
Introdução à Computação II



Ordenação

Profa.: **Adriane Beatriz de Souza Serapião**

adriane@rc.unesp.br

Ordenação de dados

- ✦ **Ordenação ou classificação** de dados constitui uma das tarefas mais frequentes e importantes em processamento de dados, sendo, normalmente, auxiliar ou preparatória, visando tornar mais simples e eficientes as demais.
- ✦ **Ordenação de dados** é o processo pelo qual é determinada a ordem que as entradas de uma tabela (os registros de um arquivo) serão apresentadas de forma que obedeça a uma organização previamente determinada por um ou mais campos. O campo pelo qual o arquivo é ordenado é chamado de chave de ordenação/classificação.
- ✦ O ambiente de ordenação é determinado pelo meio onde estão os dados a serem classificados. Uma ordenação é considerada interna se os registros que ela classificar estiverem na memória principal, e externa se alguns dos registros que ela classificar estiverem no armazenamento auxiliar.

Ordenação de dados

- ⊕ A ordenação interna apresenta como principais características classificar apenas o que se encontra na memória principal, gastar o mesmo tempo de acesso para buscar qualquer dos endereços e o tempo médio de acesso não é afetado pela sequência dos dados.
- ⊕ Na ordenação externa, os dados são transferidos em blocos para a memória principal para, só então, serem manipulados. Esses dados transferidos num acesso a disco influenciarão na eficiência do processamento e os dados são manipulados bloco a bloco.
- ⊕ É importante notar que os métodos de ordenação externa envolvem a aplicação de métodos de ordenação interna, tomando a cada vez um subconjunto de dados a classificar. Por isso, nos deteremos ao estudo dos métodos de Ordenação Interna.

3

Ordenação de dados

- ⊕ Há diferentes formas de apresentação do resultado de ordenação, as três formas clássicas são:
 - ⊕ contiguidade física – as entradas são fisicamente rearranjadas, de forma que no final da ordenação a ordem lógica é igual a ordem física dos dados;
 - ⊕ vetor indireto de ordenação (VIO) – as entradas são mantidas fisicamente com as mesmas posições originais, sendo a sequência ordenada determinada por um vetor[VIO] que é gerado durante o processo de ordenação e que possui a sequência dos endereços ordenada pelos valores classificados da tabela associada; e
 - ⊕ encadeamento – as entradas não sofrem alterações em suas posições físicas, então, é formada uma lista encadeada que inclui todas as entradas da tabela ordenada pelo valor da chave de ordenação.

4

Ordenação por contiguidade física

- ✚ Nesta alternativa, as entradas são fisicamente rearranjadas, de modo que ao final tenhamos as entradas fisicamente ordenadas na sequência ditada pela chave de ordenação. Na figura abaixo é apresentada uma tabela não ordenada (tabela a) e a mesma tabela fisicamente ordenada após a classificação (tabela b).

Chave			Chave		
1	25	...	1	12	...
2	37	...	2	15	...
3	15	...	3	21	...
4	12	...	4	25	...
5	50	...	5	37	...
6	42	...	6	42	...
7	21	...	7	50	...

A B

- ✚ É importante observar que esta alternativa envolve a movimentação das entradas da tabela para suas novas posições na sequência ordenada, o que pode significar um custo bastante elevado se estas entradas forem grandes.

5

Ordenação por vetor indireto de ordenação

- ✚ Neste caso, as entradas são mantidas em suas posições originais, sendo a sequência ordenada especificada por meio de um vetor indireto de ordenação (VIO), que é gerado durante o processo de ordenação. Na figura abaixo é apresentada uma tabela, juntamente com seu vetor indireto de ordenação.

Chave			VIO	
1	25	...	1	4
2	37	...	2	3
3	15	...	3	7
4	12	...	4	1
5	50	...	5	2
6	42	...	6	6
7	21	...	7	5

- ✚ A principal vantagem desta alternativa é que ela não envolve a movimentação das entradas da tabela de suas posições originais. Além disto, ainda é possível classificar-se a mesma tabela através de vários VIO's diferentes.

6

Ordenação por encadeamento

- ✦ Neste método, as entradas da tabela são também mantidas em suas posições originais, sendo formada uma lista encadeada que inclui todas as entradas da tabela ordenada pelo valor da chave de ordenação.
- ✦ Esta é uma alternativa parecida com o VIO, com a diferença de que não se é utilizado um vetor em separado, mas sim um campo adicional em cada entrada, para indicar a sequência ordenada. Em cada entrada, o conteúdo deste campo (*próximo*) é igual ao endereço da próxima entrada na ordem de classificação.

Primeiro		Chave		Proximo
4	1	25	...	2
	2	37	...	6
	3	15	...	7
	4	12	...	3
	5	50	...	0
	6	42	...	5
	7	21	...	1

7

Ordenação por encadeamento

- ✦ O endereço da primeira entrada na sequência ordenada pode ser indicado em uma posição (*primeiro*) fora da tabela, ou, como alternativa, pode-se reservar o primeiro endereço da tabela para armazenar esta informação.
- ✦ Similar ao método do VIO, também o encadeamento permite a ordenação da tabela por vários critérios diferentes, com o uso de campos adicionais para o encadeamento. Um exemplo disto é mostrado na tabela abaixo, que possui dois campos adicionais para ordenação, cada um ordenando a tabela segundo um critério diferente.

	Chave1	Chave2		Prox1	Prox2
1	--	--	...	6	4
2	37	1450	...	7	7
3	85	1880	...	0	6
4	28	1230	...	9	10
5	19	1750	...	4	3
6	12	1937	...	10	0
7	49	1520	...	8	8
8	63	1630	...	3	5
9	33	1427	...	2	2
10	18	1325	...	5	9

8

Classificação dos métodos de ordenação interna

Os algoritmos de ordenação são frequentemente classificados usando diferentes métricas:

- ⊕ **Complexidade computacional:** a classificação é baseada no pior, no médio e no melhor comportamento de ordenação de uma lista de tamanho (n).
 - ⊕ Para os algoritmos de ordenação típicos o comportamento aceitável/bom é $O(n \log n)$ e o comportamento inaceitável/ruim é $O(n^2)$.
 - ⊕ O comportamento ideal para uma ordenação é $O(n)$.
- ⊕ **Uso de memória (e uso de outros recursos do computador):**
 - ⊕ Alguns algoritmos de ordenação são "*in place*", de modo que apenas memória $O(1)$ ou $O(\log n)$ é necessária além dos itens a serem ordenados.
 - ⊕ Outros precisam criar estruturas de dados auxiliares para os dados serem temporariamente armazenados. Mergesort precisa de mais recursos de memória, logo não é um algoritmo "*in place*", ao passo que quicksort e heapsort são "*in place*". Radix e bucket sorts não são "*in place*".

9

Classificação dos métodos de ordenação interna

Os algoritmos de ordenação são frequentemente classificados usando diferentes métricas (cont.):

- ⊕ **Recursão:** os algoritmos são recursivos ou não-recursivos (e.g., mergesort é recursivo).
- ⊕ **Estabilidade:** algoritmos de ordenação estáveis mantêm a ordem relativa dos elementos/registros com chaves/valores iguais. Radix e bucket sorts são estáveis.
- ⊕ **Método geral:** a classificação é baseada em como a ordenação funciona internamente.
 - ⊕ Os métodos usados internamente incluem inserção, troca, seleção, intercalação, distribuição, etc.

Métodos de ordenação interna

- ✦ Ordenação por troca: efetuam a ordenação por comparação entre pares de chaves, trocando-as de posição caso estejam fora de ordem.
- ✦ Ordenação por seleção: selecionam, a cada iteração, a chave de menor (ou maior) valor da tabela, colocando-a em sua posição correta (início ou final) dentro da tabela por permutação com a chave que ocupa aquela posição.
- ✦ Ordenação por inserção: dividem a tabela em dois segmentos, sendo o 1º ordenado e o 2º não ordenado. A seguir, todos os elementos do 2º segmento vão, um a um, sendo inseridos no 1º segmento.
- ✦ Ordenação por intercalação: divide a tabela em dois ou mais segmentos, ordena estes segmentos e depois os intercalam, terminando, ao final, com um único segmento (toda a tabela) ordenado. O principal algoritmo desta família é o *mergesort*.
- ✦ Ordenação por distribuição: cada elemento é sucessivamente, distribuído em slots (escaninhos) conforme o valor dos dígitos de sua chave. O principal algoritmo desta família é o *radixsort*.¹¹

Métodos de ordenação interna

- ✦ a) Ordenação por troca
 - ✦ *Bubble Sort* (método da bolha)
 - ✦ *Shaker Sort (Cocktail Sort)* (método oscilante)
 - ✦ *Quick Sort* (método rápido)
- ✦ b) Ordenação por seleção
 - ✦ *Selection Sort* (método da seleção direta)
 - ✦ *Heap Sort*
- ✦ c) Ordenação por inserção
 - ✦ *Insertion Sort* (método da inserção direta)
 - ✦ *Shell Sort* (método dos incrementos decrescentes)
- ✦ d) Ordenação por intercalação
 - ✦ *Merge Sort*
 - ✦ *Merge Array*
- ✦ e) Ordenação por distribuição
 - ✦ *Counting Sort* (método da contagem)
 - ✦ *Bucket Sort (Bin Sort)* (método da distribuição de chaves)
 - ✦ *Radix Sort (Digital Sort)* (método da raiz)

Métodos de ordenação interna

- ⊕ Ordenadores por comparação: examinam elementos com um operador de comparação, o qual geralmente é o operador menor ou igual (\leq). Incluem:
 - ⊕ *Bubble sort*
 - ⊕ *Shaker sort*
 - ⊕ *Insertion sort*
 - ⊕ *Selection sort*
 - ⊕ *Shell sort*
 - ⊕ *Heapsort*
 - ⊕ *Mergesort*
 - ⊕ *Quicksort*
- ⊕ Ordenadores de não-comparação: usam outras técnicas para ordenar dados, assim como usam operadores de comparação. Incluem:
 - ⊕ *Counting sort* (indexa usando valores das chaves)
 - ⊕ *Bucket sort* (examina bits de chaves)
 - ⊕ *Radix sort* (examina bits individuais de chaves)

13

Ordenação

⊕ Considerações Iniciais

⊕ **Ordenação:**

- ✧ Arranjo de uma série de itens similares em ordem *crescente* ou *decrescente*.
- ⊕ Seja uma lista ordenada L de n itens:
 - ✧ Cada item é denominado *registro* (em linguagem C, *struct*).
 - ✧ Possibilidade de associação de uma *chave* ($c[k]$) a cada registro ($r[k]$).

14

Ordenação

⌘ Considerações Iniciais

- ⊕ Condição de ordenação do vetor pela chave:

Se j precedendo k implicar $v[j] < v[k]$ (ou $v[j] > v[k]$) em alguma ordenação nas chaves.

15

Ordenação

⌘ Avaliação de Algoritmos de Ordenação - Critérios Gerais:

- ⊕ Velocidade de ordenação das informações no ***melhor caso***, no ***caso médio*** e no ***pior caso***.
- ⊕ Comportamento do algoritmo (***natural*** ou ***não natural***).
- ⊕ Capacidade de rearranjo de elementos com ***chaves*** iguais.

16

Métodos de ordenação interna

- ⊞ a) Ordenação por troca
 - ⊕ *Bubble Sort* (método da bolha)
 - ⊕ *Shaker Sort (Cocktail Sort)* (método oscilante)
 - ⊕ *Quick Sort* (método rápido)
- ⊞ b) Ordenação por seleção
 - ⊕ *Selection Sort* (método da seleção direta)
 - ⊕ *Heap Sort*
- ⊞ c) Ordenação por inserção
 - ⊕ *Insertion Sort* (método da inserção direta)
 - ⊕ *Shell Sort* (método dos incrementos decrescentes)
- ⊞ d) Ordenação por intercalação
 - ⊕ *Merge Sort*
 - ⊕ *Merge Array*
- ⊞ e) Ordenação por distribuição
 - ⊕ *Counting Sort* (método da contagem)
 - ⊕ *Bucket Sort (Bin Sort)* (método da distribuição de chaves)
 - ⊕ *Radix Sort (Digital Sort)* (método da raiz)

17

Ordenação por troca

⊞ Ordenação *Borbulhante (Bubble Sort)*

⊕ Ordenação por *troca*:

- ✧ Repetição de comparações e, caso necessária, troca de itens adjacentes.
- ✧ Método mais conhecido (e difamado).

⊕ Idéia Básica:

- ✧ Permuta de itens entre posições consecutivas.
 - ✧ “ Borbulhamento ” de valores mais elevados (ou mais baixos) para o final (ou para o início) do arranjo.

18

Ordenação por troca

✚ Ordenação *Borbulhante* (*Bubble Sort*)

✚ Considere-se v um vetor de inteiros do qual os n primeiros itens devam ser ordenados de modo que $v[i] \leq v[j]$ para $1 \leq i < j \leq n$.

✧ Fundamentação do método:

- ✧ Percurso sequencial da lista de itens sucessivas vezes.
- ✧ Comparação de cada item com seu sucessor ($v[i]$ com $v[i+1]$).

✚ Troca dos itens, caso não estejam na ordem correta.

19

Ordenação por troca

✚ Ordenação *Borbulhante* (*Bubble Sort*)

✚ Exemplo 01 – Ordenação da sequência

13, 5, 1, 0, 3, 2, 8, 1

1ª I T E R A Ç Ã O	Comparação	Antes	Depois	Troca
	$v[1]$ com $v[2]$	13 e 5	5 e 13	●
	$v[2]$ com $v[3]$	13 e 1	1 e 13	●
	$v[3]$ com $v[4]$	13 e 0	0 e 13	●
	$v[4]$ com $v[5]$	13 e 3	3 e 13	●
	$v[5]$ com $v[6]$	13 e 2	2 e 13	●
	$v[6]$ com $v[7]$	13 e 8	8 e 13	●
	$v[7]$ com $v[8]$	13 e 1	1 e 13	●

20

Ordenação por troca

✚ Ordenação *Borbulhante* (*Bubble Sort*)

✚ Exemplo 01 – Ordenação da sequência

5, 1, 0, 3, 2, 8, 1, 13

2ª I T E R A Ç Ã O	Comparação	Antes	Depois	Troca
	$v[1]$ com $v[2]$	5 e 1	1 e 5	●
	$v[2]$ com $v[3]$	5 e 0	0 e 5	●
	$v[3]$ com $v[4]$	5 e 3	3 e 5	●
	$v[4]$ com $v[5]$	5 e 2	2 e 5	●
	$v[5]$ com $v[6]$	5 e 8	5 e 8	
	$v[6]$ com $v[7]$	8 e 1	1 e 8	●
	$v[7]$ com $v[8]$	8 e 13	8 e 13	

21

Ordenação por troca

✚ Ordenação *Borbulhante* (*Bubble Sort*)

✚ Exemplo 01 – Ordenação da sequência

1, 0, 3, 2, 5, 1, 8, 13

3ª I T E R A Ç Ã O	Comparação	Antes	Depois	Troca
	$v[1]$ com $v[2]$	1 e 0	0 e 1	●
	$v[2]$ com $v[3]$	1 e 3	1 e 3	
	$v[3]$ com $v[4]$	3 e 2	2 e 3	●
	$v[4]$ com $v[5]$	3 e 5	3 e 5	
	$v[5]$ com $v[6]$	5 e 1	1 e 5	●
	$v[6]$ com $v[7]$	5 e 8	5 e 8	
	$v[7]$ com $v[8]$	8 e 13	8 e 13	

22

Ordenação por troca

✚ Ordenação *Borbulhante* (*Bubble Sort*)

✚ Exemplo 01 – Ordenação da sequência

0, 1, 2, 3, 1, 5, 8, 13

4ª	Comparação	Antes	Depois	Troca
I T E R A Ç Ã O	v[1] com v[2]	0 e 1	0 e 1	
	v[2] com v[3]	1 e 2	1 e 2	
	v[3] com v[4]	2 e 3	2 e 3	
	v[4] com v[5]	3 e 1	1 e 3	●
	v[5] com v[6]	3 e 5	3 e 5	
	v[6] com v[7]	5 e 8	5 e 8	
	v[7] com v[8]	8 e 13	8 e 13	

23

Ordenação por troca

✚ Ordenação *Borbulhante* (*Bubble Sort*)

✚ Exemplo 01 – Ordenação da sequência

0, 1, 2, 1, 3, 5, 8, 13

5ª	Comparação	Antes	Depois	Troca
I T E R A Ç Ã O	v[1] com v[2]	0 e 1	0 e 1	
	v[2] com v[3]	1 e 2	1 e 2	
	v[3] com v[4]	2 e 1	1 e 2	●
	v[4] com v[5]	2 e 3	2 e 3	
	v[5] com v[6]	3 e 5	3 e 5	
	v[6] com v[7]	5 e 8	5 e 8	
	v[7] com v[8]	8 e 13	8 e 13	

24

Ordenação por troca

✚ Ordenação *Borbulhante* (*Bubble Sort*)

✚ Exemplo 01 – Ordenação da sequência

0, 1, 1, 2, 3, 5, 8, 13

6ª I T E R A Ç Ã O	Comparação	Antes	Depois	Troca
	$v[1]$ com $v[2]$	0 e 1	0 e 1	
	$v[2]$ com $v[3]$	1 e 1	1 e 1	
	$v[3]$ com $v[4]$	1 e 2	1 e 2	
	$v[4]$ com $v[5]$	2 e 3	2 e 3	
	$v[5]$ com $v[6]$	3 e 5	3 e 5	
	$v[6]$ com $v[7]$	5 e 8	5 e 8	
	$v[7]$ com $v[8]$	8 e 13	8 e 13	

25

Ordenação por troca

✚ Ordenação *Borbulhante* (*Bubble Sort*)

✚ Constatação \Rightarrow Após a 1ª iteração, o ***maior*** elemento assume a posição correta.

