

# Ordenação

Estruturas de Dados e Algoritmos – Ciência da Computação



Prof. Daniel Saad Nogueira  
Nunes

Instituto Federal de Brasília,  
Câmpus Taguatinga



# Sumário

---

- 1 Introdução
- 2 Ordenação
- 3 Ordenação por Comparações
- 4 Ordenação em  $O(n)$
- 5 Comparações
- 6 Links



# Sumário

---

## 1 Introdução



# Introdução

---

## Ordenação

- O problema da ordenação é fundamental para a Ciência da Computação. Através da resolução deste problema, podemos solucionar diversos outros.
- Formalmente o problema é postulado como, dado uma lista de elementos, ordenar cada elemento segundo uma relação de ordem  $<$  de maneira crescente. Ou seja, temos:
  - ▶ Entrada: Sequência de elementos  $\{a_0, a_1, \dots, a_{n-1}\}$ .
  - ▶ Saída: Permutação da sequência original em ordem crescente, isto é,  $\{a'_0, a'_1, \dots, a'_{n-1}\}$ ,  $a'_i < a'_{i+1}$ ,  $0 \leq i < n - 2$ .



# Introdução

---

## Exemplo

- Lista de inteiros a ser ordenada segundo a relação  $\leq$  sobre  $\mathbb{N}$ .
- Lista de reais a ser ordenada segundo a relação  $\leq$  sobre  $\mathbb{R}$ .
- Lista de palavras a ser ordenada segundo a ordem lexicográfica induzida sobre um alfabeto.



# Introdução

---

## Ordenação

- Existem diversos métodos de ordenação diferentes, cada qual com sua técnica.
- No entanto, métodos de ordenação podem compartilhar algumas propriedades:
  - ▶ In-place: Usa-se a entrada e mais um número constante de posições de memória para executar a ordenação ( $n + O(1)$ ).
  - ▶ Estável: Se dois elementos  $v[i]$  e  $v[j]$  são iguais, com  $i < j$ , eles terão a mesma posição relativa após a ordenação, isto é, o elemento  $v[i]$  vai vir antes de  $v[j]$  no vetor ordenado, apesar de terem o mesmo valor.
    - (7, 2, 1, 2, 4, 3, 6, 5)  $\rightarrow$  (1, 2, 2, 3, 4, 5, 6, 7)



# Sumário

---

## 2 Ordenação



# Sumário

---

## 2 Ordenação

- Bubblesort
- Insertionsort
- Mergesort
- Quicksort
- Heapsort





# Bubblesort

---

## Bubblesort

- O Bubblesort, em cada iteração, lê o vetor da esquerda para a direita e troca os elementos se  $v[i] > v[i + 1]$ .
- Como consequência disso, os maiores elementos são colocados em sua posição devida após cada iteração.
- Observe que são necessárias  $n - 1$  iterações para o algoritmo ordenar a sequência original, sendo que cada iteração precisa passar por toda a sequência.



# Bubblesort

## Exemplo

11	17	23	2	7	29	3	13	5	19
11	17	2	23	7	29	3	13	5	19
11	17	2	7	23	29	3	13	5	19
11	17	2	7	23	3	29	13	5	19
11	17	2	7	23	3	13	29	5	19
11	17	2	7	23	3	13	5	29	19
11	17	2	7	23	3	13	5	19	29

11	2	17	7	23	3	13	5	19	29
11	2	7	17	23	3	13	5	19	29
11	2	7	17	3	23	13	5	19	29
11	2	7	17	3	13	5	23	19	29
11	2	7	17	3	13	5	19	23	29



# Bubblesort

## Exemplo

11	2	7	17	3	13	5	19	23	29
2	11	7	17	3	13	5	19	23	29
2	7	11	17	3	13	5	19	23	29
2	7	11	3	17	13	5	19	23	29
2	7	11	3	13	17	5	19	23	29
2	7	11	3	13	5	17	19	23	29

2	7	3	11	13	5	17	19	23	29
2	7	3	11	5	13	17	19	23	29

2	3	7	11	5	13	17	19	23	29
2	3	7	5	11	13	17	19	23	29

2	3	5	7	11	13	17	19	23	29
2	3	5	7	11	13	17	19	23	29



# Bubblesort

---

## Exemplo



# Bubblesort

## Exemplo

5	1	12	-5	16	unsorted
5	1	12	-5	16	5 > 1, swap
1	5	12	-5	16	5 < 12, ok
1	5	12	-5	16	12 > -5, swap
1	5	-5	12	16	12 < 16, ok
1	5	-5	12	16	1 < 5, ok
1	5	-5	12	16	5 > -5, swap
1	-5	5	12	16	5 < 12, ok
1	-5	5	12	16	1 > -5, swap
-5	1	5	12	16	1 < 5, ok
-5	1	5	12	16	-5 < 1, ok
-5	1	5	12	16	sorted



# Bubblesort

---

---

## Function Bubblesort

---

**Input:**  $V$

**Output:**  $V$ ,  $V[i] < V[i + 1], 0 \leq i < n - 1$

```
1 trocou ← true
2 for(  $i \leftarrow 0; i < V.SIZE() \wedge \textit{trocou} = \textbf{true}; i++$  )
3   trocou ← false
4   for(  $j \leftarrow 0; j < V.SIZE() - 1; j++$  )
5     if(  $V[j] > V[j + 1]$  )
6       SWAP( $V[j], V[j + 1]$ )
7       trocou ← true
```

---



# Bubblesort

---





# Bubblesort

---

## Análise

No pior caso, são necessários  $n - 1$  iterações sobre a sequência original. Na iteração  $i$  são realizadas  $n - 1 - i$  comparações ao todo. Portanto, o custo do algoritmo é dado como:

$$\sum_{i=0}^{n-1} i = 1 + 2 + \dots + n - 1 \in \Theta(n^2)$$

In-place	Estável
✓	✓





# Sumário

---

## 2 Ordenação

- Bubblesort
- Insertionsort
- Mergesort
- Quicksort
- Heapsort



# Insertionsort

---

## Insertionsort

O projeto do algoritmo Insertionsort segue um argumento análogo à indução matemática.

