
transformação-e-conquista

Utiliza o paradigma da transformação-e-conquista, utilizando uma diferente representação do mesmo problema.

Constrói-se uma heap de máximo a partir da lista.
Trocá-se o primeiro com o último elemento da *heap*.
Retira-se o último elemento da *heap*.
Reorganiza-se a *heap* (*max-heapification*).
Repete-se até não haver elementos na *heap*.

Algoritmo

```
function heapsort
    pass in: ARRAY INT v
    var: INT n, i
    call: buildmaxheap(v)
    n ← call: length(v)
    repeat for i ← n downto 2
        swap(v[1], v[i])
        call: maxheapify(v, 1, i-1)
    end repeat
    end function
```

retira-se o último elemento da *heap* e reorganiza-se esta

function maxheapify
 pass in: ARRAY INT v, INT i
 var: INT iLeft, iRight, iMax
 iLeft ← 2 × i
 iRight ← 2 × i + 1
 if iLeft ≤ n and v[iLeft] > v[i]
 iMax ← iLeft
 else
 iMax ← iLeft
 end if
 if iRight ≤ n and v[iRight] > v[iMax]
 iMax ← iRight
 end if
 if iMax ≠ i
 swap(v[i], v[iMax])
 call: maxheapify(v, iMax, n)
 end if
end function

exemplo: lista {4,1,3,7,2}

heap

array correspondente

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 7 |
| 1 | 2 | 3 | 4 | 5 |

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

06 HEAP SORT
transformação-e-conquista

Utiliza o paradigma da transformação-e-conquista, utilizando uma diferente representação do mesmo problema.

Constrói-se uma heap de máximo a partir da lista.
Trocá-se o primeiro com o último elemento da *heap*.
Retira-se o último elemento da *heap*.
Reorganiza-se a *heap* (*max-heapification*).
Repete-se até não haver elementos na *heap*.

Algoritmo

```
function heapsort
    pass in: ARRAY INT v
    var: INT n, i
    call: buildmaxheap(v)
    n ← call: length(v)
    repeat for i ← n downto 2
        swap(v[1], v[i])
        call: maxheapify(v, 1, i-1)
    end repeat
    end function
```

function maxheapify
 pass in: ARRAY INT v, INT i
 var: INT iLeft, iRight, iMax
 iLeft ← 2 × i
 iRight ← 2 × i + 1
 if iLeft ≤ n and v[iLeft] > v[i]
 iMax ← iLeft
 else
 iMax ← iLeft
 end if
 if iRight ≤ n and v[iRight] > v[iMax]
 iMax ← iRight
 end if
 if iMax ≠ i
 swap(v[i], v[iMax])
 call: maxheapify(v, iMax, n)
 end if
end function

exemplo: lista {4,1,3,7,2}

heap

array correspondente

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 7 |
| 1 | 2 | 3 | 4 | 5 |

a lista está ordenada

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

127 utad Pedro Melo-Pinto 2022

06 HEAP SORT

transformação-e-conquista

Utiliza o paradigma da transformação-e-conquista, utilizando uma diferente representação do mesmo problema.

Constrói-se uma heap de máximo a partir da lista.

Troca-se o primeiro com o último elemento da heap.

Retira-se o último elemento da heap.

Reorganiza-se a heap (*max-heapification*).

Repete-se até não haver elementos na heap.

Análise Geral

Baseia-se no paradigma de transformação-e-conquista.

Transforma o problema de ordenação de uma lista num problema de extração sucessiva da raiz de uma heap de máximo.

The diagram shows the transformation of an array into a max-heap. It starts with the array [7, 2, 3, 1, 5, 4]. The first step shows a max-heap where node 7 is the root. The second step shows the heap after removing the root (7) and re-heapifying. The third step shows the heap after removing the new root (5). The fourth step shows the heap after removing the new root (4). The fifth step shows the heap after removing the new root (3). The final step shows the heap after removing the last element (2), leaving the sorted array [1, 5, 4, 3, 2, 7].

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

128 utad Pedro Melo-Pinto 2022

06 HEAP SORT

transformação-e-conquista

Utiliza o paradigma da transformação-e-conquista, utilizando uma diferente representação do mesmo problema.

Constrói-se uma heap de máximo a partir da lista.

Troca-se o primeiro com o último elemento da heap.

Retira-se o último elemento da heap.

Reorganiza-se a heap (*max-heapification*).

Repete-se até não haver elementos na heap.

Análise Geral

Baseia-se no paradigma de transformação-e-conquista.

Transforma o problema de ordenação de uma lista num problema de extração sucessiva da raiz de uma heap de máximo.

O processo tem dois passos principais:

- Criação da heap de máximo (*max-heap*)
Este passo tem tempo de execução:
 $C(n) = O(n)$.
- Extração sucessiva da raiz desta heap e subsequente reorganização
A extração executa em tempo constante $O(1)$, logo a complexidade depende da reorganização:
 $R(n) = O(n \log_2 n)$.

The diagram shows the transformation of an array into a max-heap. It starts with the array [7, 2, 3, 1, 5, 4]. The first step shows a max-heap where node 7 is the root. The second step shows the heap after removing the root (7) and re-heapifying. The third step shows the heap after removing the new root (5). The fourth step shows the heap after removing the new root (4). The fifth step shows the heap after removing the new root (3). The final step shows the heap after removing the last element (2), leaving the sorted array [1, 5, 4, 3, 2, 7].

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

129
utad
Pedro Melo-Pinto 2022

06 HEAP SORT

transformação-e-conquista

Utiliza o paradigma da transformação-e-conquista, utilizando uma diferente representação do mesmo problema.

Constrói-se uma heap de máximo a partir da lista.
Trocá-se o primeiro com o último elemento da *heap*.
Retira-se o último elemento da *heap*.
Reorganiza-se a *heap* (*max-heapification*).
Repete-se até não haver elementos na *heap*.

Análise Geral

Eficaz em geral.
Tempo esperado de execução : $\Theta(n \lg n)$.
É, para o caso geral, mais lento do que o método *Quick Sort*.
Não tem um caso-pior.
Efectua a ordenação *in place*: $O(1)$.
Não é estável.

Pode ser facilmente adaptado para trabalhar com listas encadeadas.

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

130
utad
Pedro Melo-Pinto 2022

06 HEAP SORT

transformação-e-conquista

Utiliza o paradigma da transformação-e-conquista, utilizando uma diferente representação do mesmo problema.

Constrói-se uma heap de máximo a partir da lista.
Trocá-se o primeiro com o último elemento da *heap*.
Retira-se o último elemento da *heap*.
Reorganiza-se a *heap* (*max-heapification*).
Repete-se até não haver elementos na *heap*.

Análise da reorganização (*heapify*) de uma *heap*

maxheapify é chamada recursivamente h vezes (no máximo), em que h é a altura da subárvore cuja raiz está em i

os restantes passos de *maxheapify* executam em tempo constante c

o **caso-pior** de h (maior desequilíbrio da *heap*) ocorre quando o último nível está meio cheio

nesse caso o número de nós da *subheap* do lado esquerdo é: $L = 2 \times 2^{h-1} - 1$, e o número de nós da *heap* é: $H = 2^h + 2^{h-1} - 1$, ou seja,
 $L = 2/3 H$

$T(n) \leq T(2n/3) + O(1)$

```
function maxheapify
    pass in: ARRAY INT v, INT i, n
    var:    INT iLeft, iRight, iMax
    iLeft ← 2 × i
    iRight ← 2 × i + 1
    if iLeft ≤ n and v[iLeft] > v[i]
        iMax ← iLeft
    else
        iMax ← i
    end if
    if iRight ≤ n and v[iRight] > v[iMax]
        iMax ← iRight
    end if
    if iMax ≠ i
        swap(v[i], v[iMax])
        call: maxheapify(v, iMax, n)
    end if
end function
```

Algoritmia (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

06 HEAP SORT
transformação-e-conquista

Utiliza o paradigma da transformação-e-conquista, utilizando uma diferente representação do mesmo problema.