✚ Em geral, $v[n-i]$ assumirá a posição correta após a iteração i .

✚ Simplicidade algorítmica x Velocidade de processamento.

✧ Uma das piores formas de ordenação.

26

Ordenação por troca

⊕ Algoritmo: Ordenação *Borbulhante* (*Bubble Sort*)

- ⊕ Seja v um vetor de elementos tidos como desordenados.
- ⊕ Repita até que o vetor esteja ordenado:
 - ✧ Percorra o vetor da esquerda para a direita, verificando se cada par de elementos (i,j) de posições consecutivas está ordenado, ou seja, se “ i ” vem antes de “ j ” na ordem desejada.
 - ✧ Se o par não estiver ordenado, troque a posição de ambos os seus elementos.

27

Ordenação por troca

⊕ Ordenação *Borbulhante* (*Bubble Sort*)

⊕ Programa 01

```
Borbulhante01(int i[ ], int n) {  
    int iteracao, k, auxi;  
    for (iteracao = 1; iteracao <= n; iteracao++)  
        for (k=1; k < n; k++)  
            if (i[k-1] > i[k]){  
                auxi = i[k-1];  
                i[k-1] = i[k];  
                i[k] = auxi;  
            }  
}
```

28

Ordenação por troca

⊞ Ordenação *Borbulhante* (*Bubble Sort*)

⊞ Programa 02 – Ripple Sort

```
Borbulhante02(char *item, int n) {  
    int iteracao, k;  
    char t;  
    for (iteracao = 1; iteracao < n; iteracao++)  
        for (k=n-1; k > =iteracao; k--)  
            if (item [k-1] > item[k]){  
                auxi = item[k-1];  
                item[k-1] = item[k];  
                item[k] = auxi;  
            }  
}
```

29

Ordenação em C XVI

⊞ Ordenação *Borbulhante* (*Bubble Sort*)

⊞ Programa 02 – Ripple Sort

```
#include "string.h"  
#include "stdio.h"  
#include "stdlib.h"  
  
void Borbulhante02(char *item, int cont);  
  
void main(void)  
{  
    char s[80];  
    printf("Digite uma string:");  
    gets(s);  
    Borbulhante02(s, strlen(s));  
    printf("A string ordenada eh: %s\n", s);  
}
```

30

Ordenação por troca

✚ Ordenação *Borbulhante* (*Bubble Sort*)

✚ Considerações Finais:

- ✧ Número de comparações $\Rightarrow \frac{1}{2}(n^2 - n)$.
- ✧ Número de trocas no melhor caso $\Rightarrow 0$.
- ✧ Número de trocas no caso médio $\Rightarrow \frac{3}{4}(n^2 - n)$.
- ✧ Número de trocas no pior caso $\Rightarrow \frac{3}{2}(n^2 - n)$.

31

Ordenação por troca

✚ Ordenação (*Shaker Sort*)

✚ Melhoramento do Método de Ordenação Borbulhante (*Bubble Sort*)

- ✧ A idéia deste algoritmo é mesclar as duas formas do método *bubble sort*.
- ✧ Este algoritmo tem 2 etapas:
 - ✧ a primeira etapa consiste em trasladar os elementos menores até a parte esquerda do array, guardando em uma variável a posição do último elemento trocado.
 - ✧ A segunda etapa consiste em levar os elementos maiores até a direita do array, guardando a posição do último elemento trocado em outra variável.

32

Ordenação por troca

✚ Ordenação (*Shaker Sort*)

- ✚ Também conhecido como *bidirectional bubble sort*, *cocktail shaker sort*, *cocktail sort* (que também pode ser referenciado como uma variação do *selection sort*), *ripple sort*, *shuttle sort* ou *happy hour sort*.

33

Ordenação por troca

✚ Ordenação *Shaker Sort*

- ✚ Exemplo 02 – Direita para esquerda

15, 67, 8, 16, 44, 27, 12, 35

1ª I T E R A Ç Ã O	Comparação	Antes	Depois	Troca
	$v[8]$ com $v[7]$	35 e 12	35 e 12	
	$v[7]$ com $v[6]$	12 e 27	27 e 12	●
	$v[6]$ com $v[5]$	12 e 44	44 e 12	●
	$v[5]$ com $v[4]$	12 e 16	16 e 12	●
	$v[4]$ com $v[3]$	12 e 8	12 e 8	
	$v[3]$ com $v[2]$	8 e 67	67 e 8	●
	$v[2]$ com $v[1]$	8 e 15	15 e 8	●

34

Ordenação por troca

✚ Ordenação *Shaker Sort*

✚ Exemplo 02 – Esquerda para direita ESQ = 3

8, 15, 67, 12, 16, 44, 27, 35

1ª I T E R A Ç Ã O	Comparação	Antes	Depois	Troca
	$v[2]$ com $v[3]$	15 e 67	15 e 67	
	$v[3]$ com $v[4]$	67 e 12	12 e 67	●
	$v[4]$ com $v[5]$	67 e 16	16 e 67	●
	$v[5]$ com $v[6]$	67 e 44	44 e 67	●
	$v[6]$ com $v[7]$	67 e 27	27 e 67	●
	$v[7]$ com $v[8]$	67 e 35	35 e 67	●

35

Ordenação por troca

✚ Ordenação *Shaker Sort*

✚ Exemplo 02 – Direita para esquerda DIR = 7

8, 15, 12, 16, 44, 27, 35, 67

2ª I T E R A Ç Ã O	Comparação	Antes	Depois	Troca
	$v[7]$ com $v[6]$	35 e 27	35 e 27	
	$v[6]$ com $v[5]$	27 e 44	44 e 27	●
	$v[5]$ com $v[4]$	27 e 16	27 e 16	
	$v[4]$ com $v[3]$	16 e 12	16 e 12	
	$v[3]$ com $v[2]$	12 e 15	15 e 12	●

36

Ordenação por troca

✚ Ordenação *Shaker Sort*

✚ Exemplo 02 – Esquerda para direita ESQ = 4

8, 12, 15, 16, 27, 44, 35, 67

2ª I T E R A Ç Ã O	Comparação	Antes	Depois	Troca
	$v[3]$ com $v[4]$	15 e 16	27 e 35	
	$v[4]$ com $v[5]$	16 e 27	16 e 27	
	$v[5]$ com $v[6]$	27 e 44	27 e 44	
	$v[6]$ com $v[7]$	44 e 35	35 e 44	●

37

Ordenação por troca

✚ Ordenação *Shaker Sort*

✚ Exemplo 02 – Direita para esquerda DIR = 6

8, 12, 15, 16, 27, 35, 44, 67

3ª I T E R A Ç Ã O	Comparação	Antes	Depois	Troca
	$v[6]$ com $v[5]$	35 e 27	35 e 27	
	$v[5]$ com $v[4]$	27 e 16	27 e 16	
	$v[4]$ com $v[3]$	16 e 15	16 e 15	

38

Ordenação por troca

✚ Ordenação *Shaker Sort*

✚ Exemplo 02 – Esquerda para direita ESQ = 7

8, 12, 15, 16, 27, 35, 44, 67

O algoritmo termina, já que não se produzem trocas e a condição $ESQ > DIR$, se faz verdadeira.

39

Ordenação por troca

✚ Ordenação *Oscilante (Shaker Sort)*

```
/* Ordenação Oscilante */  
  
void Oscilante( int item[], int cont)  
{  
    int a;  
    int troca;  
    int auxi;  
    do {  
        troca = 0;  
        for(a= cont - 1; a>0; -- a) {  
            if(item[a - 1]>item[a]) {  
                t=item[a - 1];  
                item[a - 1]=item[a];  
                item[a]=auxi;  
                troca=1;  
            }  
        }  
        for(a=1; a<cont; ++a) {  
            if(item[a - 1]>item[a]) {  
                auxi=item[a - 1];  
                item[a - 1]=item[a];  
                item[a]=auxi;  
                troca=1;  
            }  
        }  
    } while (troca); /* Ordenacao até que não existam mais trocas*/  
}
```

40

Ordenação por troca

✚ Ordenação (*Shaker Sort*)

- ✚ Insignificamente diferente do *bubble sort*.
- ✚ O pior caso ocorre com pouca frequência.
- ✚ O trabalho adicional de manipular os elementos complica cada execução.
- ✚ A complexidade algorítmica é $O(N^2)$.

41

Ordenação por troca

✚ Ordenação Rápida (*Quicksort*)

✚ Considerações Iniciais

- ✧ Superior aos demais métodos de ordenação \Rightarrow Algoritmo de ordenação mais rápido conhecido *na prática* (métodos de ordenação baseados em endereços podem ser mais rápidas).
- ✧ Concepção \Rightarrow C. A. R. Hoare, em 1962.
- ✧ Idéia Básica \Rightarrow Divisão para conquista.

42

Ordenação por troca

Ordenação Rápida (*Quicksort*)

⊕ Considerações Iniciais

✧ Fundamentação \Rightarrow Método de ordenação por trocas e na idéia de partições:

- ✧ Seleção de um valor (comparador) e partição da sequência em duas seções, com todos os elementos maiores ou iguais ao valor da partição de um lado e os menores do outro.
- ✧ Repetição do processo para cada seção restante até a ordenação completa da matriz.

43

Ordenação por troca

Ordenação Rápida (*Quicksort*)

⊕ Considerações Iniciais

✧ Procedimento Básico:

- ✧ Se o número de itens a ser ordenado for **0** ou **1**, retorno da informação.
- ✧ Caso contrário, consideração de um item qualquer da sequência $v[n]$ como elemento particionador ou *pivô* (x).
- ✧ Partição da sequência de itens em 2 conjuntos disjuntos, **S1** ($v[n] \leq x$) e **S2** ($v[n] > x$).
- ✧ Retorno de *QuickSort* (**S1**) seguido de x e de *QuickSort* (**S2**).

44

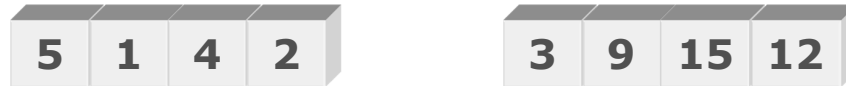
Ordenação por troca

✚ Ordenação Rápida (*Quicksort*)

✚ Exemplo 03 – Ordenação da sequência

5, 1, 4, 2, 10, 3, 9, 15, 12

Considerando o item central (10) como pivô e particionando a sequência conforme o procedimento do slide *anterior*



Ordenando os subconjuntos



Recombinando os subconjuntos com o pivô:



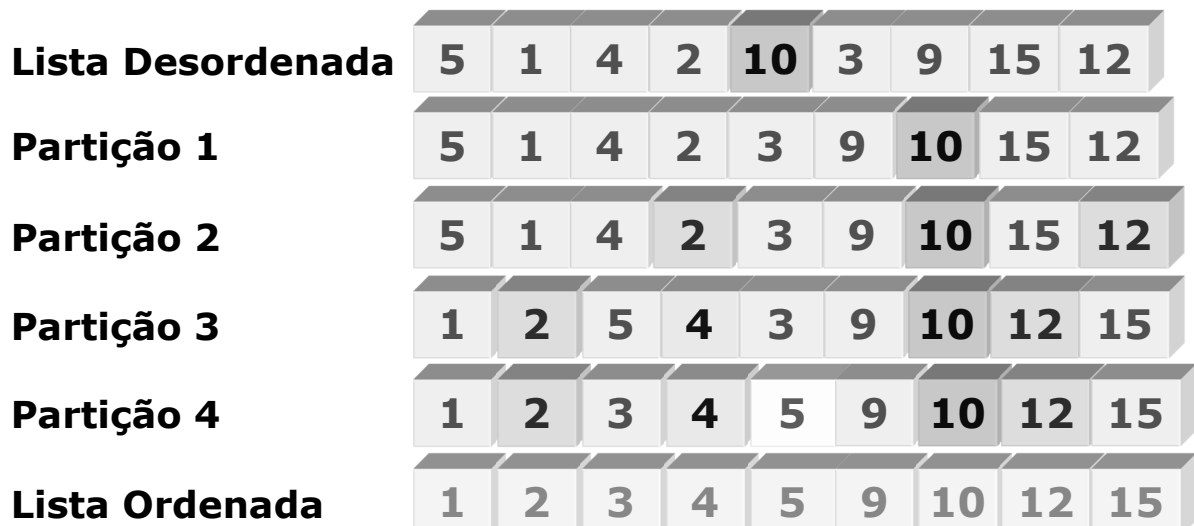
45

Ordenação por troca

✚ Ordenação Rápida (*Quicksort*)

✚ Exemplo 03 – Ordenação da sequência

5, 1, 4, 2, 10, 3, 9, 15, 12



46

Ordenação por troca

⌘ Ordenação Rápida (*Quicksort*)

⌘ Algoritmo em C

```
void quick(int item[], int cont)
{
    qs(item, 0, cont-1);
}
void qs(int *item[], int esq, int dir)
{
    int i, j;
    int x, y;
    i = esq;
    j = dir;
    x=item[(esq + dir)/2];
    do {
        while( (item[i]<x) && (i < dir)) i++;
        while( (x<item[j]) && (j > esq)) j--;
        if(i <= j){
            y = item[i];
            item[i] = item[j];
            item[j] = y;
            i++;
            j--;
        }
    } while(i <= j);
    if(esq < j) qs(item, esq, j);
    if(i < dir) qs(item, i, dir);
}
```

7

Ordenação por troca

⌘ Ordenação Rápida (*Quicksort*)

⌘ Considerações Finais:

- ✧ O valor central pode ser escolhido ***aleatoriamente*** ou selecionado através da ***média*** de um pequeno conjunto de valores retirado da matriz.
- ✧ Ordenação ***ótima*** ⇒ Necessidade de seleção de um valor ***precisamente*** no ***centro*** da faixa de valores (tarefa difícil).

Ordenação por troca

⌘ Ordenação Rápida (*Quicksort*)

⊕ Considerações Finais

- ✧ Ordenação ***péssima*** \Rightarrow Posicionamento do valor escolhido em uma extremidade e, mesmo assim, o *quicksort* seleciona o elemento central da matriz.
- ✧ Número médio de comparações \Rightarrow ***$n \log(n)$*** .
- ✧ Número médio de trocas (aproximado) \Rightarrow ***$n/6 \log(n)$*** .

49

Ordenação por troca

⌘ Ordenação Rápida (*Quicksort*)

⊕ Considerações Finais

- ✧ Ordenação ***Quicksort*** \Rightarrow Melhor dos métodos estudados:
 - ✧ Velocidade de processamento dos dados.
- ✧ Ordenação de listas de dados muito pequenas (***$n < 100$***)
 - ✧ Comprometimento da eficiência do método \Rightarrow Tempo extra das chamadas recursivas.
 - ✧ Sugestão \Rightarrow Ordenação ***borbulhante***.

50

Ordenação por troca

⌘ Ordenação Rápida (*Quicksort*)

⊕ Considerações Finais

- ✧ Fornecimento da função *qsort()* pela maioria dos compiladores (parte da biblioteca padrão).
 - ✧ Impossibilidade de aplicação de uma função generalizada a *todas* as situações
 - ✧ Parametrização de *qsort()* para operação sobre uma variedade de dados.
 - Execução mais lenta do que a de um algoritmo de ordenação semelhante operando *apenas* sobre um tipo de dados.

51

Ordenação por troca

⌘ Ordenação de Outras Estruturas de Dados

⊕ Ordenação de *Strings*

- ✧ Estratégia Vantajosa:
 - ✧ Criação de uma matriz de apontadores e caracteres para as *strings*:
 - Possibilidade de indexação fácil.
 - Manutenção da base do algoritmo *quicksort* inalterada.

52

Ordenação por troca

⌘ Quicksort para strings

```
/* Ordenação de strings */
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define N 5
#define TAM 10
void quick_string(char i[][TAM], int cont);
void qs_string(char i[][TAM], int esq, int dir);
void main(void)
{
    int k;
    char s[N][TAM];
    clrscr();
    for(k = 0; k < N; k++){
        printf("Digite uma string [%d]:", k+1);
        gets(s[k]);
    }
    quick_string(s, N);
    printf("\nStrings ordenadas:");
    for(k = 0; k < N; k++) printf("\n%s", s[k]);
    getch();
}
```

53

Ordenação por troca

⌘ Quicksort para strings

```
/* Quicksort para strings */
void quick_string(char i[][TAM], int cont){
    qs_string(i, 0, cont-1);
}
void qs_string(char i[][TAM], int esq, int dir){
    register int j, k;
    char *x;
    char auxi[10];
    i = esq; j = dir;
    x=i[(esq + dir)/2];
    do {
        while(strcmp(i[k], x) < 0 && k < dir) k++;
        while(strcmp(i[j], x) > 0 && j > esq) j--;
        if(k <= j){
            strcpy(auxi, i[k]);
            strcpy(i[k], i[j]);
            strcpy(i[j], auxi);
            i++; j--;
        }
    } while(k <= j);
    if(esq < j) qs_string(i, esq, j);
    if(i < dir) qs_string(i, k, dir);
}
```

54

Ordenação por troca

⊞ Ordenação de Outras Estruturas de Dados

⊕ Ordenação de Arquivos em Disco

✧ Modos de Acesso a Arquivos em Disco:

- ✧ ***Sequencial***
- ✧ ***Direto* ou *Randômico (aleatório)***

55

Ordenação por troca

⊞ Ordenação de Outras Estruturas de Dados

⊕ Ordenação de Arquivos em Disco

✧ Arquivos pequenos

✧ Leitura na memória:

- Ordenação eficiente a partir de algoritmos de ordenação de vetores (apresentados em slides anteriores).

56

Ordenação por troca

⊞ Ordenação de Outras Estruturas de Dados

⊞ Ordenação de Arquivos em Disco

✧ Arquivos grandes

✧ Impossibilidade de leitura na memória:

- Adoção de técnicas especiais.

57

Ordenação por troca

⊞ Ordenação de Outras Estruturas de Dados

⊞ Ordenação de Arquivos em Disco

✧ Vantagens (acesso aleatório – sequencial)

✧ Facilidade de manutenção e atualização de informações sem necessidade de cópia de toda a lista de dados.

✧ Possibilidade de tratamento como um grandes vetores (matrizes) no disco:

- Simplificação do processo de ordenação

Nota: O tratamento de um arquivo de acesso aleatório como uma matriz implica a possibilidade de uso do método *quicksort* com modificações relativamente simples.

58

Métodos de ordenação interna

- ⊞ a) Ordenação por troca
 - ⊕ *Bubble Sort* (método da bolha)
 - ⊕ *Shaker Sort (Cocktail Sort)* (método oscilante)
 - ⊕ *Quick Sort* (método rápido)
- ⊞ b) Ordenação por seleção
 - ⊕ *Selection Sort* (método da seleção direta)
 - ⊕ *Heap Sort*
- ⊞ c) Ordenação por inserção
 - ⊕ *Insertion Sort* (método da inserção direta)
 - ⊕ *Shell Sort* (método dos incrementos decrescentes)
- ⊞ d) Ordenação por intercalação
 - ⊕ *Merge Sort*
 - ⊕ *Merge Array*
- ⊞ e) Ordenação por distribuição
 - ⊕ *Counting Sort* (método da contagem)
 - ⊕ *Bucket Sort (Bin Sort)* (método da distribuição de chaves)
 - ⊕ *Radix Sort (Digital Sort)* (método da raiz)

59

Ordenação por seleção

⊞ Ordenação por *Seleção Direta*

⊕ Procedimento Básico:

✧ Primeira iteração:

- ✧ Identificação do *maior* valor e permuta com o último item da sequência.

- Posicionamento correto do maior item.

60

Ordenação por seleção

⊞ Ordenação por *Seleção Direta*

⊞ Procedimento Básico:

✧ Segunda iteração:

- ✧ Varredura apenas dos $n-1$ itens da sequência e permuta do maior deles com o item ocupante da posição $n-1$.
 - Posicionamento correto dos dois maiores itens.

61

Ordenação por seleção

⊞ Ordenação por *Seleção Direta*

⊞ Procedimento Básico:

✧ n -ésima iteração:

- ✧ Ordenação da sequência inteira após n iterações.

- ⊞ Variante simples \Rightarrow Identificação do ***menor*** item a cada iteração e seu reposicionamento na porção inicial da sequência de itens.

62

Ordenação por seleção

⊞ Algoritmo: Ordenação por *Seleção Direta*

⊞ Versão 1

- ✧ Seja v um vetor de elementos tidos como desordenados.
- ✧ Seja O a parte ordenada de v e N a parte não ordenada de v ; assim, no início do algoritmo, O é vazia e $N=v$.
- ✧ Repita até que N seja vazia:
- ✧ Seja x o menor elemento de N .
- ✧ Remova x de N .
- ✧ Insira x como último elemento de O , mantendo O ordenada.

63

Ordenação por seleção

⊞ Algoritmo: Ordenação por *Seleção Direta*

⊞ Versão 2

- ✧ Seja v um vetor de elementos tidos como desordenados.
- ✧ Para cada posição i desse vetor, faça:
- ✧ Calcule o i -ésimo menor elemento do vetor.
- ✧ Troque esse elemento de lugar com o de posição i no vetor, se necessário.

64

Ordenação por seleção

✚ Ordenação por *Seleção Direta*

✚ Exemplo 04 – Ordenar: **3, 7, 5, 2, 6, 1, 4**

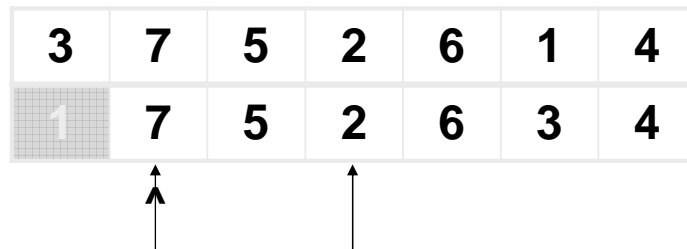


65

Ordenação por seleção

✚ Ordenação por *Seleção Direta*

✚ Exemplo 04 – Ordenar: **3, 7, 5, 2, 6, 1, 4**




66

Ordenação por seleção

✚ Ordenação por *Seleção Direta*

✚ Exemplo 04 – Ordenar: **3, 7, 5, 2, 6, 1, 4**

3	7	5	2	6	1	4
1	7	5	2	6	3	4
1	2	5	7	6	3	4




67

Ordenação por seleção

✚ Ordenação por *Seleção Direta*

✚ Exemplo 04 – Ordenar: **3, 7, 5, 2, 6, 1, 4**

3	7	5	2	6	1	4
1	7	5	2	6	3	4
1	2	5	7	6	3	4
1	2	3	7	6	5	4



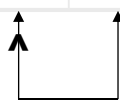
68

Ordenação por seleção

✚ Ordenação por *Seleção Direta*

✚ Exemplo 04 – Ordenar: **3, 7, 5, 2, 6, 1, 4**

3	7	5	2	6	1	4
1	7	5	2	6	3	4
1	2	5	7	6	3	4
1	2	3	7	6	5	4
1	2	3	4	6	5	7



69

Ordenação por seleção

✚ Ordenação por *Seleção Direta*

✚ Exemplo 04 – Ordenar: **3, 7, 5, 2, 6, 1, 4**

3	7	5	2	6	1	4
1	7	5	2	6	3	4
1	2	5	7	6	3	4
1	2	3	7	6	5	4
1	2	3	4	6	5	7
1	2	3	4	5	6	7
1	2	3	4	5	6	7
1	2	3	4	5	6	7

70

Ordenação por seleção

⊞ Ordenação por *Seleção Direta*

⊞ Programa em C

```
void Selecao(int i[], int n){  
    int k, j;  
    int min, auxi;  
    for (k = 0; k < n-1; k++){  
        min = k;  
        for (j = k+1; j < n; j++){  
            if (i[j] < i[min])  
                min = j;  
        }  
        auxi = i[k];  
        i[k] = i[min];  
        i[min] = auxi;  
    }  
}
```

71

Ordenação por seleção

⊞ Ordenação por *Seleção Direta*

⊞ Considerações Finais:

- ✧ Número de comparações $\Rightarrow \frac{1}{2}(n^2 - n)$.
- ✧ Número de trocas no melhor caso $\Rightarrow 3(n-1)$.
- ✧ Número de trocas no caso médio $\Rightarrow n(\ln n + \gamma)$.
 - ✧ $\gamma \cong 0,577216 \Rightarrow$ Constante de Euler
- ✧ Número de trocas no pior caso $\Rightarrow \frac{1}{4}n^2 + 3(n - 1)$.

72

Ordenação por seleção

⌘ Ordenação *Heapsort*

- ⊕ O projeto por indução que leva ao *Heapsort* é essencialmente o mesmo do *Selection Sort*: selecionamos e posicionamos o maior (ou menor) elemento do conjunto e então aplicamos a hipótese de indução para ordenar os elementos restantes.
- ⊕ A diferença importante é que no *Heapsort* utilizamos a estrutura de dados *heap* para selecionar o maior (ou menor) elemento eficientemente.
- ⊕ Um *heap* é um vetor que simula uma árvore binária completa, a menos, talvez, do último nível, com estrutura de *heap*.

73

Ordenação por seleção

⌘ Ordenação *Heapsort*

- ⊕ Na simulação da árvore binária completa com um vetor, definimos que o nó i tem como filhos esquerdo e direito os nós $2i$ e $2i + 1$ e como pai o nó $\lfloor i/2 \rfloor$.
- ⊕ Uma árvore com estrutura de *heap* é aquela em que, para toda subárvore, o nó raiz é maior ou igual (ou menor ou igual) às raízes das subárvores direita e esquerda.
- ⊕ Assim, o maior (ou menor) elemento do *heap* está sempre localizado no topo, na primeira posição do vetor.

74

Ordenação por seleção

⌘ Ordenação *Heapsort*

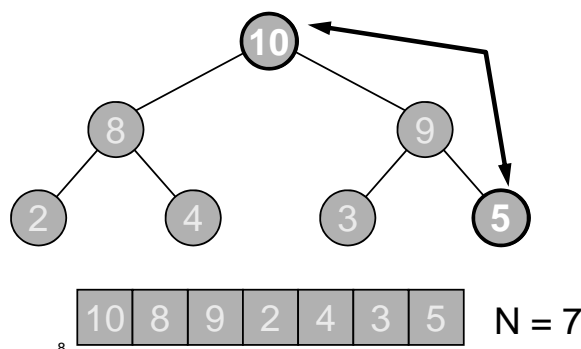
- ⌘ Então, o uso da estrutura *heap* permite que:
 - ✧ O elemento máximo do conjunto seja determinado e corretamente posicionado no vetor em tempo constante, trocando o primeiro elemento do *heap* com o último.
 - ✧ O trecho restante do vetor (do índice 1 ao $n - 1$), que pode ter deixado de ter a estrutura de *heap*, volte a tê-la com número de trocas de elementos proporcional à altura da árvore.
- ⌘ O algoritmo *Heapsort* consiste então da construção de um *heap* com os elementos a serem ordenados, seguida de sucessivas trocas do primeiro com o último elemento e rearranjos do *heap*.

75

Ordenação por seleção

⌘ Ordenação *Heapsort*

- ⌘ Exemplo 05 – Ordenar: 10, 8, 9, 2, 4, 3, 5

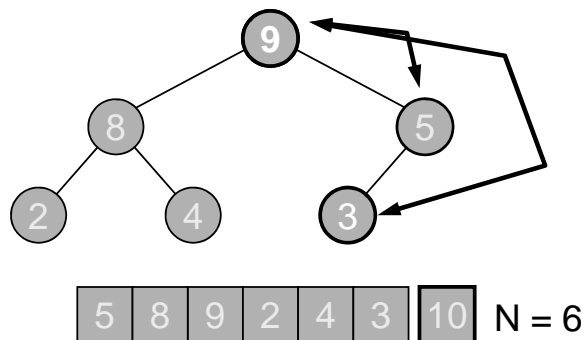


76

Ordenação por seleção

⌘ Ordenação *Heapsort*

⊕ Exemplo 05 – Ordenar: 10, 8, 9, 2, 4, 3, 5

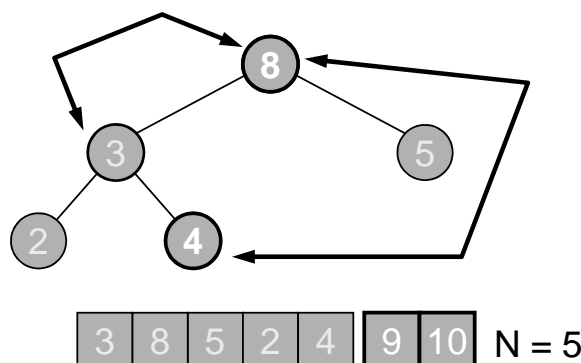


77

Ordenação por seleção

⌘ Ordenação *Heapsort*

⊕ Exemplo 05 – Ordenar: 10, 8, 9, 2, 4, 3, 5

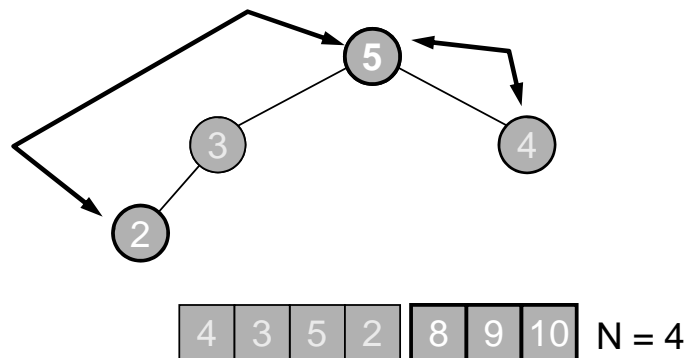


78

Ordenação por seleção

✚ Ordenação *Heapsort*

✚ Exemplo 05 – Ordenar: 10, 8, 9, 2, 4, 3, 5

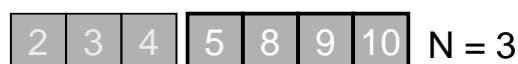
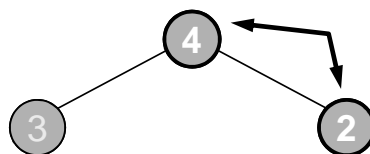


79

Ordenação por seleção

✚ Ordenação *Heapsort*

✚ Exemplo 05 – Ordenar: 10, 8, 9, 2, 4, 3, 5

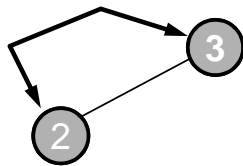


80

Ordenação por seleção

⌘ Ordenação *Heapsort*

⊕ Exemplo 05 – Ordenar: 10, 8, 9, 2, 4, 3, 5



2	3	4	5	8	9	10
---	---	---	---	---	---	----

 N = 2

81

Ordenação por seleção

⌘ Ordenação *Heapsort*

⊕ Exemplo 05 – Ordenar: 10, 8, 9, 2, 4, 3, 5



2	3	4	5	8	9	10
---	---	---	---	---	---	----

 N = 1

82

Ordenação por seleção

⌘ Ordenação *Heapsort*

```
// Recebe p em 1..m e rearranja o vetor
v[1..m] de modo
// que o "subvetor" cuja raiz é p seja um
max-heap.
// Supõe que os "subvetores" cujas raízes
são filhos
// de p já são max-heaps.

void peneira (int p, int m, int v[])
{
    int f = 2*p, x = v[p];
    while (f <= m) {
        if (f < m && v[f] < v[f+1]) ++f;
        // f é o filho "mais valioso" de p
        if (x >= v[f]) break;
        v[p] = v[f];
        p = f, f = 2*p;
    }
    v[p] = x;
}
```

```
// Rearranja os elementos do vetor v[1..n]
// de modo que fiquem em ordem crescente

void heapsort (int n, int v[])
{
    int p, m, x;
    for (p = n/2; p >= 1; --p)
        peneira (p, n, v);
    for (m = n; m >= 2; --m) {
        x = v[1], v[1] = v[m], v[m] = x;
        peneira (1, m-1, v);
    }
}
```

83

Ordenação por seleção

⌘ Ordenação *Heapsort*

```
/* Cria HEAP */
int insert(int item)
{
    int temp, pai, filho = ++n;
    if(n==MAX)
    { /* overflow check */
        printf("Heap está cheio\n");
        return(0);
    }
    heap[filho] = item;
    pai = (filho%2 == 0)? filho/2-1
        : filho/2;
    while(pai >= 0)
    {
        if(heap[filho] > heap[pai])
        {
            temp = heap[filho];
            heap[filho] = heap[pai];
            heap[pai] = temp;
        }
        filho = pai;
        pai = (filho%2 == 0)? filho/2-1
            : filho/2;
    }
    return(1);
} /* insert */
```

```
int adjust(int , int n)
{
    int item, temp, pai=0, filho = 1;
    if(n==1)
    { /* underflow check */
        printf("Heap está vazio\n");
        return(0);
    }
    item = heap[0];
    heap[0] = heap[n--];
    while(filho <= n)
    {
        if(filho < n && heap[filho] <
            heap[filho+1])
            filho++;
        if(heap[filho] > heap[pai])
        {
            temp = heap[filho];
            heap[filho] = heap[pai];
            heap[pai] = temp;
        }
        pai = filho;
        filho = pai*2 + 1;
    }
    return(item);
} /*adjust */
```

4

Ordenação por seleção

⌘ Ordenação *Heapsort*

⊕ Desempenho:

✧ O mais lento dos ordenadores $O(N\log_2 N)$.

✧ Vantagens:

✧ Não necessita da recursão do *quicksort*.

✧ Não necessita do espaço extra do *mergesort*.

✧ O pior caso é ainda $O(N\log_2 N)$ diferentemente dos outros dois.