Constrói-se uma heap de máximo a partir da lista.
Troca-se o primeiro com o último elemento da *heap*.
Retira-se o último elemento da *heap*.
Reorganiza-se a *heap* (*max-heapification*).
Repete-se até não haver elementos na *heap*.

Análise da reorganização (*heapify*) de uma *heap*

chamada recursivamente h vezes (no máximo), em que h é a altura da subárvore cuja raiz está em i

| | | | |
|---------------|---|-------------|--|
| log n vezes | no caso-pior realizam-se duas comparações entre elementos | e uma troca | function maxheapify |
| | | | pass in: ARRAY INT v, INT i, n var: INT iLeft, iRight, iMax iLeft ← 2 × i iRight ← 2 × i + 1 |
| | | | if iLeft ≤ n and v[iLeft] > v[i] iMax ← iLeft else iMax ← i end if if iRight ≤ n and v[iRight] > v[iMax] iMax ← iRight end if if iMax ≠ i swap(v[i], v[iMax]) call: maxheapify(v, iMax, n) |
| | | | end if end function |

comparações: $\leq 2 \times \log n$
trocas: $\leq \log n$

Algoritmia (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

06 HEAP SORT
transformação-e-conquista

Utiliza o paradigma da transformação-e-conquista, utilizando uma diferente representação do mesmo problema.

Constrói-se uma heap de máximo a partir da lista.
Troca-se o primeiro com o último elemento da *heap*.
Retira-se o último elemento da *heap*.
Reorganiza-se a *heap* (*max-heapification*).
Repete-se até não haver elementos na *heap*.

Análise da criação da *heap*

Uma análise rápida indica que a criação de uma *heap* tem uma complexidade temporal de $O(n \log n)$, uma vez que são feitas $O(n)$ chamadas de **maxheapify**, e cada uma dessas chamadas tem complexidade $O(\log n)$.

| | |
|---|--|
| Podemos, no entanto, obter um limite mais apertado deste valor. | function buildmaxheap |
| | pass in: ARRAY INT v var: INT n, i n ← call:length(v) repeat for i ← floor(n/2) downto 1 call: maxheapify(v, i, n) end repeat end function |

Algoritmia (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

133 utad
Pedro Melo-Pinto 2022

06 HEAP SORT

transformação-e-conquista

Utiliza o paradigma da transformação-e-conquista, utilizando uma diferente representação do mesmo problema.

Constrói-se uma heap de máximo a partir da lista.

Troca-se o primeiro com o último elemento da *heap*.
Retira-se o último elemento da *heap*.
Reorganiza-se a *heap* (*max-heapification*).
Repete-se até não haver elementos na *heap*.

Análise da criação da heap

Podemos obter um limite mais apertado da complexidade temporal:

buildmaxheap executa um ciclo em que o custo de cada chamada de **maxheapify** depende da altura h da subárvore em questão.

Uma *heap* de tamanho n tem, no máximo, $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nós com altura h . Logo:

$$\begin{aligned} T(n) &= \sum_{h=0}^{\lfloor \log_2(n) \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \times O(h) \\ &= O(n \times \sum_{h=0}^{\infty} \frac{n}{2^h}) \end{aligned}$$

Como $\sum_{n=0}^{\infty} x^n = \frac{1}{1-x}$, para $x < 1$, derivando ambos os lados e multiplicando por x obtemos $\sum_{n=0}^{\infty} nx^n = \frac{x}{(1-x)^2}$.

$$\begin{aligned} &= O\left(n \times \frac{1/2}{(1-1/2)^2}\right) \\ &= O(n) \end{aligned}$$

```
function buildmaxheap
    pass in: ARRAY INT v
    var:   INT n, i
    n ← call:length(v)
    repeat for i ← floor(n/2) down to 1
        call: maxheapify(v, i, n)
    end repeat
end function
```

Algoritmia (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

134 utad
Pedro Melo-Pinto 2022

06 HEAP SORT

transformação-e-conquista

Utiliza o paradigma da transformação-e-conquista, utilizando uma diferente representação do mesmo problema.

Constrói-se uma heap de máximo a partir da lista.

Troca-se o primeiro com o último elemento da *heap*.
Retira-se o último elemento da *heap*.
Reorganiza-se a *heap* (*max-heapification*).
Repete-se até não haver elementos na *heap*.

Análise da criação da heap

São feitas $O(n)$ chamadas de **maxheapify**, e cada uma dessas chamadas tem complexidade $O(\log n)$.

O número máximo de comparações necessárias é:

$$2 \times \sum_{k=0}^{d-1} 2^k(d - k - 1) = 2 \times (2^d - d - 1) = 2 \times (n - \lceil \log n \rceil)$$

comparações: $\leq 2 \times (n - \lceil \log n \rceil)$
trocas (1 para cada 2 comparações): $\leq n - \lceil \log n \rceil + 1$

```
function buildmaxheap
    pass in: ARRAY INT v
    var:   INT n, i
    n ← call:length(v)
    repeat for i ← floor(n/2) down to 1
        call: maxheapify(v, i, n)
    end repeat
end function
```

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

135 utad Pedro Melo-Pinto 2022

06 HEAP SORT

transformação-e-conquista

Utiliza o paradigma da transformação-e-conquista, utilizando uma diferente representação do mesmo problema.

Constrói-se uma heap de máximo a partir da lista.

Troca-se o primeiro com o último elemento da heap.

Retira-se o último elemento da heap.

Reorganiza-se a heap (*max-heapification*).

Repete-se até não haver elementos na heap.

Análise geral do desempenho

Baseia-se no paradigma de transformação-e-conquista.

Transforma o problema de ordenação de uma lista num problema de extração sucessiva da raiz de uma heap de máximo.

O processo tem dois passos principais:

- Criação da heap de máximo (*max-heap*)
Este passo tem tempo de execução:
 $C(n) = O(n)$.
- Extração sucessiva da raiz desta heap e subsequente reorganização
A extração executa em tempo constante $O(1)$, logo a complexidade depende da reorganização:
 $R(n) = O(n \log_2 n)$.

The diagram shows the transformation of an array into a max-heap. It consists of four rows. Row 1: A single node '7'. Row 2: Two nodes, '6' and '5'. Row 3: Three nodes, '6' (root), '4' (left child), and '3' (right child). Row 4: Seven nodes, '6' (root), '4' (left child), '3' (right child), and leaf nodes '2', '1', and three empty circles. This illustrates the process of building a max-heap from an array.

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

136 utad Pedro Melo-Pinto 2022

06 HEAP SORT

transformação-e-conquista

Utiliza o paradigma da transformação-e-conquista, utilizando uma diferente representação do mesmo problema.

Constrói-se uma heap de máximo a partir da lista.

Troca-se o primeiro com o último elemento da heap.

Retira-se o último elemento da heap.

Reorganiza-se a heap (*max-heapification*).

Repete-se até não haver elementos na heap.

Análise do desempenho

O tempo total de execução do algoritmo vai ser a soma do tempo de construção da heap mais o tempo de execução da troca e reorganização (*maxheapify*).

A determinação do tamanho da heap leva tempo constante e é ignorado.

```
function heapsort
    pass in: ARRAY INT v
    var: INT n,i
    call: buildmaxheap(v)
    n ← call.length(v)
    repeat for i ← n downto 2
        swap(v[1], v[i])
        call: maxheapify(v, 1, i-1)
    end repeat
end function
```

Algoritmia (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

137 utad Pedro Melo-Pinto 2022

06 HEAP SORT

transformação-e-conquista

Utiliza o paradigma da transformação-e-conquista, utilizando uma diferente representação do mesmo problema.

Constrói-se uma heap de máximo a partir da lista.
Trocá-se o primeiro com o último elemento da heap.
Retira-se o último elemento da heap.
Reorganiza-se a heap (*max-heapification*).
Repete-se até não haver elementos na heap.