85

Métodos de ordenação interna

⌘ a) Ordenação por troca

⊕ *Bubble Sort* (método da bolha)

⊕ *Shaker Sort (Cocktail Sort)* (método oscilante)

⊕ *Quick Sort* (método rápido)

⌘ b) Ordenação por seleção

⊕ *Selection Sort* (método da seleção direta)

⊕ *Heap Sort*

⌘ c) Ordenação por inserção

⊕ *Insertion Sort* (método da inserção direta)

⊕ *Shell Sort* (método dos incrementos decrescentes)

⌘ d) Ordenação por intercalação

⊕ *Merge Sort*

⊕ *Merge Array*

⌘ e) Ordenação por distribuição

⊕ *Counting Sort* (método da contagem)

⊕ *Bucket Sort (Bin Sort)* (método da distribuição de chaves)

⊕ *Radix Sort (Digital Sort)* (método da raiz)

86

Ordenação por inserção

⌘ Ordenação por *Inserção Direta*

⊕ Procedimento:

- ✧ Ordenação dos **2 primeiros** itens da sequência.
- ✧ Inserção do **3º item** na sua posição ordenada com relação aos 2 primeiros.
- ✧ Inserção do **4º item** na lista dos 3 elementos.
- ✧ Prosseguimento até a ordenação de ***todos*** os elementos.

87

Ordenação por inserção

⌘ Algoritmo: Ordenação *Inserção Direta*

- ⊕ Seja ***v*** um vetor de elementos tidos como desordenados.
- ⊕ Seja ***O*** a parte ordenada de ***v*** e ***N*** a parte não ordenada de ***v***; assim, no início do algoritmo, ***O*** é vazia e ***N=v***.
- ⊕ Repita até que ***N*** seja vazia:
 - ✧ Seja ***x*** o primeiro elemento de ***N***.
 - ✧ Remova ***x*** de ***N***.
 - ✧ Insira ***x*** em ***O*** de tal forma que ***O*** continue ordenada.

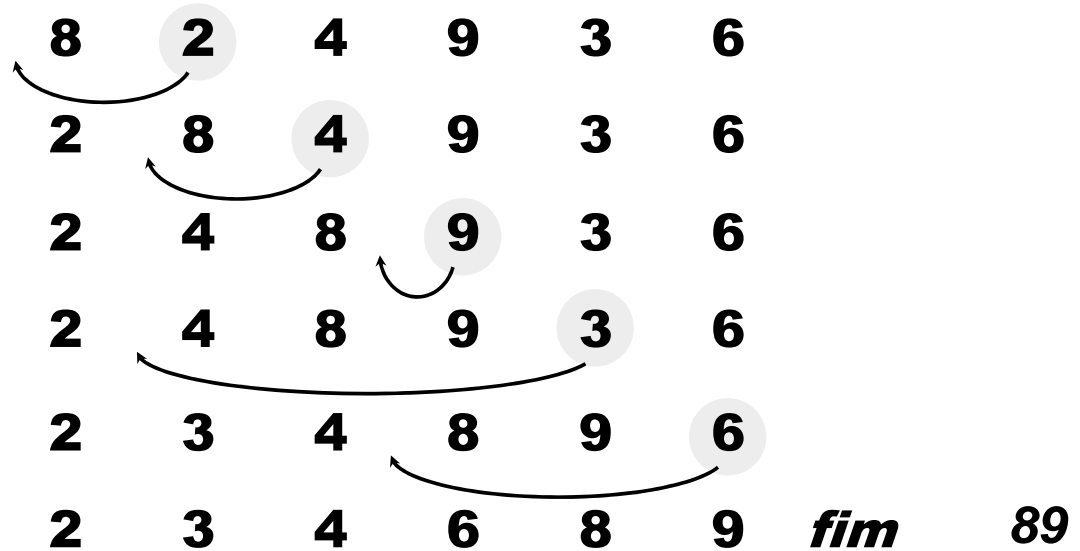
88

Ordenação por inserção

✚ Ordenação *Inserção Direta*

✚ Exemplo 06 – Ordenação da sequência

8, 2, 4, 9, 3, 6



Ordenação por inserção

✚ Ordenação *Inserção Direta*

✚ Algoritmo em C

```
void Insercao(int i[], int n){
    int k, j, chave;

    for (k=0; k < n; k++){
        chave = i[k];
        j = k;
        while ((j > 0) && (i[j-1] > chave)){
            i[j] = i[j-1];
            j = j - 1;
        }
        i[j] = chave;
    }
}
```

Ordenação por inserção

✚ Ordenação *Inserção Direta*

✚ Vantagens:

✧ Comportamento natural:

- ✧ Menor carga de trabalho quando os itens já se encontram ordenados.
- ✧ Máxima carga de trabalho quando a lista está ordenada no sentido inverso.

✧ Não rearranjo de itens de mesma chave:

- ✧ Uma lista ordenada por duas chaves permanecerá ordenada para ambas as chaves após a ordenação.

91

Ordenação por inserção

✚ Ordenação *Inserção Direta*

✚ Considerações Finais:

✧ Número de comparações:

- ✧ Lista em ordem $\Rightarrow (n - 1)$
- ✧ Lista sem ordem $\Rightarrow \frac{1}{2}(n^2 - n) - 1$

✧ Número de trocas no melhor caso $\Rightarrow 2(n - 1)$.

✧ Número de trocas no caso médio $\Rightarrow \frac{1}{4}(n^2 + 9n - 10)$.

✧ Número de trocas no pior caso $\Rightarrow \frac{1}{2}(n^2 + 3n - 4)$.

92

Ordenação por inserção

⊞ Métodos Melhores de Ordenação

⊕ Ineficiência de todos os algoritmos estudados até este ponto \Rightarrow Tempo de processamento da ordem de n^2 .

✧ Ordenações lentas para grandes listas ($n > 100$).

⊕ Solução:

✧ Outras formas de ordenação:

✧ E.g. métodos *Shell* e *Quicksort*

93

Ordenação por inserção

⊞ Ordenação *Shell*

⊕ Considerações Iniciais:

✧ Concepção do método \Rightarrow *D. L. Shell*

✧ Método de operação frequentemente associado ao mecanismo de empilhamento de conchas do mar.

✧ Derivação:

✧ Ordenação por Inserção.

✧ Fundamentação na redução dos incrementos.

94

Ordenação por inserção

Ordenação *Shell*

⊕ Procedimento:

✧ Consideração Inicial:

- ✧ Composição do vetor por vários segmentos, cada um dos quais reúne os itens que se encontram a uma distância d entre si.

✧ Ordenação de cada segmento.

95

Ordenação por inserção

Ordenação *Shell*

⊕ Procedimento:

- ✧ Redução de d e repetição do processo para segmentos nos quais o intervalo entre itens é *menor* do que na primeira iteração do processo.
- ✧ Repetições sucessivas do processo, até que $d = 1$ (valor obrigatório para o intervalo na última iteração).

96

Ordenação por inserção

✚ Ordenação *Shell*

✚ Definição do valor inicial de d – Exemplo:

- ✧ Ordenação de ***todos*** os itens afastados entre si de **3** posições.
- ✧ Ordenação de ***todos*** os itens afastados entre si de **2** posições.
- ✧ Finalmente, ordenação de ***todos*** os itens ***adjacentes***.

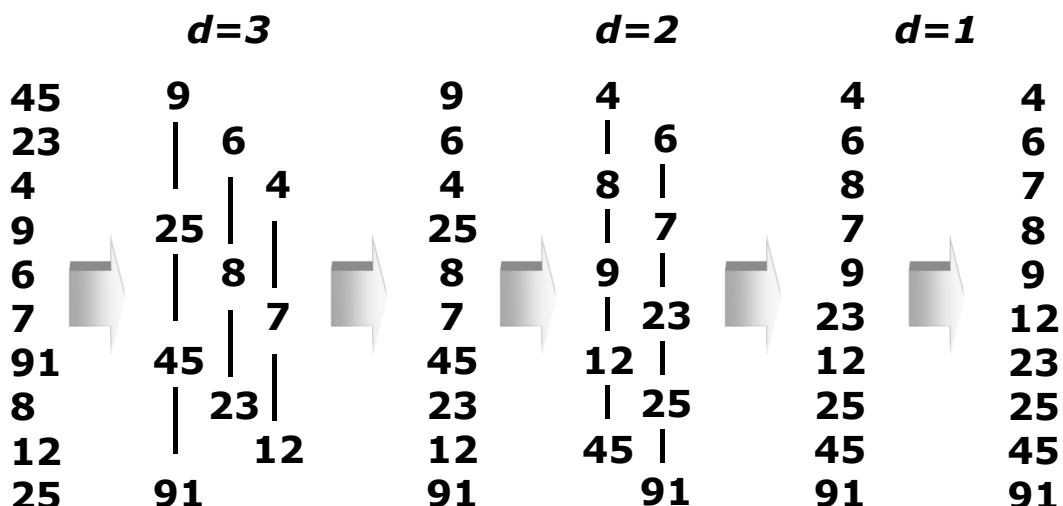
97

Ordenação por inserção

✚ Ordenação *Shell*

✚ Exemplo 07 – Ordenação da sequência

45, 23, 4, 9, 6, 7, 91, 8, 12, 25

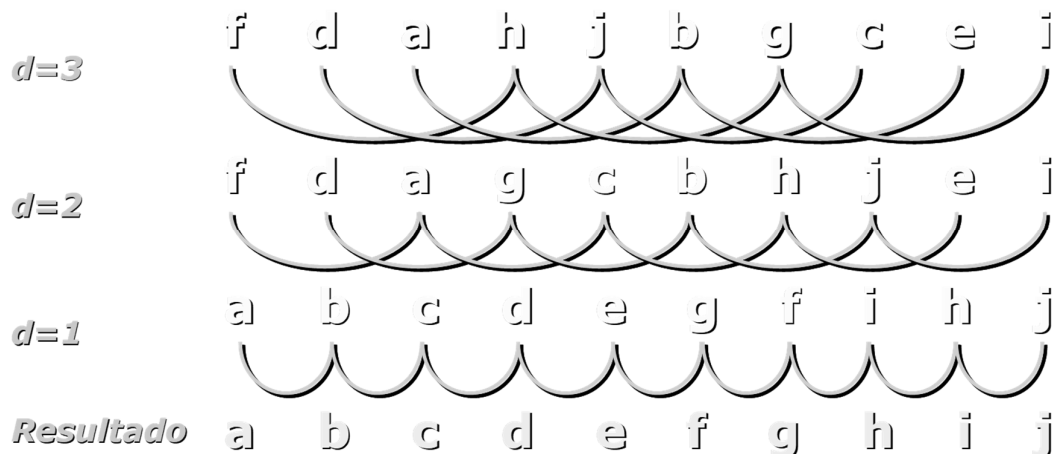


98

Ordenação por inserção

⌘ Ordenação *Shell*

⊕ Exemplo 08 – Ordenação da sequência ***f, d, a, h, j, b, g, c, e, i***



99

Ordenação por inserção

⌘ Ordenação *Shell*

⊕ Algoritmo em C

```
void ShellSort(int x[], int n, int incrmnts[], int numinc)
{
    int incr, j, k, tam, y;
    for(incr = 0; incr < numinc; incr++)
    {
        tam = incrmnts[incr];
        for(j = tam; j < n; j++)
        {
            y = x[j];
            for(k = j - tam; k >= 0 && y < x[k]; k -= tam)
                x[k + tam] = x[k];
            x[k + tam] = y;
        }
    }
}
```

100

Ordenação em C

⊞ Ordenação *Shell*

⊞ Algoritmo em C

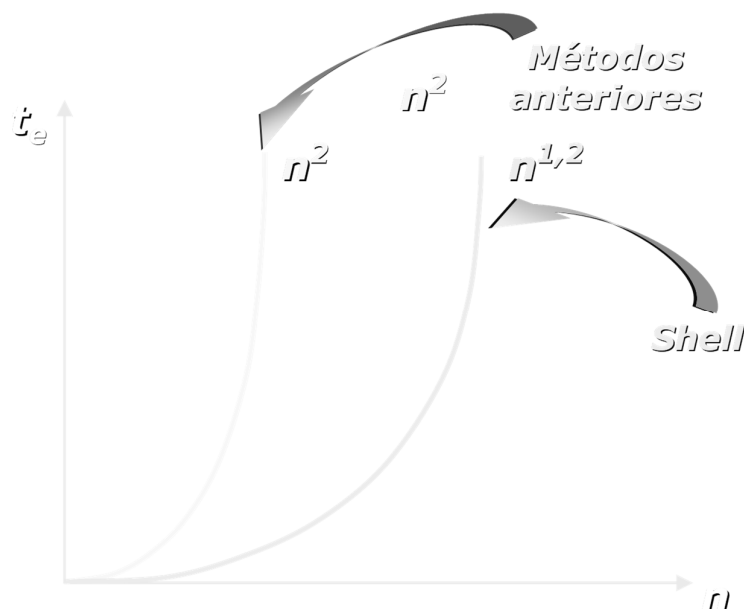
```
void Shell(){  
    int i[n], d, i, j, auxi;  
    for(d= n/2; d > 0; d = d/2)  
        for(i = d; i < n; i++)  
            for(j = i - d; j >= 0 && i[j] > i[j+d]; j = j - d){  
                auxi = i[j];  
                i[j] = i[j+d];  
                i[j+d] = auxi;  
            }  
}
```

101

Ordenação por inserção

⊞ Ordenação *Shell*

⊞ *Shell* x Métodos de Ordenação Anteriores



102

Métodos de ordenação interna

- ⊞ a) Ordenação por troca
 - ⊕ *Bubble Sort* (método da bolha)
 - ⊕ *Shaker Sort (Cocktail Sort)* (método oscilante)
 - ⊕ *Quick Sort* (método rápido)
- ⊞ b) Ordenação por seleção
 - ⊕ *Selection Sort* (método da seleção direta)
 - ⊕ *Heap Sort*
- ⊞ c) Ordenação por inserção
 - ⊕ *Insertion Sort* (método da inserção direta)
 - ⊕ *Shell Sort* (método dos incrementos decrescentes)
- ⊞ d) Ordenação por intercalação
 - ⊕ *Merge Sort*
 - ⊕ *Merge Array*
- ⊞ e) Ordenação por distribuição
 - ⊕ *Counting Sort* (método da contagem)
 - ⊕ *Bucket Sort (Bin Sort)* (método da distribuição de chaves)
 - ⊕ *Radix Sort (Digital Sort)* (método da raiz)

103

Ordenação por intercalação

⊞ Ordenação Rápida (*Merge Sort*)

- ⊕ Baseia-se em junções sucessivas (*merge*) de 2 sequências ordenadas em uma única sequência ordenada.
- ⊕ Aplica um método “dividir para conquistar”:
 - ✧ Divide o vetor em 2 segmentos (sub-vetores) de comprimento $\lfloor n/2 \rfloor$ e $\lceil n/2 \rceil$.
 - ✧ Ordena recursivamente cada sub-vetor (dividindo novamente, quando possível).
 - ✧ Faz o *merge* dos 2 sub-vetores ordenados para obter o vetor ordenado completo.

104

Ordenação por intercalação

⌘ Algoritmo: Ordenação *Mergesort*

- ⊕ Seja v um vetor de n elementos.
- ⊕ Se v contiver apenas um elemento, então ele já está ordenado; logo, retorne v como resposta.
- ⊕ Caso contrário:
 - ✧ Quebre v em duas metades, $v1$ e $v2$, de forma que $v1$ tenha $n/2$ elementos e $v2$ tenha $n-n/2$ elementos.
 - ✧ Ordene por *Merge Sort* a metade $v1$.
 - ✧ Ordene por *Merge Sort* a metade $v2$.
 - ✧ Intercale os elementos que compõem $v1$ e $v2$ de modo a formar um vetor v ordenado. ("Merge")
 - ✧ Retorne v como resposta.

105

Ordenação por intercalação

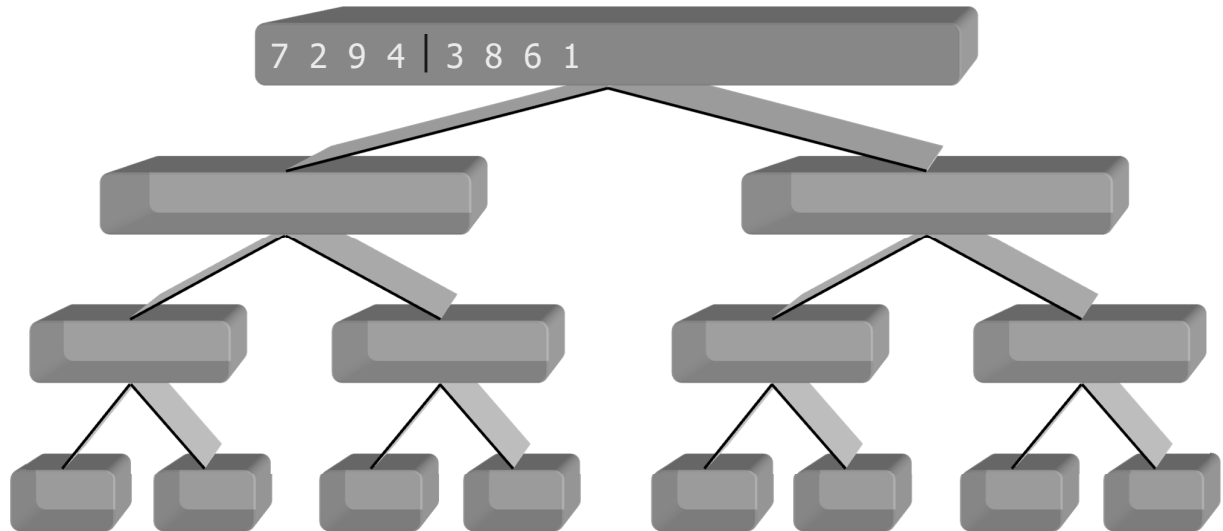
⌘ Ordenação *Mergesort*

```
#define NUMELEM 1000
void MergeSort(int x[], int n)
{
    int aux[NUMELEM], i, j, k, a1, a2, u1, u2, tam;
    tam = 1; //intercala subvetores de tamanho 1
    while (tam < n )
    {
        a1 = k = 0;
        while ((a1 + tam) < n )
        {
            a2 = a1 + tam;
            u1 = a2 - 1;
            u2 = (a2+tam-1 < n) ? a2+tam-1 : n-1;
            for(i = a1, j = a2; i <= u1 && j <= u2; k++)
            {
                if (x[i] <= x[j]) aux[k] = x[i++];
                else aux[k] = x[j++];
            }
            for( ; i <= u1; k++) aux[k] = x[i++];
            for( ; j <= u2; k++) aux[k] = x[j++];
            a1 = u2+1;
        }
        //copia o restante de x em aux
        for(i = a1; k < n; i++) aux[k++] = x[i];
        //copia aux em x
        for(i = 0; i < n; i++) x[i] = aux[i];
        tam *= 2;
    }
}
```

106

Ordenação por intercalação

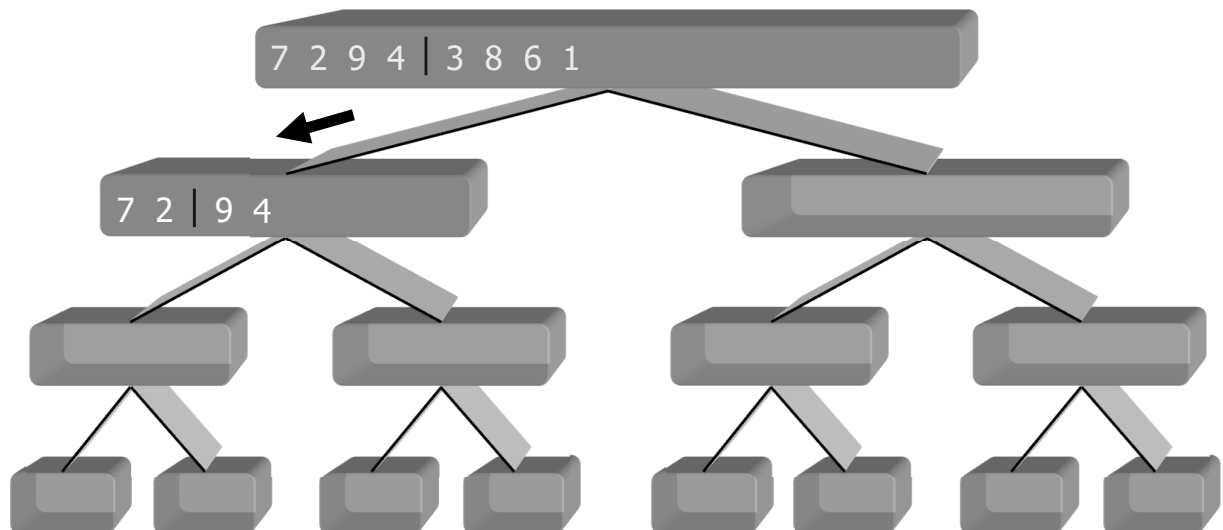
Partição



107

Ordenação por intercalação

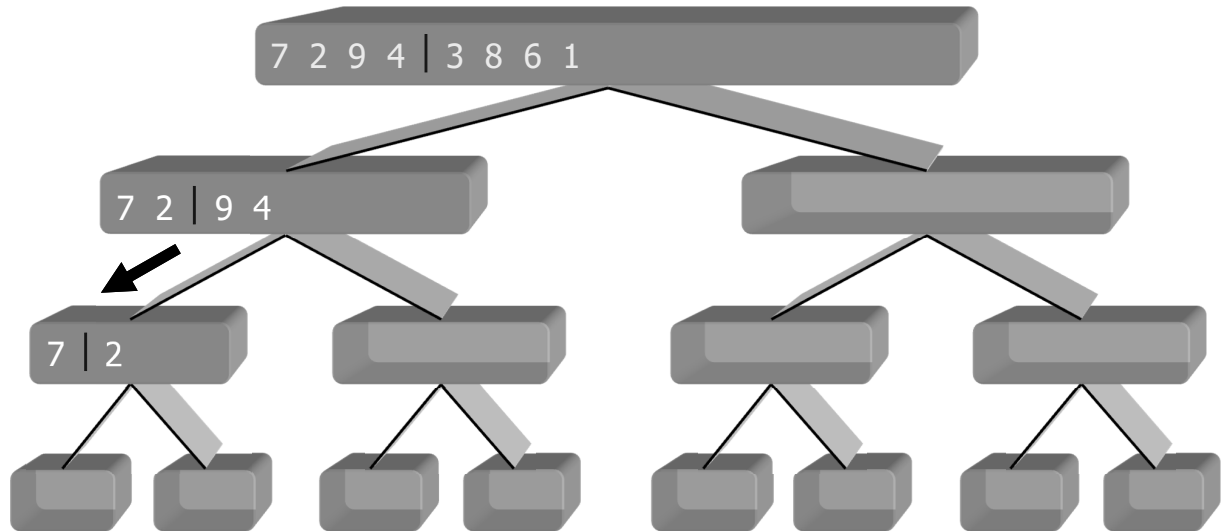
Chamada recursiva, partição



108

Ordenação por intercalação

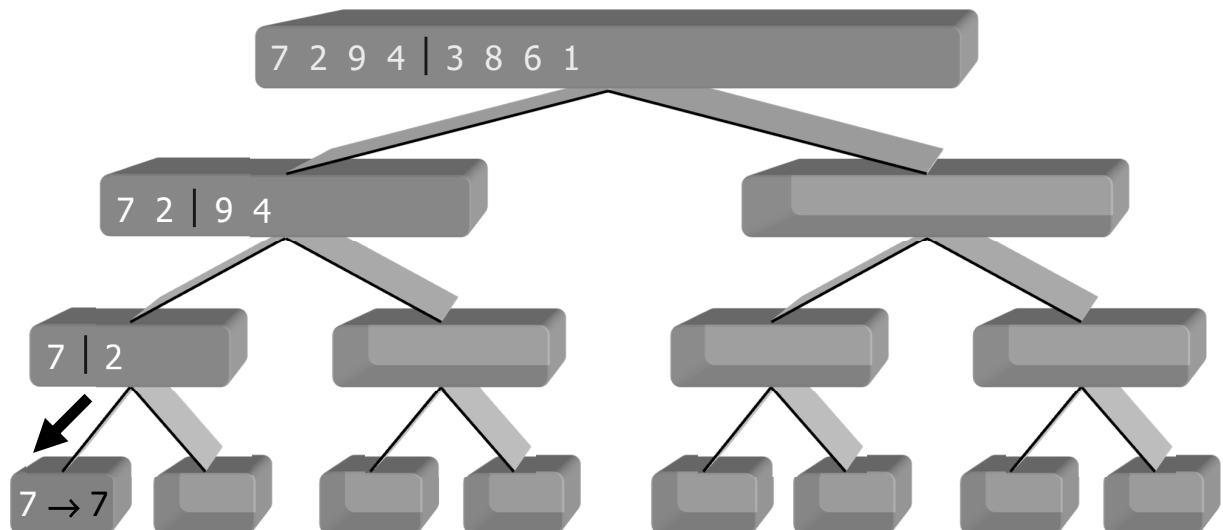
Chamada recursiva, partição



109

Ordenação por intercalação

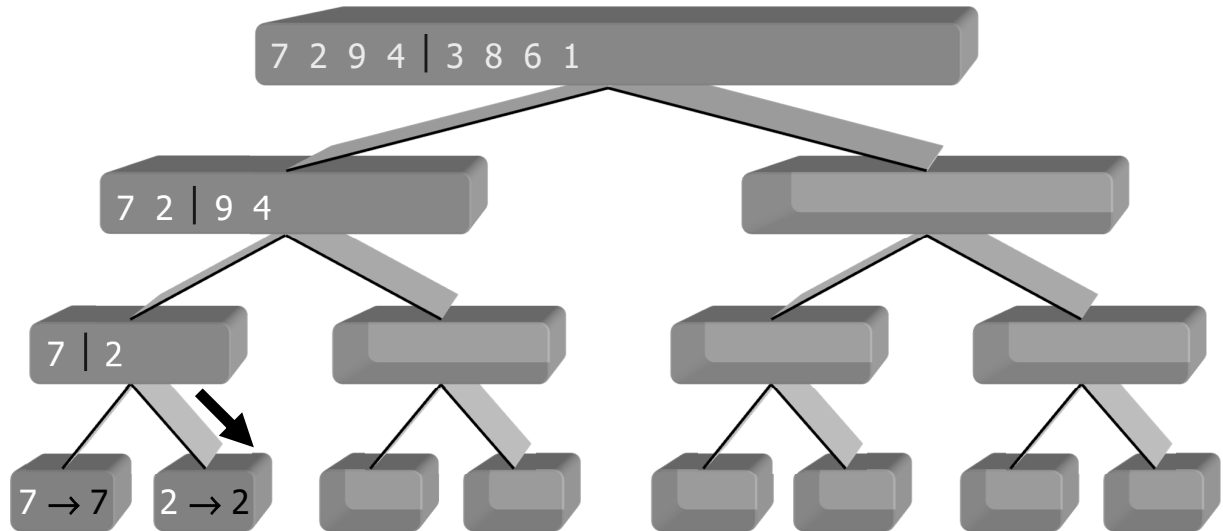
Chamada recursiva, partição



110

Ordenação por intercalação

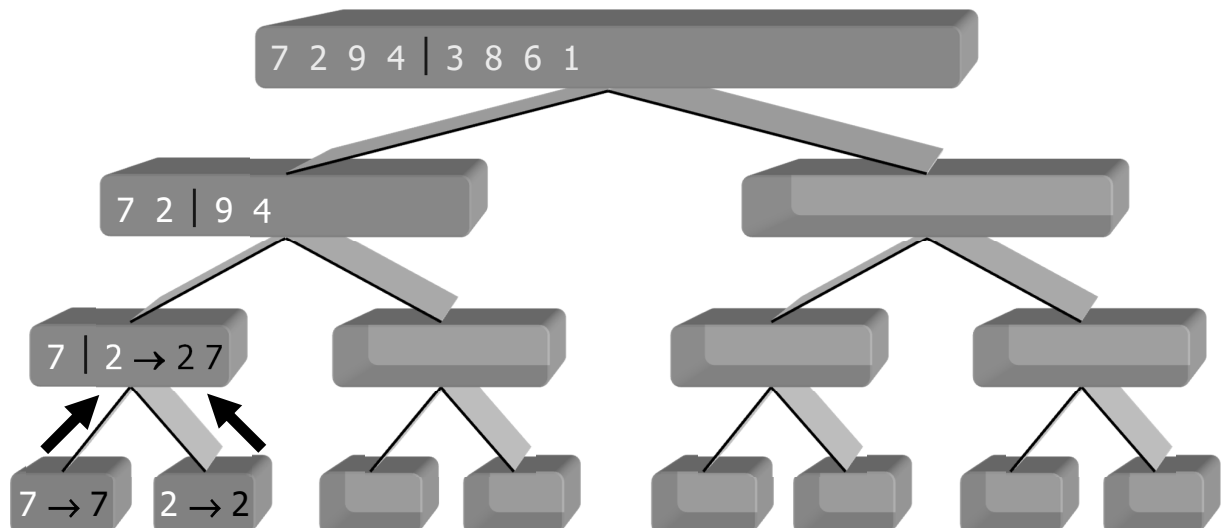
⌘ Chamada recursiva, caso base



111

Ordenação por intercalação

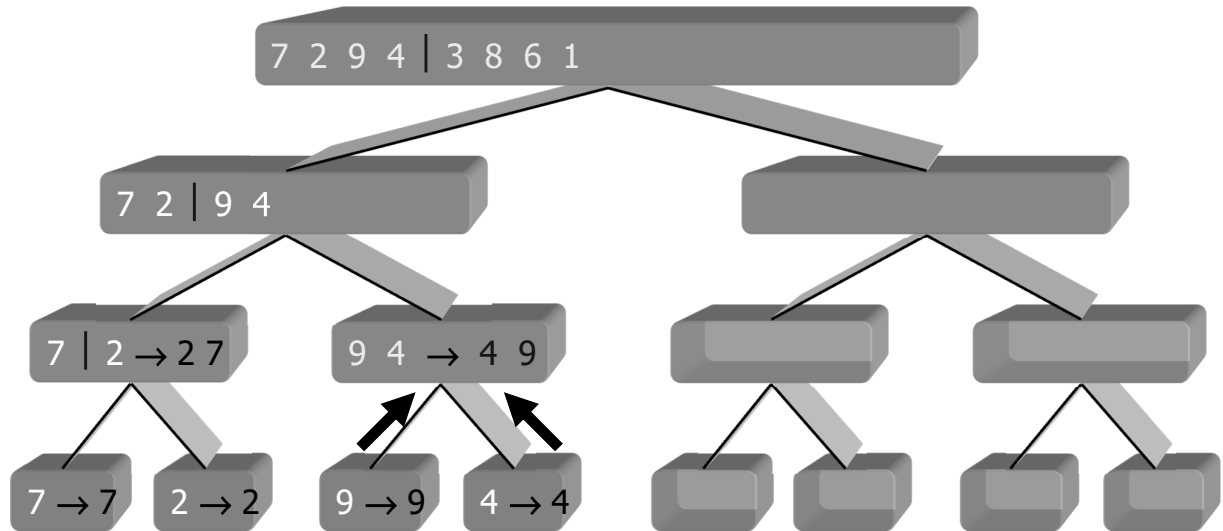
⌘ Unificação



112

Ordenação por intercalação

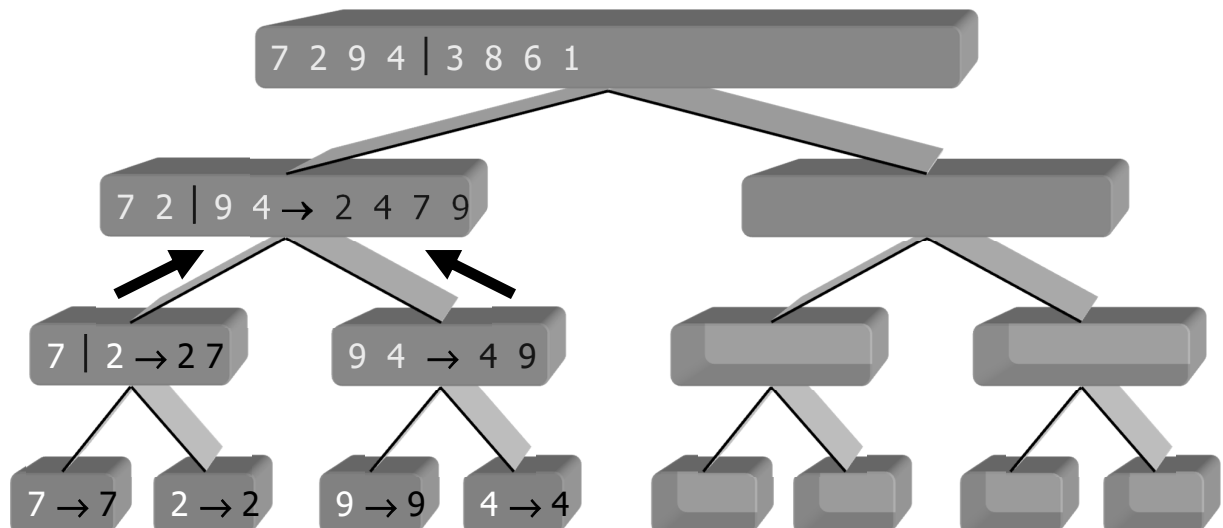
⊕ Chamada recursiva, ..., caso base, unificação