Análise do desempenho

O tempo total de execução do algoritmo vai ser a soma do tempo de construção da heap mais o tempo de execução da troca e reorganização (*maxheapify*).
A determinação do tamanho da heap leva tempo constante e é ignorado.

Como visto, a construção da heap tem um desempenho $O(n)$.

```

function heapsort
    pass in: ARRAY INT v
    var: INT n,i
    call: buildmaxheap(v)
    n ← call:length(v)
    repeat for i ← n downto 2
        swap(v[1],v[i])
        call: maxheapify(v, 1, i-1)
    end repeat
end function

function buildmaxheap
    pass in: ARRAY INT v
    var: INT n,i
    n ← call:length(v)
    repeat for i ← floor(n/2) downto 1
        call: maxheapify(v, i, n)
    end repeat
end function

```

Algoritmia (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

138 utad Pedro Melo-Pinto 2022

06 HEAP SORT

transformação-e-conquista

Utiliza o paradigma da transformação-e-conquista, utilizando uma diferente representação do mesmo problema.

Constrói-se uma heap de máximo a partir da lista.
Trocá-se o primeiro com o último elemento da heap.
Retira-se o último elemento da heap.
Reorganiza-se a heap (*max-heapification*).
Repete-se até não haver elementos na heap.

Análise do desempenho da construção da heap

O tempo total de execução do algoritmo vai ser a soma do tempo de construção da heap mais o tempo de execução da troca e reorganização (*maxheapify*)

Vamos considerar $n = 2^{h+1} - 1$ pois o lado esquerdo da árvore comporta-se como se esta fosse perfeita.

O desempenho de *maxheapify* depende da altura do nó em questão.
As 2^h folhas não são chamadas.
Os 2^{h-1} nós do penúltimo nível podem ter de descer um nível.
Os 2^{h-2} nós do antepenúltimo nível podem ter de descer dois níveis, etc.

Em geral, num nível j a contar do fundo, existem 2^{h-j} nós que podem descer j níveis

$$T(n) = \sum_{j=0}^h j 2^{h-j} = \sum_{j=0}^h j \frac{2^h}{2^j} = 2^h \sum_{j=0}^h \frac{j}{2^j}$$

```

function buildmaxheap
    pass in: ARRAY INT v
    var: INT n,i
    n ← call:length(v)
    repeat for i ← floor(n/2) downto 1
        call: maxheapify(v, i, n)
    end repeat
end function

```

número de níveis que os nós poderão descer

Algoritmia (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

06 HEAP SORT
transformação-e-conquista

Utiliza o paradigma da transformação-e-conquista, utilizando uma diferente representação do mesmo problema.

Constrói-se uma heap de máximo a partir da lista.
Trocá-se o primeiro com o último elemento da heap.
Retira-se o último elemento da heap.
Reorganiza-se a heap (*max-heapification*).
Repete-se até não haver elementos na heap.

Análise do desempenho da construção da heap

O tempo total de execução do algoritmo vai ser a soma do tempo de construção da *heap* mais o tempo de execução da troca e reorganização (*maxheapify*).

Vamos considerar $n = 2^{h+1} - 1$ pois o lado esquerdo da árvore comporta-se como se esta fosse perfeita
Em geral, num nível j a contar do fundo, existem 2^{h-j} nós que podem descer j níveis

$$T(n) = \sum_{j=0}^h j 2^{h-j} = 2^h \sum_{j=0}^h \frac{j}{2^j}$$

Como $\sum_{j=0}^{\infty} x^j = \frac{1}{1-x}$, para $x < 1$, derivando (em relação a x) ambos os lados e multiplicando por x obtemos

$$T(n) = 2^h \sum_{j=0}^h \frac{j}{2^j} \leq 2^h \sum_{j=0}^{\infty} \frac{j}{2^j} \leq 2^h \times 2 = 2^{h+1}$$

Como $n = 2^{h+1} - 1$, $T(n) \leq n + 1$ ou seja $T(n) = O(n)$

function buildmaxheap
pass in: ARRAY INT v
var: INT n, i
 $n \leftarrow \text{call: length}(v)$
repeat for $i \leftarrow \text{floor}(n/2)$ down to 1
 call: maxheapify(v, i, n)
end repeat
end function

número de níveis que os nós poderão descer

Algoritmia (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

06 HEAP SORT
transformação-e-conquista

Utiliza o paradigma da transformação-e-conquista, utilizando uma diferente representação do mesmo problema.

Constrói-se uma heap de máximo a partir da lista.
Trocá-se o primeiro com o último elemento da heap.
Retira-se o último elemento da heap.
Reorganiza-se a heap (*max-heapification*).
Repete-se até não haver elementos na heap.

Análise do desempenho da construção da heap

O tempo total de execução do algoritmo vai ser a soma do tempo de construção da *heap* mais o tempo de execução da troca e reorganização (*maxheapify*).

Vamos considerar $n = 2^{h+1} - 1$ pois o lado esquerdo da árvore comporta-se como se esta fosse perfeita
Em geral, num nível j a contar do fundo, existem 2^{h-j} nós que podem descer j níveis

$$T(n) = \sum_{j=0}^h j 2^{h-j} = 2^h \sum_{j=0}^h \frac{j}{2^j}$$

$$T(n) = 2^h \sum_{j=0}^h \frac{j}{2^j} \leq 2^h \sum_{j=0}^{\infty} \frac{j}{2^j} \leq 2^h \times 2 = 2^{h+1}$$

Como $n = 2^{h+1} - 1$, $T(n) \leq n + 1$ ou seja $T(n) = O(n)$

O algoritmo demora pelo menos $\Omega(n)$ – tem de aceder pelo menos uma vez a todos os elementos da lista, logo $T(n) = \Theta(n)$

function buildmaxheap
pass in: ARRAY INT v
var: INT n, i
 $n \leftarrow \text{call: length}(v)$
repeat for $i \leftarrow \text{floor}(n/2)$ down to 1
 call: maxheapify(v, i, n)
end repeat
end function

número de níveis que os nós poderão descer

Algoritmia (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

141

06 HEAP SORT
transformação-e-conquista

Utiliza o paradigma da transformação-e-conquista, utilizando uma diferente representação do mesmo problema.

Constrói-se uma heap de máximo a partir da lista.
Trocá-se o primeiro com o último elemento da heap.
Retira-se o último elemento da heap.
Reorganiza-se a heap (*max-heapification*).
Repete-se até não haver elementos na heap.

Análise do desempenho da reorganização da heap

maxheapify é chamada recursivamente h vezes (no máximo), em que h é a altura da subárvore cuja raiz está em i

no caso-pior o número de nós da *subheap* do lado esquerdo é: $L = 2 \times 2^{h-1} - 1$, e o número de nós da *heap* é: $H = 2^h + 2^{h-1} - 1$, ou seja,
 $L = 2/3 n$

$T(n) \leq T(2n/3) + O(1)$
utilizando o caso 2 do Teorema Principal (Master Theorem)

$T(n) = O(\log n)$

```
function maxheapify
    pass in: ARRAY INT v, INT i, n
    var: INT iLeft, iRight, iMax
    iLeft ← 2 × i
    iRight ← 2 × i + 1
    if iLeft ≤ n and v[iLeft] > v[i]
        iMax ← iLeft
    else
        iMax ← i
    end if
    if iRight ≤ n and v[iRight] > v[iMax]
        iMax ← iRight
    end if
    if iMax[i] ≠ i
        swap(v[i], v[iMax])
        call: maxheapify(v, iMax, n)
    end if
end function
```