113

Ordenação por intercalação

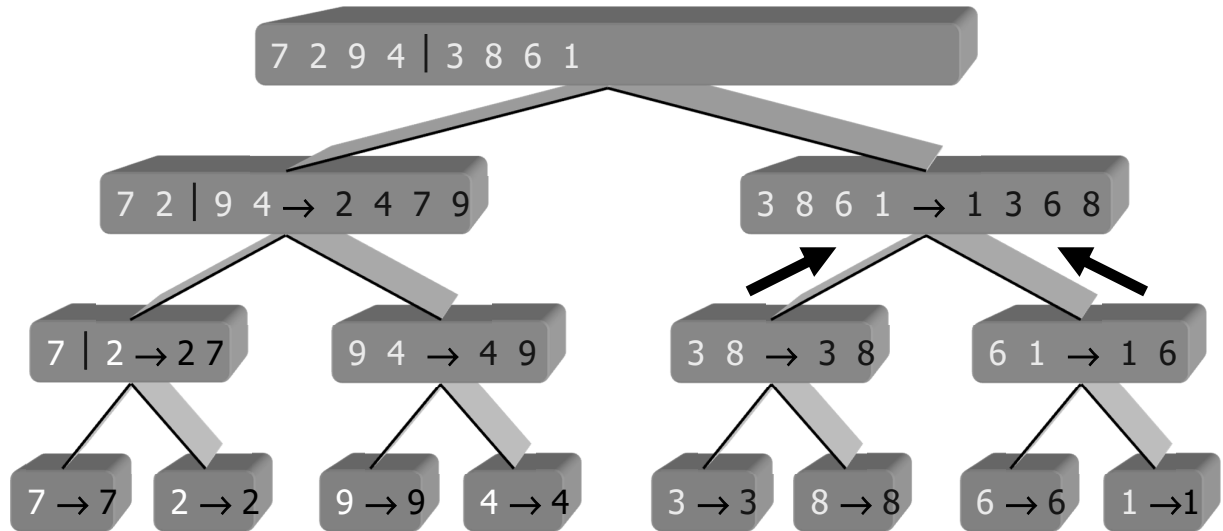
⊕ Unificação



114

Ordenação por intercalação

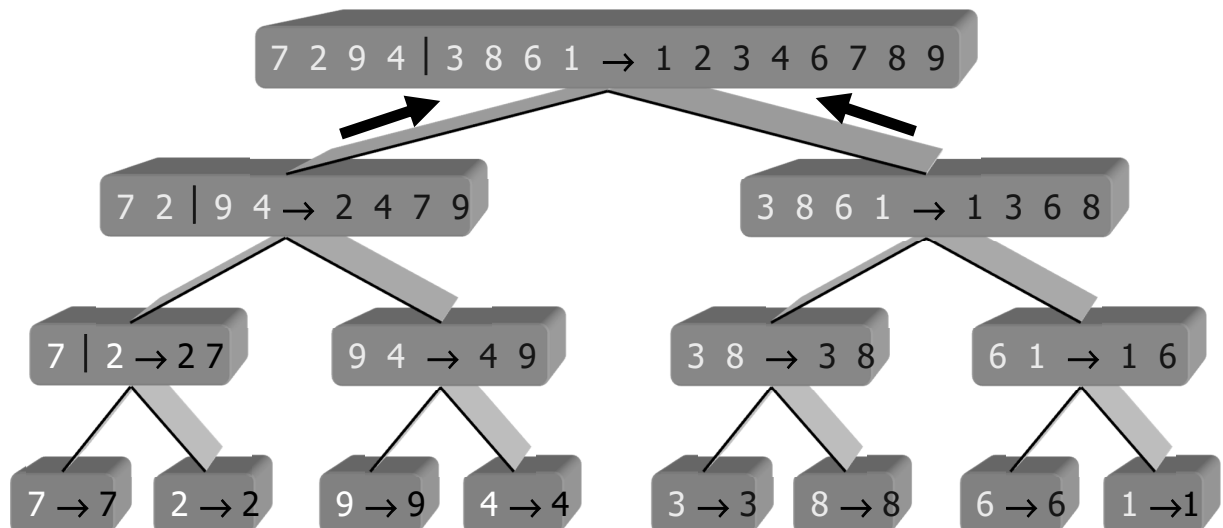
Chamada recursiva, ..., merge, merge



115

Ordenação por intercalação

Unificação



116

Ordenação por intercalação

⊞ Ordenação Rápida (*Merge Sort*)

- ⊕ A etapa de intercalação tem complexidade $O(n)$, logo, o número de comparações e de trocas é dado pela recorrência:

$$T(n) = \begin{cases} 0, & n = 1 \\ T\left[\frac{n}{2}\right] + T\left[\frac{n}{2}\right] + n, & n > 1 \end{cases}$$

- ⊕ Portanto, $O(n \log n)$ comparações e trocas são executadas no pior caso.
- ⊕ Ou seja, algoritmo *Mergesort* é assintoticamente mais eficiente que todos os anteriores.
- ⊕ Em contrapartida, o algoritmo *Mergesort* usa o dobro de memória. Ainda assim, assintoticamente o gasto de memória é equivalente ao dos demais algoritmos.

117

Ordenação

⊞ Considerações sobre Algoritmos de Ordenação

- ⊕ **Classes de Algoritmos em função de sua Complexidade:**

- ✧ **Constatação de correlação direta entre a complexidade algorítmica e eficiência relativa do algoritmo.**

- ✧ **Complexidade Algorítmica:**

- ✧ Geralmente expressa em notação **Big-O**:

- **O** ⇒ Complexidade do algoritmo.
- **n** ⇒ Tamanho do conjunto de dados processados .

118

Ordenação

⊕ Considerações sobre Algoritmos de Ordenação

⊕ Classes de Algoritmos em função de sua Complexidade:

✧ Complexidade *Quadrática* $\Rightarrow O(n^2)$:

✧ Ordenações *bubble*, *shaker*, por *inserção*, por *seleção* e *shell*.

✧ Complexidade *Logarítmica* $\Rightarrow O(n \log n)$:

✧ *Heapsort*, *mergesort* e *quicksort*.

<http://www.softpanorama.org/Algorithms/sorting.shtml>

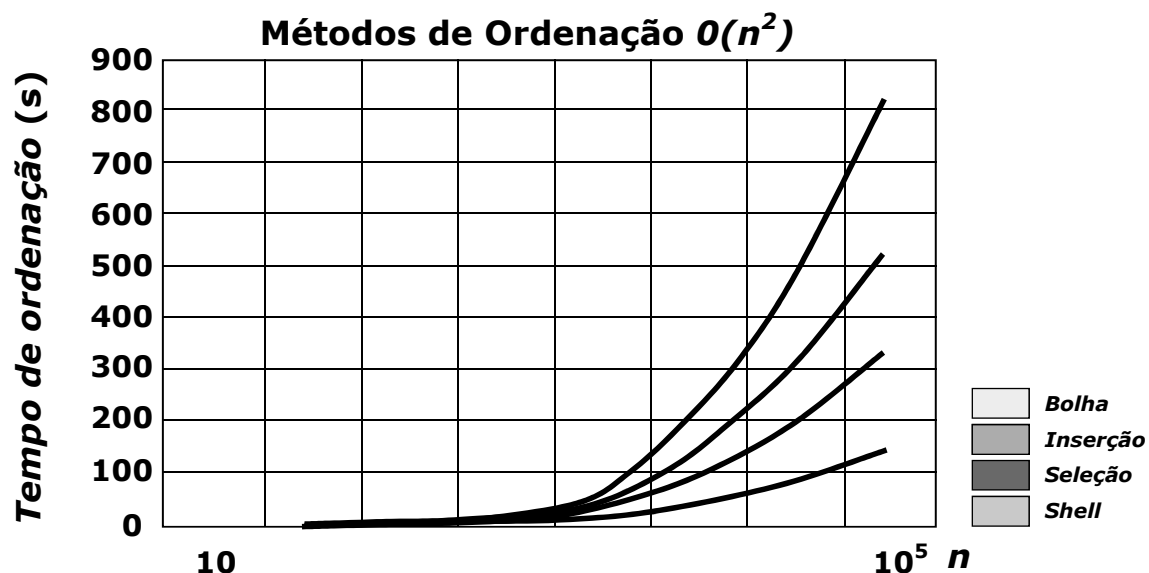
<http://www.devx.com/vb2themax/Article/19900>

119

Ordenação

⊕ Considerações sobre Algoritmos de Ordenação

⊕ Complexidade *Quadrática* $\Rightarrow O(n^2)$

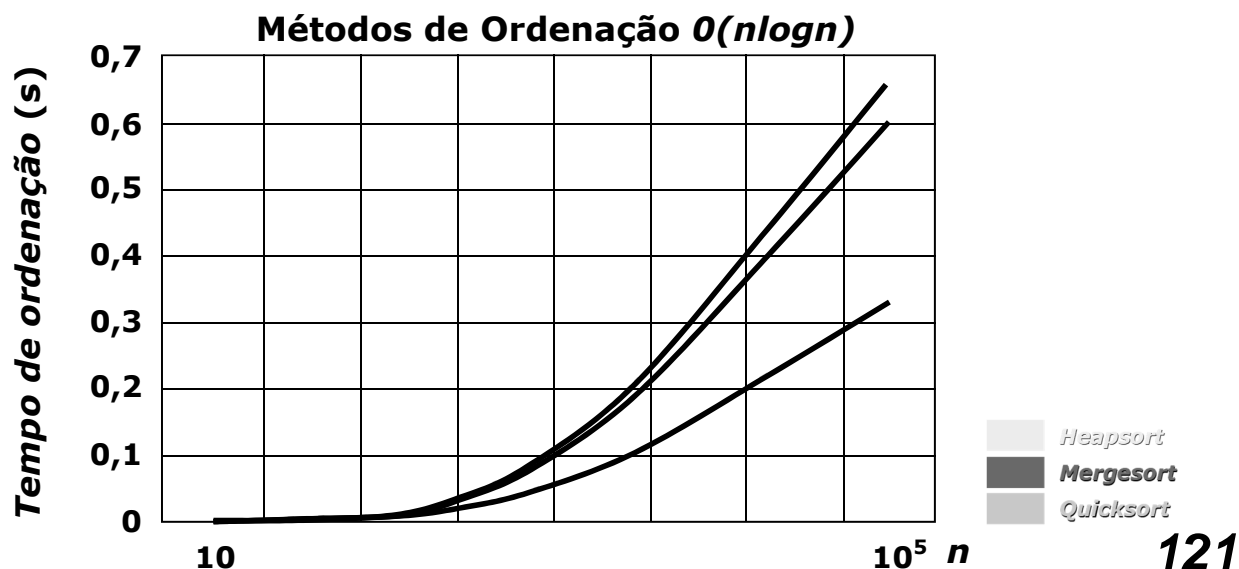


120

Ordenação

⌘ Considerações sobre Algoritmos de Ordenação

⊕ Complexidade *Logarítmica* $\Rightarrow O(n \log n)$



Ordenação

⌘ Considerações sobre Algoritmos de Ordenação

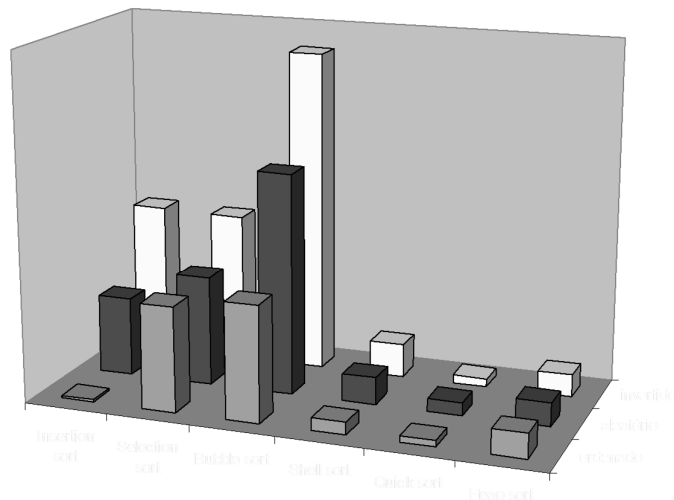
⊕ Comparação dos métodos para ordenação de *arrays*

	ordenado	aleatório	invertido
MÉTODO			
Insertion sort	12	366	704
Selection sort	489	509	695
Bubble sort	540	1026	1492
Shell sort	58	127	157
Quick sort	31	60	37
Heap sort	116	110	104

Ordenação

⊞ Considerações sobre Algoritmos de Ordenação

⊞ Comparação dos métodos para ordenação de *arrays*



123

Ordenação

⊞ Considerações sobre Algoritmos de Ordenação

⊞ O qualificativo *melhor* é função do contexto de uso.

✧ *Quicksort* pode parecer o método mais rápido:

✧ Usá-lo em uma lista de 15 itens é como matar um mosquito com uma bazuca.

⊞ Necessidade de conhecimento dos diferentes métodos de ordenação.

✧ Escolha do algoritmo mais adequado à resolução do problema de interesse.

124

Ordenação

⌘ Considerações sobre Algoritmos de Ordenação

⊕ Os algoritmos da 2ª categoria são indiscutivelmente mais rápidos do que os da 1ª categoria.

✧ Desvantagens:

- ✧ Custo computacional.
- ✧ Uso de recursão.
- ✧ Uso de *arrays* múltiplos.

125

Ordenação

⌘ A tabela abaixo apresenta quadros comparativos do tempo real para ordenar arranjos (vetores) de forma ascendente com 500, 5000, 10000 e 30000 registros que estão em ordem aleatória, ordenada e invertida, respectivamente. O método que levou menos tempo real para executar recebeu valor 1 e os outros valores são relativos a ele.

Método	aleatória				ordenada				invertida			
	500	5000	10000	30000	500	5000	10000	30000	500	5000	10000	30000
Inserção direta	11,3	87	161	-	1	1	1	1	40,3	305	575	-
Seleção direta	16,2	124	228	-	128	1524	3066	-	29,3	221	407	-
Shellsort	1,2	1,6	1,7	2	3,9	6,8	7,3	8,1	1,5	1,5	1,6	1,6
Quick sort	1	1	1	1	4,1	6,3	6,8	7,1	1	1	1	1
Heapsort	1,5	1,6	1,6	1,6	12,2	20,8	22,4	24,6	2,5	2,7	2,7	2,9

126

Ordenação

⌘ Considerações sobre Algoritmos de Ordenação – Observa-se que:

1. Shellsort, quicksort e heapsort têm praticamente a mesma ordem de grandeza para ordem aleatória e invertida.
2. O quicksort é mais rápido para todos os tamanhos aleatórios experimentados.
3. A relação heapsort/quicksort se mantém constante para todos os tamanhos, sendo o heapsort mais lento.
4. A relação shellsort/quicksort aumenta a medida que o número de elementos aumenta.
5. O shellsort é mais rápido que o heapsort para arquivos pequenos (500 registros) porém quando o tamanho do arquivo aumenta a relação se inverte.
6. A inserção é mais rápida em qualquer tamanho se os registros já estiverem ordenados (melhor caso). E é menos rápida se os registros estiverem em ordem inversa (pior caso).

127

Ordenação

⌘ Considerações sobre Algoritmos de Ordenação – Conclui-se que:

1. O método da Inserção é o mais interessante para arquivos pequenos (até 20 elementos). É estável e seu comportamento é melhor do que o método da bolha ou Bubblesort. Seu desempenho é da ordem de $O(n)$ - custo linear.
2. O método da Seleção deve ser usado para arquivos grandes não maior que 1000 registros.
3. O método Shellsort é o escolhido pela maioria das aplicações por ser muito eficiente para arquivos de até 10000 registros.
4. O método Quicksort é o algoritmo mais eficiente que existe para uma grande variedade de situações. Mas é um método frágil porque um erro de implementação é difícil de ser detectado. O algoritmo é recursivo o que demanda memória adicional. Seu desempenho é da ordem de $O(n^2)$ – custo quadrático.
5. O Heapsort é um método de ordenação elegante e eficiente. Apesar de possuir um anel interno complexo que o torna 2 vezes mais lento que o Quicksort, ele não necessita de nenhuma memória adicional. Aplicações que não podem tolerar variações no tempo esperado de execução devem usar este método.

128

Comparação entre algoritmos de ordenação

- ⊕ **Bubble Sort:** a cada passo, trocamos dois elementos de posições consecutivas no vetor, mas que estejam em ordem invertida entre si.
- ⊕ **Select Sort:** a cada passo, selecionamos o menor elemento da parte não ordenada da sequência e o colocamos no final da parte ordenada.
- ⊕ **Insert Sort:** a cada passo, inserimos ordenadamente na parte ordenada da sequência um dos elementos ainda não ordenados.
- ⊕ **Quick Sort:** a cada passo, escolhemos um pivô, separamos à sua esquerda os números menores que ele, à sua direita os maiores que ele, e por fim chamamos recursivamente o próprio **Quick Sort** para ordenar as partes à direita e à esquerda do pivô.
- ⊕ **Merge Sort:** a cada passo, dividimos o vetor em duas metades, chamamos recursivamente o **Merge Sort** para ordenar cada uma delas, e por fim as intercalamos ordenadamente, gerando um vetor ordenado.

129

Algoritmos *in-place* e estáveis

- ⊕ Algoritmos de ordenação podem ser ou não *in-place* ou *estáveis*.
- ⊕ Um algoritmo de ordenação é *in-place* se a memória adicional requerida é independente do tamanho do vetor que esta sendo ordenado.
 - ⊕ Exemplos: o **Quicksort** e o **Heapsort** são métodos de ordenação *in-place*, já o **Mergesort** e o **Counting Sort** não.
- ⊕ Um método de ordenação é estável se elementos iguais ocorrem no vetor ordenado na mesma ordem em que são passados na entrada.
 - ⊕ Exemplos: o **Counting Sort** e o **Quicksort** são exemplos de métodos estáveis (desde que certos cuidados sejam tomados na implementação). O **Heapsort** não é.

130

Métodos de ordenação interna

- ⌘ a) Ordenação por troca
 - ⊕ *Bubble Sort* (método da bolha)
 - ⊕ *Shaker Sort (Cocktail Sort)* (método oscilante)
 - ⊕ *Quick Sort* (método rápido)
- ⌘ b) Ordenação por seleção
 - ⊕ *Selection Sort* (método da seleção direta)
 - ⊕ *Heap Sort*
- ⌘ c) Ordenação por inserção
 - ⊕ *Insertion Sort* (método da inserção direta)
 - ⊕ *Shell Sort* (método dos incrementos decrescentes)
- ⌘ d) Ordenação por intercalação
 - ⊕ *Merge Sort*
 - ⊕ *Merge Array*
- ⌘ e) Ordenação por distribuição
 - ⊕ *Counting Sort* (método da contagem)
 - ⊕ *Bucket Sort (Bin Sort)* (método da distribuição de chaves)
 - ⊕ *Radix Sort (Digital Sort)* (método da raiz)

131

Ordenação por distribuição

- ⌘ Não faz comparações entre elementos para realizar a ordenação.
- ⌘ Não pode ser aplicado para a ordenação de qualquer conjunto de dados:
 - ⊕ algumas restrições devem ser atendidas
- ⌘ Ordenação em tempo linear ($O(n)$).
- ⌘ Exemplos:
 - ⊕ *Counting Sort*
 - ⊕ *BucketSort*
 - ⊕ *RadixSort*

132

Ordenação por distribuição

- ✦ **Counting Sort:** elementos são números inteiros “pequenos”; mais precisamente, inteiros x onde $x \in O(n)$.
- ✦ **Radix Sort:** elementos são números inteiros de comprimento máximo constante, isto é, independente de n .
- ✦ **Bucket Sort:** elementos são números reais uniformemente distribuídos no intervalo $[0..1)$.

133

Ordenação por distribuição

✦ Ordenação *por Contagem* (*Counting Sort*)

- ✦ Count sort é um algoritmo de ordenação que leva um tempo linear $\Theta(n)$, que é o melhor desempenho possível para um algoritmo de ordenação.
- ✦ Ele assume que cada um dos n elementos de entrada é um inteiro na faixa de 0 a k , onde k é um inteiro. Quando $k = O(n)$, a ordenação é executada no tempo $\Theta(n)$.
- ✦ É um algoritmo de ordenação estável que não usa comparação.

134

Ordenação por distribuição

⊞ Ordenação *por Contagem* (*Counting Sort*)

- ⊕ Suponha que o vetor A a ser ordenado contenha n números inteiros, todos menores ou iguais a k , onde $k \in O(n)$.
- ⊕ O algoritmo de ordenação usa dois vetores auxiliares:
 - ✧ C (contador) de tamanho k que guarda em $C[i]$ o número de ocorrências de elementos i em A .
 - ✧ B (resultado) de tamanho n onde se constrói o vetor ordenado.

135

Ordenação por distribuição

⊞ Ordenação *por Contagem* (*Counting Sort*)

- ⊕ Cada elemento do vetor contador C armazena o número de vezes que os elementos ocorrem no vetor de entrada A , começando pelo valor da menor chave (índice) até o maior valor da chave.
- ⊕ Reescreva o vetor C com as frequências acumuladas.
- ⊕ Faça interações sobre o vetor de entrada A para colocar os elementos de A no vetor de resultados B , usando o vetor contador C através do número de ocorrências como indexador.

136

Ordenação por distribuição

Ordenação *por Contagem* (*Counting Sort*)

```
void CountingSort(int A[], int B[], int k)
{
    int i, j;
    for (i=1; i<=k; i++)
        C[i] = 0;
    for (j=1; j<=n; j++)
        C[A[j]] = C[A[j]] + 1;
    for (i=2; i<=k; i++)
        C[i] = C[i] + C[i-1];
    for (j=n; j>=1; j--)
    {
        B[C[A[j]]] = A[j];
        C[A[j]] = C[A[j]] - 1;
    }
}
```

137

Ordenação por distribuição

Exemplo *Counting Sort*

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	0	0	0	0

B:					
-----------	--	--	--	--	--

for $i \leftarrow 1$ to k do
 $C[i] \leftarrow 0$

Loop 1

138

Ordenação por distribuição

⌘ Exemplo *Counting Sort*

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	0	0	0	1

B:					
-----------	--	--	--	--	--

for $j \leftarrow 1$ to n do

$C[A[j]] \leftarrow C[A[j]] + 1 \quad \triangleright C[i] = |\{\text{key} = i\}|$

Loop 2

139

Ordenação por distribuição

⌘ Exemplo *Counting Sort*

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	1	0	0	1

B:					
-----------	--	--	--	--	--

for $j \leftarrow 1$ to n do

$C[A[j]] \leftarrow C[A[j]] + 1 \quad \triangleright C[i] = |\{\text{key} = i\}|$

Loop 2

140

Ordenação por distribuição

⌘ Exemplo *Counting Sort*

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	1	0	1	1

B:					
-----------	--	--	--	--	--

for $j \leftarrow 1$ to n do

$C[A[j]] \leftarrow C[A[j]] + 1 \quad \triangleright C[i] = |\{\text{key} = i\}|$

Loop 2

141

Ordenação por distribuição

⌘ Exemplo *Counting Sort*

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	1	0	1	2

B:					
-----------	--	--	--	--	--

for $j \leftarrow 1$ to n do

$C[A[j]] \leftarrow C[A[j]] + 1 \quad \triangleright C[i] = |\{\text{key} = i\}|$

Loop 2

142

Ordenação por distribuição

⌘ Exemplo *Counting Sort*

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	1	0	2	2

B:					
-----------	--	--	--	--	--

for $j \leftarrow 1$ to n do

$C[A[j]] \leftarrow C[A[j]] + 1 \quad \triangleright C[i] = |\{\text{key} = i\}|$

Loop 2

143

Ordenação por distribuição

⌘ Exemplo *Counting Sort*

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	1	0	2	2

B:					
-----------	--	--	--	--	--

C:	1	1	2	2
-----------	---	---	---	---

for $i \leftarrow 2$ to k do

$C[i] \leftarrow C[i] + C[i-1] \quad \triangleright C[i] = |\{\text{key} \leftarrow i\}|$

Loop 3

144

Ordenação por distribuição

⌘ Exemplo *Counting Sort*

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	1	0	2	2

B:					
-----------	--	--	--	--	--

C:	1	1	3	2
-----------	---	---	---	---

for $i \leftarrow 2$ to k do

$C[i] \leftarrow C[i] + C[i-1] \quad \triangleright C[i] = |\{\text{key} \leftarrow i\}|$

Loop 3

145

Ordenação por distribuição

⌘ Exemplo *Counting Sort*

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	1	0	2	2

B:					
-----------	--	--	--	--	--

C:	1	1	3	5
-----------	---	---	---	---

for $i \leftarrow 2$ to k do

$C[i] \leftarrow C[i] + C[i-1] \quad \triangleright C[i] = |\{\text{key} \leftarrow i\}|$

Loop 3

146

Ordenação por distribuição

⌘ Exemplo *Counting Sort*

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	1	1	3	5

B:			3		
-----------	--	--	---	--	--

C:	1	1	2	5
-----------	---	---	---	---

for $j \leftarrow n$ downto 1 do
 $B[C[A[j]]] \leftarrow A[j]$
 $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4

147

Ordenação por distribuição

⌘ Exemplo *Counting Sort*

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	1	1	2	5

B:			3		4
-----------	--	--	---	--	---

C:	1	1	2	4
-----------	---	---	---	---

for $j \leftarrow n$ downto 1 do
 $B[C[A[j]]] \leftarrow A[j]$
 $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4

148

Ordenação por distribuição

⌘ Exemplo *Counting Sort*

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	1	1	2	4

B:		3	3		4
-----------	--	---	---	--	---

C:	1	1	1	4
-----------	---	---	---	---

for $j \leftarrow n$ downto 1 do
 $B[C[A[j]]] \leftarrow A[j]$
 $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4

149

Ordenação por distribuição

⌘ Exemplo *Counting Sort*

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	1	1	1	4

B:	1	3	3		4
-----------	---	---	---	--	---

C:	0	1	1	4
-----------	---	---	---	---

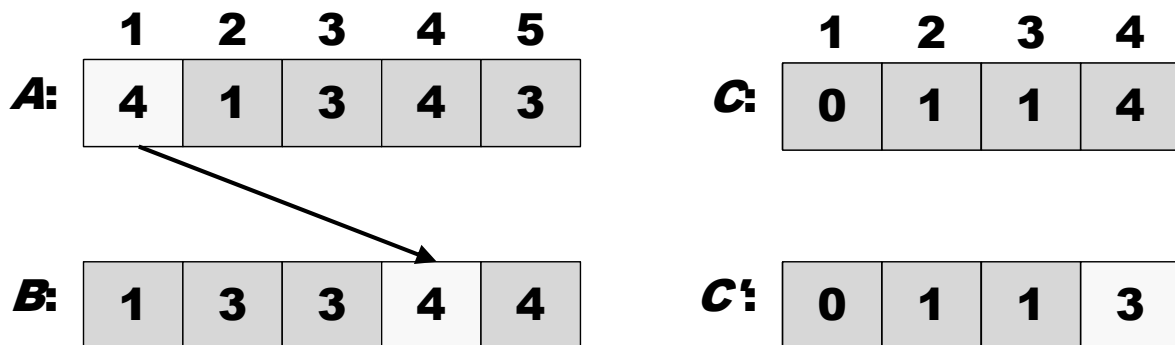
for $j \leftarrow n$ downto 1 do
 $B[C[A[j]]] \leftarrow A[j]$
 $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4

150

Ordenação por distribuição

⌘ Exemplo *Counting Sort*



```
for  $j \leftarrow n$  downto 1 do  
   $B[C[A[j]]] \leftarrow A[j]$   
   $C[A[j]] \leftarrow C[A[j]] - 1$ 
```

Loop 4

151

Ordenação por distribuição

⌘ Ordenação *por Contagem (Counting Sort)*

- ⊕ O algoritmo não faz comparações entre elementos de A.
- ⊕ Sua complexidade deve ser medida em função do número das outras operações, aritméticas, atribuições, etc.
- ⊕ Claramente, o número de tais operações é uma função em $O(n + k)$, já que temos dois *loops* simples com n iterações e dois com k iterações.
- ⊕ Assim, quando $k \in O(n)$, este algoritmo tem complexidade $O(n)$.

152

Ordenação por distribuição

⊕ Ordenação *por Contagem* (*Counting Sort*)

for (i = 1; i <= k; i++) C[i] = 0	O(k)
for (i = 1; i <= length(A); i++) C[A[j]] = C[A[j]] + 1	O(n)
for (i = 2; i <= k; i++) C[i] = C[i] + C[i-1]	O(k)
for (j = length[A]; j >= 1; j--) B[C[A[j]]] = A[j] C[A[j]] = C[A[j]] - 1	O(n)
	O(n+k)

153

Ordenação por distribuição

⊕ Ordenação *por distribuição de chaves* (*Bucket Sort*)

- ⊕ Assim como o *Counting Sort*, supõe que os n elementos a serem ordenados têm uma natureza peculiar. Neste caso, os elementos estão distribuídos uniformemente no intervalo semi-aberto $[0; 1)$.
- ⊕ A idéia é dividir o intervalo $[0; 1)$ em n segmentos de mesmo tamanho (*buckets*) e distribuir os n elementos nos seus respectivos segmentos. Como os elementos estão distribuídos uniformemente, espera-se que o número de elementos seja aproximadamente o mesmo em todos os segmentos.
- ⊕ Em seguida, os elementos de cada segmento são ordenados por um método qualquer. Finalmente, os segmentos ordenados são concatenados em ordem crescente.

154

Ordenação por distribuição

⊕ Ordenação *por distribuição de chaves* (*Bucket Sort*)

⊕ Restrições:

- ✧ ordena somente elementos que são números inteiros;
 - ✧ m depende do conjunto de dígitos ($m = 10$, em geral);
- ✧ considera um pequeno conjunto de dados que não ultrapassa um valor máximo pequeno m :
 - ✧ valor dos elementos encontra-se no intervalo $[0, m]$

⊕ Exemplo:

- ✧ ordenação dos 80 empregados da empresa pelo seu tempo de serviço (em anos):
 - ✧ $n = 80$
 - ✧ $m = 50$

155

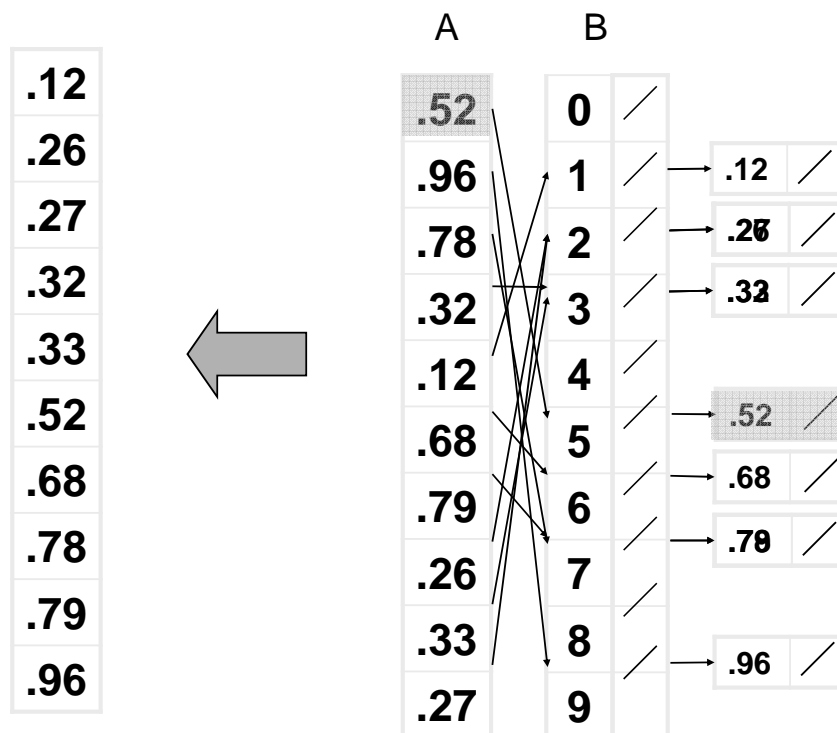
Ordenação por distribuição

⊕ Ordenação *por distribuição de chaves* (*Bucket Sort*)

- ⊕ Utiliza um vetor auxiliar que mantém o número de ocorrências de cada valor de elemento.
- ⊕ Distribui os valores dos elementos ordenadamente no vetor com base nos números de ocorrências no vetor auxiliar.

156

Ordenação por distribuição



Exemplo *Bucket Sort* 157

Ordenação por distribuição

⌘ **Algoritmo: Ordenação *por distribuição de chaves* (*Bucket Sort*)**

```
Bucket-Sort(A)
{
  n ← tamanho(A)
  for i ← 0 to n do
    insere A[i] dentro da lista B[floor(n*A[i])]
  for i ← 0 to n-1 do
    Selection-Sort(B[i]) OU Insertion-Sort(B[i])
  Concatena listas B[0], B[1], ... B[n-1] na ordem
  Retorna a lista concatenada
}
```

Ordenação por distribuição

⊕ Ordenação *por distribuição de chaves* (*Bucket Sort*)

for i = 1 to n do B[i] = nil	O(n)
for i = 1 to n do B[⌊nA[i]⌋] = A[i]	O(n)
for i = 1 to n do B[⌊nA[i]⌋] = A[i]	O(n)
for i = 0 to n - 1 do Selection_Sort (B[i])	O(n)
Merge_List B[0], ..., B[n-1]	O(n)

159

Ordenação por distribuição

⊕ Ordenação *por distribuição de chaves* (*Bucket Sort*)

⊕ **Bucket Sort** – Complexidades envolvidas:

✧ $O(m)$, $O(n)$ e $O(m+n)$

⊕ Considerando que m deve ser pequeno, sua complexidade é assumida como linear no número de dados ($m = O(n)$).

⊕ Complexidade do *BucketSort*: $O(n)$.

160

Ordenação por distribuição

⌘ Ordenação *por Raízes* (*Radix Sort*)

- ⊕ *Radix sort* é um algoritmo de distribuição em vários passos para um escaninho (*bucket*), de acordo com parte da chave do item.
 - ✧ Após cada passo, itens são reunidos dos *buckets*, mantidos em ordem, e então redistribuídos de acordo com a próxima parte mais significativa da chave.
- ⊕ Ordena chaves dígito-a-dígito (também chamado de *digital sort*), ou, se as chaves forem cadeias de caracteres que querem ser ordenadas alfabeticamente, ordena caracter-por-caracter.

161

Ordenação por distribuição

⌘ Ordenação *por Raízes* (*Radix Sort*)

- ⊕ Foi usado me máquinas de ordenar cartões.
- ⊕ *Radix sort* usa *bucket* ou *counting sort* como o algoritmo de ordenação, onde a ordem relativa inicial de chaves iguais não é mudada.
- ⊕ Representações de inteiros podem ser usadas para representar cadeias de caracteres ou números inteiros. Assim, qualquer coisa que possa ser representada por inteiros pode ser rearranjada para ser ordenada pelo *radix sort*.