Pedro Melo-Pinto 2022

Algoritmia (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

142

06 HEAP SORT
transformação-e-conquista

| Número de Elementos (ordenado) | Inteiros | | | Strings | | |
|-----------------------------------|-------------|----------|-------|-------------|----------|-------|
| | Sel. e ins. | Inserção | Fusão | Sel. e ins. | Inserção | Fusão |
| 1000 | 64 | 4.0 | 354 | 179 | 8.3 | 854 |
| 10000 | 256 | 16.3 | 1591 | 762 | 36.5 | 2224 |
| 100000 | 1024 | 65.6 | 6359 | 3069 | 154.3 | 10162 |
| 1000000 | 4096 | 257.3 | 25761 | 12531 | 510.7 | 4096 |
| 10000000 | 16384 | 4.1 | 4.0 | 4.4 | 4.4 | 4.0 |
| 100000000 | 65536 | 16.4 | 16.1 | 17.0 | 18.6 | 18.3 |

| Número de Elementos (ordenado) | Inteiros | | | Strings | | |
|-----------------------------------|-------------|----------|-------|-------------|----------|-------|
| | Sel. e ins. | Inserção | Fusão | Sel. e ins. | Inserção | Fusão |
| 1000 | 66 | 0 | 0 | 273 | 0 | 0 |
| 10000 | 257 | 0 | 0 | 1090 | 0 | 0 |
| 100000 | 1024 | 0 | 0 | 4147 | 0 | 0 |
| 1000000 | 4096 | 0 | 0 | 3 | 2 | 2 |
| 10000000 | 16384 | — | — | 1.0 | — | — |
| 100000000 | 65536 | — | — | 15.9 | — | 15.2 |

| Número de Elementos (random) | Inteiros | | | Strings | | |
|----------------------------------|----------|-------|------|---------|-------|------|
| | Quick | Merge | Heap | Quick | Merge | Heap |
| 20000 | 7 | 8 | 27 | 14 | 18 | 45 |
| 50000 | 17 | 21 | 75 | 47 | 57 | 133 |
| 100000 | 39 | 45 | 161 | 109 | 137 | 310 |
| evolução de $n \times \log_2(n)$ | 2,73 | 2,6 | 2,8 | 3,4 | 3,2 | 3,0 |
| | 5,81 | 5,6 | 6,0 | 7,8 | 7,6 | 6,9 |

143

utad

Pedro Melo-Pinto 2022

E ordenação em tempo linear?

144

utad

Pedro Melo-Pinto 2022

07 RADIX SORT

Herman Hollerith , c.1901
Harold H. Seward, 1954 (computer version)

Algoritmia (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

145

utad
Pedro Melo-Pinto 2022

07 RADIX SORT

A radix sorting algorithm was originally used to sort punched cards in several passes.

Algoritmia (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

146

utad
Pedro Melo-Pinto 2022

07 RADIX SORT

Comecemos pelo método *Counting Sort*:
Este método não envolve comparações entre elementos

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

147
utad
Pedro Melo-Pinto 2022

07 RADIX SORT

Comecemos pelo método *Counting Sort*:

Este método não envolve comparações entre elementos

Pressupostos:
 n inteiros $\in [0 \dots k]$
 k é de ordem n , ou seja, $k = O(n)$

A Para cada elemento x , determinar o número de elementos menores ou iguais a x
B Colocar x na posição correcta no *array* de saída

exemplo:
 $x = 6$
 n° de elementos $\leq x = 4$

| | | | | | | | | |
|-----|---|---|---|---|---|---|---|---|
| 2 | 3 | 8 | 0 | 5 | 7 | 9 | 8 | 6 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| | | | | 6 | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| pos | | | | | | | | |

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

148
utad
Pedro Melo-Pinto 2022

07 RADIX SORT

Comecemos pelo método *Counting Sort*:

Este método não envolve comparações entre elementos

Pressupostos:
 n inteiros $\in [0 \dots k]$
 k é de ordem n , ou seja, $k = O(n)$

Para cada elemento x , determinar o número de elementos menores ou iguais a x
Colocar x na posição correcta no *array* de saída

1 Calcular as frequências dos elementos (histograma) do *array*
2 Calcular as frequências acumuladas dos elementos do *array*
3 Colocar os elementos do *array* na posição correcta no *array* de saída

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

149

utad
Pedro Melo-Pinto 2022

07 RADIX SORT

Comecemos pelo método *Counting Sort*:

Este método não envolve comparações entre elementos

Pressupostos:
 n inteiros $\in [0 \dots k]$
 k é de ordem n , ou seja, $k = O(n)$

Para cada elemento x , determinar o número de elementos menores ou iguais a x
Colocar x na posição correcta no array de saída

- ➊ Calcular as frequências dos elementos (histograma) do array
- ➋ Calcular as frequências acumuladas dos elementos do array
- ➌ Colocar os elementos do array na posição correcta no array de saída

exemplo:
 $x \in [0 \dots 9]$

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

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

Frequências dos elementos

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

150

utad
Pedro Melo-Pinto 2022

07 RADIX SORT

Comecemos pelo método *Counting Sort*:

Este método não envolve comparações entre elementos

Pressupostos:
 n inteiros $\in [0 \dots k]$
 k é de ordem n , ou seja, $k = O(n)$

Para cada elemento x , determinar o número de elementos menores ou iguais a x
Colocar x na posição correcta no array de saída

- ➊ Calcular as frequências dos elementos (histograma) do array
- ➋ Calcular as frequências acumuladas dos elementos do array
- ➌ Colocar os elementos do array na posição correcta no array de saída

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

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

Frequências dos elementos

Frequências acumuladas

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

151

utad
Pedro Melo-Pinto 2022

07 RADIX SORT

Comecemos pelo método *Counting Sort*:

Este método não envolve comparações entre elementos

Para cada elemento x , determinar o número de elementos menores ou iguais a x
Colocar x na posição correcta no *array* de saída

- ➊ Calcular as frequências dos elementos (histograma) do *array*
- ➋ Calcular as frequências acumuladas dos elementos do *array*
- ➌ Colocar os elementos do *array* na posição correcta no *array* de saída
 - Começar no último elemento
 - Colocar $A[i]$ no seu lugar correcto no *array* de saída
 - Decrementar a frequência acumulada de $A[i]$ em uma unidade

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

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

Frequências acumuladas

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

Saída

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

152

utad
Pedro Melo-Pinto 2022

07 RADIX SORT

Comecemos pelo método *Counting Sort*:

Este método não envolve comparações entre elementos

Para cada elemento x , determinar o número de elementos menores ou iguais a x
Colocar x na posição correcta no *array* de saída

- ➊ Calcular as frequências dos elementos (histograma) do *array*
- ➋ Calcular as frequências acumuladas dos elementos do *array*
- ➌ Colocar os elementos do *array* na posição correcta no *array* de saída
 - Começar no último elemento
 - Colocar $A[i]$ no seu lugar correcto no *array* de saída
 - Decrementar a frequência acumulada de $A[i]$ em uma unidade

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

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

Frequências acumuladas

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

Saída

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

153

utad
Pedro Melo-Pinto 2022

07 RADIX SORT

Comecemos pelo método *Counting Sort*:

Este método **não envolve comparações entre elementos**

Para cada elemento x , determinar o número de elementos menores ou iguais a x
Colocar x na posição correcta no *array* de saída

- ➊ Calcular as frequências dos elementos (histograma) do *array*
- ➋ Calcular as frequências acumuladas dos elementos do *array*
- ➌ Colocar os elementos do *array* na posição correcta no *array* de saída

Começar no último elemento
Colocar $A[i]$ no seu lugar correcto no *array* de saída
Decrementar a frequência acumulada de $A[i]$ em uma unidade

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

| | | | | | | | | | |
|---|---|-------|---|---|---|---|---|---|---|
| 0 | 1 | 2 → 1 | 2 | 3 | 3 | 4 | 5 | 6 | 8 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Frequências acumuladas

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

Saída

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

154

utad
Pedro Melo-Pinto 2022

07 RADIX SORT

Comecemos pelo método *Counting Sort*:

Este método **não envolve comparações entre elementos**

Para cada elemento x , determinar o número de elementos menores ou iguais a x
Colocar x na posição correcta no *array* de saída

- ➊ Calcular as frequências dos elementos (histograma) do *array*
- ➋ Calcular as frequências acumuladas dos elementos do *array*
- ➌ Colocar os elementos do *array* na posição correcta no *array* de saída

Começar no último elemento
Colocar $A[i]$ no seu lugar correcto no *array* de saída
Decrementar a frequência acumulada de $A[i]$ em uma unidade

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

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

Saída

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

155 utad Pedro Melo-Pinto 2022

07 RADIX SORT

Algoritmo

```

function counting_sort
    pass in: ARRAY INT v, f, out
    var: INT n, k
    // n: nº elementos
    // k: valor máximo dos elementos
    repeat for i ← 0 to k
        f[i] ← 0
    end repeat
    repeat for i ← 1 to n
        f[v[i]] ← f[v[i]] + 1
    end repeat
    repeat for i ← 1 to k
        f[i] ← f[i] + f[i-1]
    end repeat
    repeat for i ← n down to 1
        out(f[v[i]]) ← v[i]
        f[v[i]] ← f[v[i]] - 1
    end repeat
end function

```

| | |
|--|--------|
| inicialização do <i>array</i> de frequências | $O(k)$ |
| cálculo das frequências dos elementos | $O(n)$ |
| cálculo das frequências acumuladas | $O(k)$ |
| colocação dos elementos no <i>array</i> de saída | $O(n)$ |
| TOTAL: $O(n+k)$ | |

Como, na prática, se utiliza o método *Counting Sort* quando $k = O(n) \rightarrow$ complexidade do método é $O(n)$
Complexidade (memória): $O(n+k)$. É um método estável

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

156 utad Pedro Melo-Pinto 2022

07 RADIX SORT

Comecemos pelo método *Counting Sort*:
Este método não envolve comparações entre elementos

Complexidade: $O(n+k)$
Como, na prática, se utiliza o método *Counting Sort* quando $k = O(n) \rightarrow$ complexidade do método é $O(n)$
Complexidade (memória): $O(n+k)$
É um método estável

Its running time is $O(k + n)$ and, when k is $O(n)$, it is a linear-time sorting algorithm and beats every comparison-based algorithm

Porque não utilizamos sempre este método?

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

157

utad
Pedro Melo-Pinto 2022

07 RADIX SORT

Comecemos pelo método *Counting Sort*:
Este método não envolve comparações entre elementos

Complexidade: $O(n+k)$
Como, na prática, se utiliza o método *Counting Sort* quando $k = O(n) \rightarrow$ complexidade do método é $O(n)$

Complexidade (memória): $O(n+k)$
É um método estável

Dá para ordenar inteiros de 32 bits?

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

158

utad
Pedro Melo-Pinto 2022

07 RADIX SORT

As chaves de ordenação são representadas por números inteiros com d -dígitos (numa base k)
 $\text{chave} = x_1x_2x_3 \dots x_d$

Pressupostos:
 n inteiros (a ordenar) com d -dígitos
 cada dígito $\in [0 \dots k-1]$
 k é de ordem n , ou seja, $k = O(n)$
 $d = O(1)$

exemplos:
 $\text{chave}_{10} = 4078$ ($d=4$, base 10, $x \in [0,k-1]$) $x \in [0,9]$
 $\text{chave}_{10} = 25$ ($d=2$, base 10, $x \in [0,k-1]$) $x \in [0,9]$
 $\text{chave}_2 = 11101$ ($d=5$, base 2, $x \in [0,k-1]$) $x \in [0,1]$

Algoritmia (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

159

utad
Pedro Melo-Pinto 2022

07 RADIX SORT

As chaves de ordenação são representadas por números inteiros com d -dígitos (numa base k)
 $\text{chave} = x_1x_2x_3 \dots x_d$

Pressupostos:
 n inteiros com d -dígitos $\in [0 \dots k-1]$; k é de ordem n , ou seja, $k = O(n)$
 $d = O(1)$

O método Radix Sort ordena cada coluna dos elementos/números em sequência

- Ⓐ LSD (*Least Significant Digit*) – do algarismo menos significativo para o mais significativo
- Ⓑ MSD (*Most Significant Digit*) – do algarismo mais significativo para o menos significativo
(para, por exemplo, os casos de ordenação lexicográfica)

Algoritmia (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

160

utad
Pedro Melo-Pinto 2022

07 RADIX SORT

As chaves de ordenação são representadas por números inteiros com d -dígitos (numa base k)
 $\text{chave} = x_1x_2x_3 \dots x_d$

Pressupostos:
 n inteiros com d -dígitos $\in [0 \dots k-1]$; k é de ordem n , ou seja, $k = O(n)$
 $d = O(1)$

O método Radix Sort ordena cada coluna dos elementos/números em sequência

- Ⓐ LSD (*Least Significant Digit*) – do algarismo menos significativo para o mais significativo

Para um número com d dígitos:

1. Ordenar (*counting sort*) pelo algarismo menos significativo
2. Continuar, até todos os dígitos terem sido percorridos, a ordenar pelo dígito menos significativo seguinte

counting sort ou outro método estável

Algoritmia (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

161 utad Pedro Melo-Pinto 2022

07 RADIX SORT

As chaves de ordenação são representadas por números inteiros com d -dígitos (numa base k)
 $\text{chave} = x_1x_2x_3 \dots x_d$

Pressupostos:
 n inteiros com d -dígitos $\in [0 \dots k-1]$; k é de ordem n , ou seja, $k = O(n)$
 $d = O(1)$

O método Radix Sort ordena cada coluna dos elementos/números em sequência
 • LSD (*Least Significant Digit*) – do algarismo menos significativo para o mais significativo

Para um número com d dígitos:

1. Ordenar (*counting sort*) pelo algarismo menos significativo
2. Continuar, até todos os dígitos terem sido percorridos, a ordenar pelo dígito menos significativo seguinte

exemplo:
 $d=3$, base 10

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

Algoritmia (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

162 utad Pedro Melo-Pinto 2022

07 RADIX SORT

As chaves de ordenação são representadas por números inteiros com d -dígitos (numa base k)
 $\text{chave} = x_1x_2x_3 \dots x_d$

Pressupostos:
 n inteiros com d -dígitos $\in [0 \dots k-1]$; k é de ordem n , ou seja, $k = O(n)$
 $d = O(1)$

O método Radix Sort ordena cada coluna dos elementos/números em sequência
 • LSD (*Least Significant Digit*) – do algarismo menos significativo para o mais significativo

Para um número com d dígitos:

1. Ordenar (*counting sort*) pelo algarismo menos significativo
2. Continuar, até todos os dígitos terem sido percorridos, a ordenar pelo dígito menos significativo seguinte

exemplo:
 $d=3$, base 10

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

Algoritmia (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

163 utad Pedro Melo-Pinto 2022

07 RADIX SORT

As chaves de ordenação são representadas por números inteiros com d -dígitos (numa base k)
 $\text{chave} = x_1x_2x_3 \dots x_d$

Pressupostos:
 n inteiros com d -dígitos $\in [0 \dots k-1]$; k é de ordem n , ou seja, $k = O(n)$
 $d = O(1)$

O método Radix Sort ordena cada coluna dos elementos/números em sequência
 • LSD (*Least Significant Digit*) – do algarismo menos significativo para o mais significativo

Para um número com d dígitos:

1. Ordenar (*counting sort*) pelo algarismo menos significativo
2. Continuar, até todos os dígitos terem sido percorridos, a ordenar pelo dígito menos significativo seguinte

exemplo:
 $d=3$, base 10

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

Algoritmia (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

164 utad Pedro Melo-Pinto 2022

07 RADIX SORT

As chaves de ordenação são representadas por números inteiros com d -dígitos (numa base k)
 $\text{chave} = x_1x_2x_3 \dots x_d$

Pressupostos:
 n inteiros com d -dígitos $\in [0 \dots k-1]$; k é de ordem n , ou seja, $k = O(n)$
 $d = O(1)$