162

Ordenação por distribuição

⊞ Ordenação *por Raízes* (*Radix Sort*)

- ⊞ Algoritmo de ordenação por distribuição que ordena com base nos dígitos de um número:
 - ✧ prioriza (inicia por) dígitos menos significativos.
- ⊞ Comparação com o *BucketSort*:
 - ✧ contabiliza também ocorrências de elementos;
 - ✧ melhor desempenho médio:
 - ✧ m não está associado a um valor máximo previsto para um conjunto de elementos e sim ao conjunto de dígitos que podem existir em um número.
 - Sistema decimal: $m = 10$ (m é pequeno!)
 - ✧ o número de iterações do algoritmo depende do número máximo de dígitos (d) que um número pode ter (d é pequeno, em geral).

163

Ordenação por distribuição

⊞ Ordenação *por Raízes* (*Radix Sort*)

- ⊞ Classificação baseada em como trabalha internamente:
 - ✧ *least significant digit (LSD) radix sort*: o processamento começa pelo dígito menos significativo e se move em direção ao dígito mais significativo.
 - ✧ *most significant digit (MSD) radix sort (RadixExchange Sort)*: o processamento começa pelo dígito mais significativo e se move em direção ao dígito menos significativo. Isto é recursivo.

164

Ordenação por distribuição

⊕ Ordenação *por Raízes* (*Radix Sort*)

⊕ As diferenças entre a os *radix sort* *LSD* e *MSD* são:

- ✧ No *MSD*, se é conhecido o número mínimo de caracteres necessários para distinguir todas as cadeias de caracteres, pode-se apenas ordenar este número de caracteres.
- ✧ A abordagem *LSD* requer o preenchimento de chaves curtas o comprimento da chave for variável, e garante que todos os dígitos serão examinados mesmo se os primeiros 3-4 dígitos contiverem toda a informação necessária para realizar a ordenação.
- ✧ *MSD* é recursivo. *LSD* é não-recursivo.
- ✧ *MSD radix sort* requer muito mais memória para ordenar elementos. *LSD radix sort* é a implementação preferida entre os dois.

165

Ordenação por distribuição

⊕ Ordenação *por Raízes* (*Radix Sort*)

- ⊕ *MSD recursive radix sorting* tem aplicações na computação paralela, de modo que cada um dos *sub-buckets* pode ser ordenado independentemente do resto. Cada recursão pode ser entregue para o próximo processador disponível.
- ⊕ O *Postman's sort* é uma variação do *MSD radix sort*, onde os atributos da chave são descritos de modo que o algoritmo possa alocar *buckets* eficientemente. Este é o algoritmo usado pelas máquinas de ordenação de cartas nos correios: primeiro estados, então caixa postal, então rua, etc. Os menores *buckets* são então recursivamente ordenados.

166

Ordenação por distribuição

⌘ Ordenação *por Raízes* (*Radix Sort*)

⊕ Vantagens:

- ✧ *Radix* e *bucket sorts* são estáveis, preservando a ordem de existência de chaves iguais.
- ✧ Eles trabalham em tempo linear, diferente da maior parte dos algoritmos de ordenação. Isto é, eles não pioram quando um grande número de itens precisam ser ordenados. A maioria dos ordenadores tem tempo de execução em $O(n \log n)$ ou $O(n^2)$.
- ✧ O tempo de ordenação por item é constante, assim nenhuma comparação entre itens é feita. Em outros algoritmos de ordenação, o tempo de ordenação aumenta com o número de itens.
- ✧ *Radix sort* é particularmente eficiente quando se tem um grande número de registros com chaves curtas. 167

Ordenação por distribuição

⌘ Ordenação *por Raízes* (*Radix Sort*)

⊕ Desvantagens:

- ✧ *Radix* e *bucket sorts* não trabalham bem quando as chaves são muito longas, assim o tempo de ordenação total é proporcional ao comprimento da chave e do número de itens a ordenar.
- ✧ Eles não são “*in-place*”, usando mais memória de trabalho que uma ordenação tradicional.

Ordenação por distribuição

⊞ Ordenação *por Raízes* (*Radix Sort*)

⊕ Utiliza dois vetores auxiliares:

- ✧ *C* (número de ocorrências de elementos);
- ✧ *T* (temporário - mantém a ordenação do vetor por um certo dígito).

⊕ Realiza *p* iterações. Na *i*-ésima iteração:

- ✧ ordena-se o vetor pelo *i*-ésimo dígito menos significativo;
- ✧ *C* recebe o número de ocorrências de cada *i*-ésimo dígito;
- ✧ uma vez preenchido o *C*, são contabilizados nele os *offsets* para cada dígito (posições onde iniciam elementos que possuem um certo valor de dígito);
- ✧ com base nestes *offsets*, *T* recebe os elementos do vetor ordenado pelo *i*-ésimo dígito;
- ✧ transfere-se os elementos de *T* para o vetor final.

169

Ordenação por distribuição

⊞ Ordenação *por Raízes* (*Radix Sort*)

```
RadixSort(A, d)    (* LSB radix sort *)
{
    for i = 1 to d
        Ordenar A no dígito i com algoritmo de
        ordenação estável
}
```

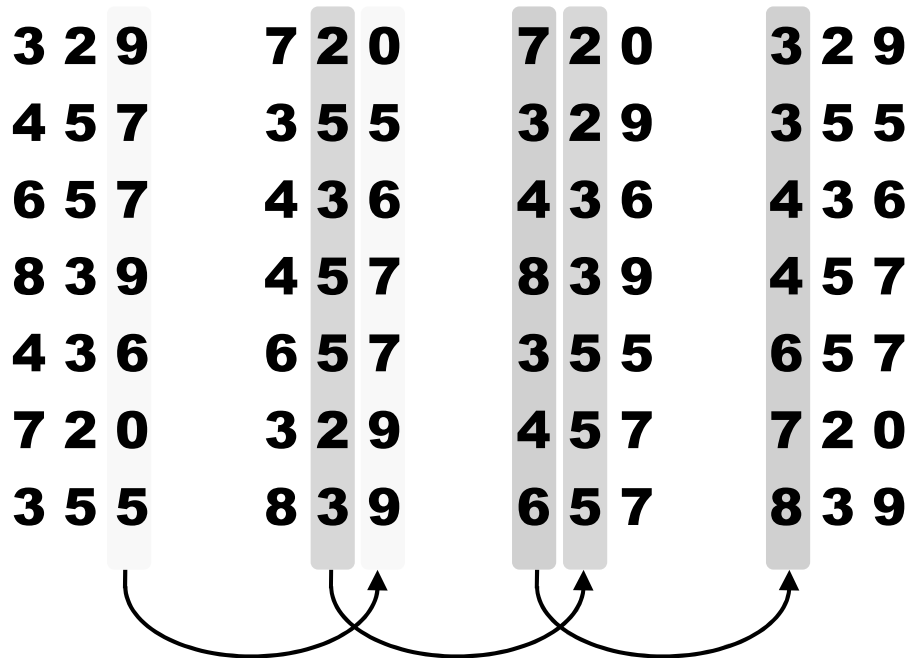
⊞ Se utilizar *CountingSort*

- ⊕ Complexidade: $O(d(n + m))$
- ⊕ Não ordena no lugar, i.e. requer memória adicional...

170

Ordenação por distribuição

⊕ Exemplo *Radix Sort*



171

Ordenação por distribuição

⊕ Ordenação *por Raízes (Radix Sort)*

⊕ Complexidades envolvidas:

- ✧ $O(m.d)$ e $O(n.d) \Rightarrow O(d(m+n))$
- ✧ considerando que m e d são pequenos ($m = 10$ e $d \leq 10$), sua complexidade é assumida como linear no número de dados.

⊕ Complexidade do *RadixSort*: $O(n)$

172

Ordenação por distribuição

⊕ Ordenação *por Raízes (Radix Sort)*

⊕ O tempo de execução para este exemplo é:
 $T(n) = O(d(n+m))$

- ✧ m = número de *buckets* = 10 (0 a 9).
- ✧ n = número de elementos a ordenar = 15
- ✧ d = dígitos ou tamanho má. do elemento = 2

⊕ Neste exemplo o algoritmo levará:

$$\begin{aligned} T(n) &= O(d(n+m)) \\ &= O(2(15+10)) \\ &= O(50) \text{ tempo de execução.} \end{aligned}$$

173

Ordenação por distribuição

⊕ Ordenação *por Raízes (RadixExchange Sort)*

⊕ *Most significant digit (MSD) radix sort*

⊕ **Algoritmo:**

- ✧ Se é ordenação de cadeias de caracteres, deve-se criar um *bucket* para 'a', 'b', 'c' até 'z'.
- ✧ Após o primeiro passo, as cadeias de caracteres são grosseiramente ordenadas de modo que duas cadeias de caracteres quaisquer que comecem com letras diferentes estão na ordem correta.
- ✧ Se um escaninho tem mais de uma cadeia de caracter, seus elementos são recursivamente ordenados (ordenação nos *buckets* pelo próximo caracter mais significativo).
- ✧ Os conteúdos dos *buckets* são concatenados.

174

Ordenação por distribuição

✚ Ordenação *por Raízes (MSD)*

```
#define bitsword 32
#define bitsbyte 8
#define bytesword 4
#define R (1 << bitsbyte)
#define digit(A, B) (((A) >> (bitsword-((B)+1)*bitsbyte)) & (R-1))
// Para strings:
// #define digit(A, B) A[B]

#define bin(A) 1+count[A]
void radixMSD(Item a[], int l, int r, int w)
{
    int i, j, count[R+1];
    if (w > bytesword) return;
    if (r-l <= M) { insertion(a, l, r); return; }
    for (j = 0; j < R; j++) count[j] = 0;
    for (i = l; i <= r; i++)
        count[digit(a[i], w) + 1]++;
    for (j = 1; j < R; j++)
        count[j] += count[j-1];
    for (i = l; i <= r; i++)
        aux[l+count[digit(a[i], w)]] = a[i];
    for (i = l; i <= r; i++) a[i] = aux[i];
    radixMSD(a, l, bin(0)-1, w+1);
    for (j = 0; j < R-1; j++)
        radixMSD(a, bin(j), bin(j+1)-1, w+1);
}
```

175

Ordenação por distribuição

✚ Ordenação *por Raízes (LSD)*

```
#define bitsword 32
#define bitsbyte 8
#define bytesword 4
#define R (1 << bitsbyte)
#define digit(A, B) (((A) >> (bitsword-((B)+1)*bitsbyte)) & (R-1))
// Para strings:
// #define digit(A, B) A[B]

void radixLSD(Item a[], int l, int r)
{
    int i, j, w, count[R+1];
    for (w = bytesword-1; w >= 0; w--)
    {
        for (j = 0; j < R; j++) count[j] = 0;
        for (i = l; i <= r; i++)
            count[digit(a[i], w) + 1]++;
        for (j = 1; j < R; j++)
            count[j] += count[j-1];
        for (i = l; i <= r; i++)
            aux[count[digit(a[i], w)]] = a[i];
        for (i = l; i <= r; i++) a[i] = aux[i];
    }
}
```

176

Ordenação por distribuição

✚ Ordenação *por Raízes* (RadixExchange Sort)

```
void radix(int x[], int n)
{
    int k,i,j,ndig,y;
    Inic_Vetor(F,Livre);
    for(k=0; k < 10; k++ ) Inic_Fila(frente[k],fim[k]);
    ndig = num_digitos(x[pega_maior(x,n)]);
    for(k=0; k < ndig; k++)
    {
        for(i=0; i < n; i++)
        {
            y = pega_digito(x[i],k+1);
            Ins_Fila(F,x[i],frente[y],fim[y],Livre);
        }
        j = 0;
        for(i=0; i < 10; i++)
        {
            while( !Fila_Vazia(fim[i]) )
            {
                Del_Fila(F,y,frente[i],fim[i],Livre);
                x[j++] = y;
            }
        }
        for(i=0; i < n; i++ ) cout << x[i] << " ";
        cout << endl;
    }
}
```

177

Ordenação por distribuição

A 00001	A 00001	A 00001	A 00001	A 00001	A 00001
S 10011	E 00101	E 00101	E 00101	E 00101	E 00101
O 01111	O 01111	A 00001	E 00101	E 00101	E 00101
R 10010	L 01100	E 00101	G 00111	G 00111	G 00111
T 10100	M 01101	I 01001	I 01001	I 01001	I 01001
I 01001	I 01001	N 01110	N 01110	N 01110	N 01110
N 01110	N 01110	M 01101	L 01100	L 01100	L 01100
G 00111	G 00111	O 01111	S 10011	S 10011	S 10011
E 00101	E 00101	X 11000	T 10100	T 10100	T 10100
X 11000	A 00001	P 10000	P 10000	P 10000	P 10000
A 00001	X 11000	R 10010	R 10010	R 10010	R 10010
M 01101	T 10100	P 10000	P 10000	P 10000	P 10000
P 10000	P 10000	R 10010	R 10010	R 10010	R 10010
L 01100	R 10010	X 11000	X 11000	X 11000	X 11000
E 00101	S 10011				

Figure 10.2 Radix exchange

✚ Ordenação *por Raízes*

A 00001	R 10010	T 10100	X 11000	P 10000	A 00001
S 10011	T 10100	X 11000	P 10000	A 00001	A 00001
O 01111	N 01110	P 10000	A 00001	E 00101	E 00101
R 10010	X 11000	L 01100	I 01001	R 10010	E 00101
T 10100	P 10000	A 00001	A 00001	S 10011	G 00111
I 01001	L 01100	I 01001	R 10010	T 10100	I 01001
N 01110	A 00001	E 00101	S 10011	E 00101	L 01100
G 00111	S 10011	A 00001	T 10100	E 00101	M 01101
E 00101	O 01111	M 01101	L 01100	G 00111	N 01110
X 11000	I 01001	E 00101	E 00101	X 11000	O 01111
A 00001	G 00111	R 10010	M 01101	I 01001	P 10000
M 01101	E 00101	N 01110	E 00101	L 01100	R 10010
P 10000	A 00001	S 10011	N 01110	M 01101	S 10011
L 01100	M 01101	O 01111	O 01111	N 01110	T 10100
E 00101	E 00101	G 00111	G 00111	O 01111	X 11000

Figure 10.4 Straight radix sort ("right-to-left" radix sort).

Complexidade dos algoritmos

Algoritmos de ordenação com $O(N^2)$

- ✚ Bubble Sort
- ✚ Shaker Sort
- ✚ Selection Sort
- ✚ Insertion Sort
- ✚ Shell Sort*
- ✚ Comb Sort

Algoritmos de ordenação com $O(N \log N)$

- ✚ Heap Sort
- ✚ Merge Sort
- ✚ Quick Sort

Algoritmos de ordenação com $O(N)$

- ✚ Counting Sort
- ✚ Bucket Sort
- ✚ Radix Sort

179

Comparação dos algoritmos de ordenação

- ✚ *Insertion sort*: adequado apenas para n pequeno.
- ✚ *Merge sort*: garante ser rápido mesmo em seu pior caso; estável.
- ✚ *Heapsort*: requer um mínimo de memória e garante execução rápida; tanto o tempo médio como o máximo levam em torno do dobro do tempo médio do *quicksort*.
- ✚ *Quicksort*: mais útil para ordenação de propósito geral por exigir memória muito pequena e tempo médio mais rápido (na prática, escolher a mediana de três elementos como pivô).
- ✚ *Counting sort*: muito útil quando as chaves têm um range pequeno; estável; espaço de memória para contadores e para $2n$ registros.
- ✚ *Radix sort*: apropriado tanto para chaves com sequências curtas como para ordenar sequências lexicográficas.
- ✚ *Bucket sort*: assume que as chaves têm distribuição uniforme.

180

Outros algoritmos de ordenação

- | | |
|---|----------------------------|
| ⊕ American flag sort | ⊕ Oet sort |
| ⊕ Bingo sort | ⊕ Partition sort |
| ⊕ Bitonic sort | ⊕ Perm sort |
| ⊕ Bogo sort (stupid sort, bozo sort, awful sort, tennis court, blort sort, monkey sort, random sort e drunk man sort) | ⊕ Pigeonhole sort |
| ⊕ Brick sort | ⊕ Plasel sort |
| ⊕ Btm sort | ⊕ Postman's sort |
| ⊕ Comb sort | ⊕ Q sort |
| ⊕ Diminishing increment sort | ⊕ QM sort |
| ⊕ Distributive partitioning sort | ⊕ Range sort |
| ⊕ Exchange sort | ⊕ Rapid sort |
| ⊕ Gnome sort | ⊕ Restricted universe sort |
| ⊕ Integer sort | ⊕ Several unique sort |
| ⊕ Intro sort | ⊕ Shear sort |
| ⊕ Histogram sort | ⊕ Shuffle sort |
| ⊕ J sort | ⊕ Simple sort |
| ⊕ Jump sort | ⊕ Smooth sort |
| ⊕ Linear probing sort | ⊕ Stooge sort |
| ⊕ Lucky sort | ⊕ Strand sort |
| ⊕ Ms sort | ⊕ Sunrise sort |
| ⊕ Odd-Even transposition sort | ⊕ Swap sort |
| | ⊕ Trippel sort |
| | ⊕ UnShuffle sort |

181

Bibliografia

- ⊕ Paulo A. Azeredo. *Métodos de Classificação de Dados e Análise de suas Complexidades*. Editora Campus, 1996.
- ⊕ Robert Sedgewick. *Algorithms in C++*, Addison-Wesley, 1992.
- ⊕ William Ford and William Topp. *Data Structures with C++*, Prentice Hall, 1996.
- ⊕ <http://walderson.com/site/>

182