O método Radix Sort ordena cada coluna dos elementos/números em sequência
 • LSD (*Least Significant Digit*) – do algarismo menos significativo para o mais significativo

Para um número com d dígitos:

1. Ordenar (*counting sort*) pelo algarismo menos significativo
2. Continuar, até todos os dígitos terem sido percorridos, a ordenar pelo dígito menos significativo seguinte

exemplo:
 $d=3$, base 10

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

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

165 utad
Pedro Melo-Pinto 2022

07 RADIX SORT

Análise: geral

Qual o melhor método de ordenação das colunas/dígitos ?
Counting Sort parece ser uma escolha óbvia: $O(n+k)$ ou $O(n)$ quando $k = O(n)$ é constante

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

166 utad
Pedro Melo-Pinto 2022

07 RADIX SORT

Algoritmo

```

function radix_sort
    pass in: ARRAY INT v, f, out
    var: INT n, k, d
    // n: nº elementos
    // k: valor máximo dos elementos
    // d: nº de dígitos (fixo)

    repeat for j < 1 to d
        repeat for i < 0 to k
            f[i] <- 0
        end repeat
        repeat for i < 1 to n
            f[v[i] mod(10^j)] <- f[v[i] mod(10^j)] + 1
        end repeat
        repeat for i < 1 to k
            f[i] <- f[i] + f[i-1]
        end repeat
        repeat for i < n downto 1
            out(f[v[i] mod(10^j)]) <- v[i]
            f[v[i] mod(10^j)] <- f[v[i] mod(10^j)] - 1
        end repeat
        repeat for i < 1 to n
            v[i] <- out[i]
        end repeat
    end function

```

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

167

07 RADIX SORT n inteiros (a ordenar) com d dígitos
cada dígito $\in [0 \dots k-1]$

Análise: geral

Muito eficaz (quando aplicável).
Tempo expectável de execução : $O(d(n+k))$.

Não efectua a ordenação *in place*.
 $O(n+k)$ espaço extra

Estável (LSD).
Não Estável (MSD). pode ser implementado como estável com custo de memória

utad
Pedro Melo-Pinto 2022

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

168

07 RADIX SORT

Análise: eficácia/complexidade

Necessita de efectuar d passagens (uma por dígito) na lista de elementos.

Cada passagem leva um tempo $O(n+k)$, em que n é o número de elementos e k o número de valores possíveis para cada dígito (supondo que utilizamos o método Counting Sort).

Assumindo que:
 $k = O(n)$, $d = O(1)$

Total: $T = O(n)$

utad
Pedro Melo-Pinto 2022

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

169

07 RADIX SORT

Análise: eficácia/complexidade

Necessita de efectuar d passagens (uma por dígito) na lista de elementos.

Cada passagem leva um tempo $O(n+k)$, em que n é o número de elementos e k o número de valores possíveis para cada dígito (supondo que utilizamos o método *Counting Sort*).

Assumindo que:
 $k = O(n)$, $d = O(1) \rightarrow T = O(n)$

MAS, SERÁ QUE $d = O(1)$?

Se cada elemento $\in \{0, 1, \dots, n\}$, são necessários (aprox.) $d = \log_{10}n$ dígitos (no sistema decimal) para o representar
Cada chamada de *Counting Sort* leva $O(n+10) = O(n)$, pois $k = 10$

Então, o tempo total de execução é:
 $O(d(n+k)) = O(\log_{10}n \times n) = O(n \lg n)$

É pior do que o tempo esperado do método *Counting Sort* !!!

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

170

07 RADIX SORT

Análise: eficácia/complexidade

Necessita de efectuar d passagens (uma por dígito) na lista de elementos.

Cada passagem leva um tempo $O(n+k)$, em que n é o número de elementos e k o número de valores possíveis para cada dígito (supondo que utilizamos o método *Counting Sort*).

Assumindo que:
 $k = O(n)$, $d = O(1) \rightarrow T = O(n)$

Senão, o que fazer?

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

171

07 RADIX SORT

Análise: eficácia/complexidade

Necessita de efectuar d passagens (uma por dígito) na lista de elementos.

Cada passagem leva um tempo $O(n+k)$, em que n é o número de elementos e k o número de valores possíveis para cada dígito (supondo que utilizamos o método *Counting Sort*).

Assumindo que:

$$k = O(n), d = O(1) \rightarrow T = O(n)$$

Um elemento pode ser visto como uma palavra (de computador) de b bits

Exemplo: Palavra de 32 bits

Cada palavra (tamanho b) pode ser vista como tendo $\lceil b/r \rceil$ dígitos ($r \leq b$), em que cada dígito $\in [0, \dots, 2^r - 1]$

Exemplo: $b = 32, r = 8, k = 2^8 - 1 = 255, d$ (nº dígitos) $= \lceil b/r \rceil = 4$

Podemos também interpretar a palavra/elemento de d dígitos como representada numa base 2^r

O nº de vezes que o método *Counting Sort* tem de ser aplicado é, neste caso: $d = \lceil b/r \rceil = 4$
(ou, p.e., $d = 2$ se $r = 16$)

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

172

07 RADIX SORT

Análise: eficácia/complexidade

Necessita de efectuar d passagens (uma por dígito) na lista de elementos.

Cada passagem leva um tempo $O(n+k)$, em que n é o número de elementos e k o número de valores possíveis para cada dígito (supondo que utilizamos o método *Counting Sort*).

Assumindo que:

$$k = O(n), d = O(1) \rightarrow T = O(n)$$

Senão, como fazer?

Escolhermos uma base (2^r) em que cada dígito tenha uma gama maior de valores (d menor).

Chega?

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

173

utad
Pedro Melo-Pinto 2022

07 RADIX SORT

Análise: eficácia/complexidade

Necessita de efectuar d passagens (uma por dígito) na lista de elementos.

Cada passagem leva um tempo $O(n+k)$, em que n é o número de elementos e k o número de valores possíveis para cada dígito (supondo que utilizamos o método *Counting Sort*).

Assumindo que:

$$k = O(n), d = O(1) \rightarrow T = O(n)$$

Senão, como fazer?

Escolhemos uma base (2^r) em que cada dígito tenha uma gama maior de valores (d menor).

Considere que cada elemento $\in \{0, 1, \dots, 2^b - 1\}$, e pode ser representado por $b = dr$ bits (d partições de r bits cada)

Chamemos a cada partição um dígito (sistema de base 2^r)

Cada chamada de *Counting Sort* leva $O(n + 2^r)$

Logo:

$$T(n, b) : O(d(n + k)) = O\left(\frac{b}{r}(n + 2^r)\right)$$

melhor do que *Counting Sort* ?

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

174

utad
Pedro Melo-Pinto 2022

07 RADIX SORT

Análise: eficácia/complexidade

Necessita de efectuar d passagens (uma por dígito) na lista de elementos.

Cada passagem leva um tempo $O(n+k)$, em que n é o número de elementos e k o número de valores possíveis para cada dígito (supondo que utilizamos o método *Counting Sort*).

Assumindo que:

$$k = O(n), d = O(1) \rightarrow T = O(n)$$

Senão, como fazer?

Escolhemos uma base (2^r) em que cada dígito tenha uma gama maior de valores (d menor).

Considere que cada elemento $\in \{0, 1, \dots, 2^b - 1\}$, e pode ser representado por $b = dr$ bits (d partições de r bits cada)

Chamemos a cada partição um dígito (sistema de base 2^r)

Cada chamada de *Counting Sort* leva $O(n + 2^r)$

$T(n, b) : O(b/r(n + 2^r))$

Como conseguir que o método Radix Sort seja melhor do que o método Counting Sort?

Escolhendo r de tal modo que minimizemos $T(n, b)$

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

175

utad
Pedro Melo-Pinto 2022

07 RADIX SORT

Análise: eficácia/complexidade

Necessita de efectuar d passagens (uma por dígito) na lista de elementos.

Cada passagem leva um tempo $O(n+k)$, em que n é o número de elementos e k o número de valores possíveis para cada dígito (supondo que utilizamos o método *Counting Sort*).

Assumindo que:

$$k = O(n), d = O(1) \rightarrow T = O(n)$$

Senão, como fazer?

Escolhemos uma base (2^r) em que cada dígito tenha uma gama maior de valores (d menor).

Considere que cada elemento $\in \{0, 1, \dots, 2^b - 1\}$, e pode ser representado por $b = dr$ bits (d partições de r bits cada)

Chamemos a cada partição um dígito (sistema de base 2^r)

Cada chamada de *Counting Sort* leva $O(n + 2^r)$

$$T(n, b) : O(b/r(n + 2^r))$$

Como conseguir que o método Radix Sort seja melhor do que o método Counting Sort?

Escolhendo r de tal modo que minimizemos $T(n, b)$:

- ➊ aumentar r significa menos passagens (mas $T(\text{counting sort})$ aumenta)
- ➋ quando $r \gg \lg n$ ($2^r \gg n$), o tempo de execução cresce exponencialmente

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

176

utad
Pedro Melo-Pinto 2022

07 RADIX SORT

Análise: eficácia/complexidade

Necessita de efectuar d passagens (uma por dígito) na lista de elementos.

Cada passagem leva um tempo $O(n+k)$, em que n é o número de elementos e k o número de valores possíveis para cada dígito (supondo que utilizamos o método *Counting Sort*).

Assumindo que:

$$k = O(n), d = O(1) \rightarrow T = O(n)$$

Senão, como fazer?

$$T(n, b) : O(b/r(n + 2^r))$$

Escolhemos uma base (2^r) em que cada dígito tenha uma gama maior de valores (d menor) e em que r não seja muito maior do que $\lg n$

se $r = \lg n \rightarrow \text{Radix Sort executa em tempo linear}$

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

177

utad
Pedro Melo-Pinto 2022

07 RADIX SORT

Análise: eficácia/complexidade

Necessita de efectuar d passagens (uma por dígito) na lista de elementos.

Cada passagem leva um tempo $O(n+k)$, em que n é o número de elementos e k o número de valores possíveis para cada dígito (supondo que utilizamos o método *Counting Sort*).

Assumindo que:
 $r = \lg(n), k = 2^r \rightarrow T = O(n)$

Este método é rápido para grandes colecções
exemplo (números de 32 bits):
São necessárias, no máximo, 3 passagens para ordenar ≥ 2000 elementos
Os métodos *mergesort* ou *quicksort* necessitam de, pelo menos, $\lceil \lg 2000 \rceil = 11$ passagens

Não trabalha *in place*, precisa de memória extra (se utilizar *Counting Sort*).
Apresenta poucas características de localidade de referência espacial (*locality of reference*).
Assim, o método *quicksort* **bem implementado e afinado** apresenta melhor desempenho com hierarquias de memória como as existentes nos computadores actuais

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

178

utad
Pedro Melo-Pinto 2022

07 RADIX SORT

Ordenação de *strings*

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

07 RADIX SORT
ordenação de strings

Quantas comparações de caracteres são necessárias para comparar duas *strings* de comprimento W ?

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| p | e | r | f | e | i | t | t | o |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | |
| p | e | r | f | i | l | a | r | |

O tempo de execução é proporcional ao comprimento do maior prefixo comum

1. Proporcional a W (caso pior)
2. Normalmente sublinear em W

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

07 RADIX SORT
ordenação de strings

Radix: number of digits in alphabet

| name | r | $\lg r$ | characters |
|----------------|-----|---------|-----------------|
| Binary | 2 | 1 | 01 |
| Octal | 8 | 3 | 01234567 |
| Decimal | 10 | 4 | 0123456789 |
| Hexadecimal | 16 | 4 | 0123456789ABCDE |
| DNA | 4 | 2 | ACTG |
| ASCII | 128 | 7 | |
| Extended ASCII | 256 | 8 | |

Table of ASCII characters:

| Hex Dec Char | Hex Dec Char | Hex Dec Char | Hex Dec Char |
|------------------------------------|---------------|--------------|--------------|
| 0x00 0 NULL null | 0x20 32 Space | 0x40 64 # | 0x60 96 ^ |
| 0x01 1 SG0 Start of heading | 0x21 33 ! | 0x41 65 A | 0x61 97 a |
| 0x02 2 SG1 Start of text | 0x22 34 " | 0x42 66 B | 0x62 98 b |
| 0x03 3 RTX End of text | 0x23 35 # | 0x43 67 C | 0x63 99 c |
| 0x04 4 EOT End of transmission | 0x24 36 \$ | 0x44 68 D | 0x64 100 d |
| 0x05 5 ENQ Enquiry | 0x25 37 % | 0x45 69 E | 0x65 101 e |
| 0x06 6 ACK Acknowledge | 0x26 38 & | 0x46 6A F | 0x66 102 f |
| 0x07 7 BELL Bell | 0x27 39 ^ | 0x47 71 G | 0x67 103 g |
| 0x08 8 BS Backspace | 0x28 40 (| 0x48 72 H | 0x68 104 h |
| 0x09 9 TAB Horizontal tab | 0x29 41) | 0x49 73 I | 0x69 105 i |
| 0x0A 10 LF Line feed | 0x2A 42 , | 0x4A 74 J | 0x6A 106 j |
| 0x0B 11 VT Vertical tab | 0x2B 43 + | 0x4B 75 K | 0x6B 107 k |
| 0x0C 12 FF Form Feed | 0x2C 44 , | 0x4C 76 L | 0x6C 108 l |
| 0x0D 14 CR Carriage return | 0x2D 45 = | 0x4D 77 M | 0x6D 109 m |
| 0x0E 15 LS Line separator | 0x2E 46 ; | 0x4E 78 N | 0x6E 110 n |
| 0x0F 15 SI Shift in | 0x2F 47 / | 0x4F 79 O | 0x6F 111 o |
| 0x10 16 DEL Data link escape | 0x30 48 0 | 0x50 80 P | 0x70 112 p |
| 0x11 17 DC1 Device control 1 | 0x31 49 1 | 0x51 81 Q | 0x71 113 q |
| 0x12 18 DC2 Device control 2 | 0x32 4A 2 | 0x52 82 R | 0x72 114 r |
| 0x13 19 DC3 Device control 3 | 0x33 51 3 | 0x53 83 S | 0x73 115 s |
| 0x14 20 DC4 Device control 4 | 0x34 52 4 | 0x54 84 T | 0x74 116 t |
| 0x15 21 NAK Negative ack | 0x35 53 5 | 0x55 85 U | 0x75 117 u |
| 0x16 22 SYN Synchronous idle | 0x36 54 6 | 0x56 86 V | 0x76 118 v |
| 0x17 23 ETB End transmission block | 0x37 55 7 | 0x57 87 W | 0x77 119 w |
| 0x18 24 CAN Cancel | 0x38 56 8 | 0x58 88 X | 0x78 120 x |
| 0x19 25 EM Group separator | 0x39 57 9 | 0x59 89 Y | 0x79 121 y |
| 0x1A 26 SUB Substitute | 0x3A 58 9 | 0x5A 90 Z | 0x7A 122 z |
| 0x1B 27 FSC Escape | 0x3B 59 ; | 0x5B 91 { | 0x7B 123 { |
| 0x1C 28 FS File separator | 0x3C 60 < | 0x5C 92 \ | 0x7C 124 \ |
| 0x1D 29 GS Group separator | 0x3D 61 > | 0x5D 93 } | 0x7D 125 } |
| 0x1E 30 RS Record separator | 0x3E 62 ? | 0x5E 94 _ | 0x7F 126 _ |
| 0x1F 31 US Unit separator | 0x3F 63 ? | 0x5F 95 ~ | 0x7F 127 DEL |

Algoritmia (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

181

07 RADIX SORT
ordenação de strings

| | | | |
|-----------|-----------------|-------------|--------|
| algorithm | time complexity | extra space | stable |
| LSD sort | $W(n + r)$ | $n + r$ | YES |

Ⓐ LSD (Least Significant Digit) – do algarismo menos significativo para o mais significativo

exemplo:

ordenação estável (as setas não se cruzam)

| | | |
|---|---|---|
| d | a | b |
| a | d | d |
| c | a | b |
| f | a | d |
| f | e | e |
| b | a | d |
| d | a | d |
| b | e | e |
| f | e | d |
| b | e | d |
| e | b | b |
| a | c | e |

| | | |
|---|---|---|
| d | a | b |
| c | a | b |
| e | b | b |
| a | d | d |
| f | a | d |
| b | a | d |
| d | a | d |
| f | e | d |
| b | e | d |
| b | e | e |
| f | e | e |
| a | c | e |

| | | |
|---|---|---|
| a | c | e |
| a | d | d |
| b | a | d |
| b | e | d |
| b | e | e |
| c | a | b |
| d | a | b |
| d | a | d |
| e | b | b |
| f | e | d |
| f | e | e |
| f | a | d |

Algoritmia (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

182

07 RADIX SORT

E se as *strings* não tiverem todas o mesmo comprimento ?

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

183

07 RADIX SORT
ordenação de strings

| algorithm | time complexity | extra space | stable |
|-----------|-----------------|-------------|----------|
| LSD sort | $W(n+r)$ | $n+r$ | YES |
| MSD sort | $n \log n$ | $n+dr$ | NO (YES) |

di. profundidade da stack de chamada das funções
(comprimento do maior prefixo comum)

• MSD (Most Significant Digit) – do algarismo mais significativo para o menos significativo

- ① dividir a coleção em r partes segundo o primeiro caractere (*counting sort*)
- ② ordenar recursivamente as *strings* que começam pelo mesmo caractere

exemplo:

Algoritma (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

184

07 RADIX SORT
ordenação de strings

| algorithm | time complexity | extra space | stable |
|-----------|-----------------|-------------|----------|
| LSD sort | $W(n+r)$ | $n+r$ | YES |
| MSD sort | $n \log n$ | $n+dr$ | NO (YES) |

di. profundidade da stack de chamada das funções
(comprimento do maior prefixo comum)

• MSD (Most Significant Digit) – do algarismo mais significativo para o menos significativo

- ① dividir a coleção em r partes segundo o primeiro caractere (*counting sort*)
- ② ordenar recursivamente as *strings* que começam pelo mesmo caractere

Funciona com *strings* de comprimentos diferentes
(consegue-se parar a recursividade através da detecção do caractere de fim de *string*)

Porque razão tem o caractere de fim de *string* ser menor do que qualquer outro ?

Sedgewick, R. and Wayne, K., 2011
Algorithms, 4th edition

Algoritmia (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

185

07 RADIX SORT
ordenação de strings

• MSD (Most Significant Digit) – do algarismo mais significativo para o menos significativo

| algorithm | time complexity | extra space | stable |
|-----------|-----------------|-------------|----------|
| MSD sort | $n \log n$ | $n + dr$ | NO (YES) |

d: profundidade da stack de chamada das funções
(comprimento do maior prefixo comum)

Duas observações:

- Torna-se muito lento com subarrays pequenos
 - Cada chamada de função necessita do seu próprio array de frequências
 - ASCII (256 freqs): 100× mais lento do que a passagem de posicionamento para $n = 2$.
 - Unicode (65,536 freqs): 32 000× mais lento para $n = 2$.
- Grande número de subarrays pequenos devido à recursividade

Sedgewick, R. and Wayne, K., 2011
Algorithms, 4th edition

Pedro Melo-Pinto 2022

utad

Algoritmia (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

186

07 RADIX SORT
ordenação de strings

• MSD (Most Significant Digit) – do algarismo mais significativo para o menos significativo

| algorithm | time complexity | extra space | stable |
|-----------|-----------------|-------------|----------|
| MSD sort | $n \log n$ | $n + dr$ | NO (YES) |

d: profundidade da stack de chamada das funções
(comprimento do maior prefixo comum)

Duas observações:

- Torna-se muito lento com subarrays pequenos
- Grande número de subarrays pequenos devido à recursividade

Solução:
Mudança para o método de inserção para subarrays pequenos

Sedgewick, R. and Wayne, K., 2011
Algorithms, 4th edition

Algoritmia (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

187 utad Pedro Melo-Pinto 2022

07 RADIX SORT

ordenação de strings

• MSD (Most Significant Digit) – do algarismo mais significativo para o menos significativo

| algorithm | time complexity | extra space | stable |
|-----------|-----------------|-------------|----------|
| MSD sort | $n \log n$ | $n + dr$ | NO (YES) |

d: profundidade da stack de chamada das funções (comprimento do maior prefixo comum)

Sedgewick, R. and Wayne, K., 2011
Algoritmia, 4th edition

- ➊ O método MSD examina só os caracteres necessários à ordenação
- ➋ Este número depende das chaves (*strings*)
- ➌ O desempenho pode ser sublinear em n

nota: métodos de ordenação baseados em comparações tb podem ser sublineares em n

| Random (sublinear) | Non-random with duplicates (nearly linear) | Worst case (linear) |
|--------------------|--|---------------------|
| 1E10402 | are | 1DNB377 |
| 1HYL490 | by | 1DNB377 |
| 1R0Z572 | sea | 1DNB377 |
| 2HE734 | seashells | 1DNB377 |
| 2IVE230 | seashells | 1DNB377 |
| 2XOR846 | sells | 1DNB377 |
| 3CD8573 | sells | 1DNB377 |
| 3CVP720 | she | 1DNB377 |
| 3IG319 | she | 1DNB377 |
| 3KNA382 | shells | 1DNB377 |
| 3TAV879 | shore | 1DNB377 |
| 4CQP781 | surely | 1DNB377 |
| 4QGI284 | the | 1DNB377 |
| 4YHV229 | the | 1DNB377 |

Characters examined by MSD string sort

Algoritmia (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

188 utad Pedro Melo-Pinto 2022

07 RADIX SORT

ordenação de strings

• MSD (Most Significant Digit) – do algarismo mais significativo para o menos significativo

| algorithm | time complexity (garantido-caso pior) | time complexity (aleatório) | extra space | stable |
|-----------|---------------------------------------|-----------------------------|-------------|----------|
| MSD sort | $2W(n + r)$ | $n \log n$ | $n + dr$ | NO (YES) |

d: profundidade da stack de chamada das funções (comprimento do maior prefixo comum)
W: comprimento médio das strings (no LSD o tamanho W tem de ser fixo)

Desvantagens da ordenação MSD em *strings*:

- Necessita de espaço extra para cálculo das frequências
- Necessita de espaço auxiliar extra
- O ciclo interno tem muitas operações (com peso nas constantes)
- Acede a memória “aleatoriamente” (eventual uso ineficaz da cache).

vs.

Desvantagens da ordenação *quicksort* em *strings*:

- Número de comparações *linearithmic* (não linear).
- Tem de ver várias vezes muitos caracteres em *strings* com prefixes comuns compridos

Algoritmia (e Estruturas de Dados)
ALGORITMOS ORDENAÇÃO

189

utad
Pedro Melo-Pinto 2022

Leitura Adicional:

- ① Cormen T.H., Leiserson C.E., Rivest R.L. and Stein C., 2009.
Introduction to Algorithms 3rd Edition. **CAPÍTULOS 2, 6-8**
- ② Sedgewick R. and Wayne, K., 2011
Algorithms 4th Edition. **CAPÍTULO 2, SUBCAPÍTULO 5.1**
- ③ Adrego da Rocha A., 2014
Estruturas de Dados e Algoritmos em C 3^a Edição. **CAPÍTULO 5**
- ④ Adrego da Rocha A., 2014
Análise da Complexidade de Algoritmos. **CAPÍTULO 3**