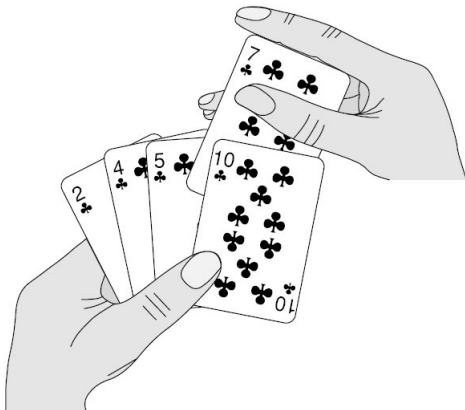
- Caso base: uma sequência com um elemento está ordenada.
- Passo de indução: a inserção de um elemento em uma sequência ordenada na posição correta também gera uma sequência ordenada.



# Insertionsort

---

## Analogia com Baralho





# Insertionsort

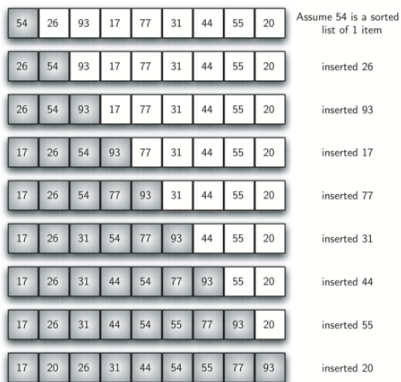
---

## Exemplo



# Insertionsort

## Exemplo





# Insertionsort

---

---

**Function** Insertionsort

---

**Input:**  $V$

**Output:**  $V, \quad V[i] < V[i + 1], 0 \leq i < V.SIZE() - 1$

```
1 for(  $i \leftarrow 1; i < V.SIZE(); i++$  )
2    $chave \leftarrow V[i]$ 
3   for(  $j \leftarrow i - 1; j \geq 0 \wedge V[j] > chave; j--$  )
4      $V[j + 1] \leftarrow V[j]$ 
5    $V[j + 1] \leftarrow chave$ 
```

---



# Insertionsort

---





# Insertionsort

---

## Análise

- No pior caso, são necessários  $n - 1$  iterações sobre a sequência original. Na iteração  $i$  são realizadas no máximo,  $i$  comparações ao todo.
- A inserção do elemento na posição correta, também necessita de  $i$  operações de troca (em vetores). Portanto, o número de comparações do algoritmo (o mesmo número de trocas em vetores) é dado como:

$$\sum_{i=0}^{n-1} i = 1 + 2 + \dots + n - 1 \in \Theta(n^2)$$





# Insertionsort

---

## Observação

- Eficiente para entradas pequenas.
- Mais rápido na prática do que outros algoritmos quadráticos (como o Bubblesort).

In-place	Estável
✓	✓



# Sumário

---

## 2 Ordenação

- Bubblesort
- Insertionsort
- Mergesort
- Quicksort
- Heapsort



# Mergesort

---

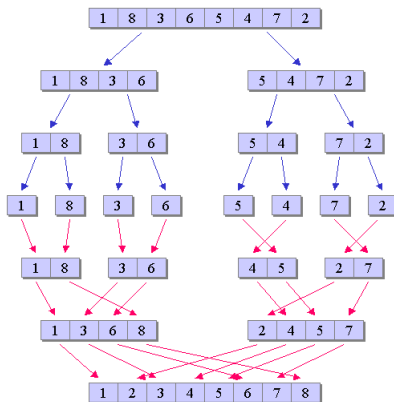
## Mergesort

- O Mergesort se baseia no conceito de Merge (junção) de duas sequências ordenadas. Primeiramente ele subdivide a sequência original na metade e ordena recursivamente essas sequências.
- Por fim, faz a junção das duas sequências ordenadas para compor uma sequência maior ordenada.
- $(1, 3, 5, 7, 9) + (0, 2, 4, 6, 8) \xrightarrow{\text{merge}} (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)$



# Mergesort

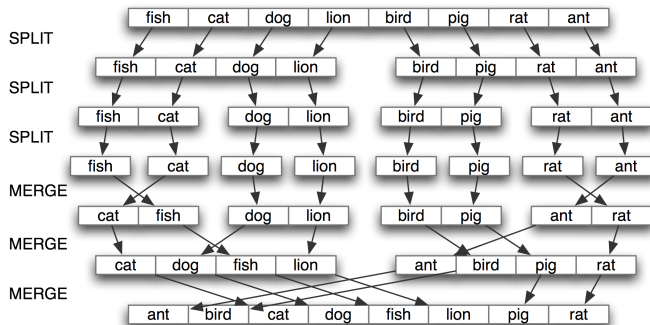
## Exemplo





# Mergesort

## Exemplo





# Mergesort

---

---

## Function Mergesort

---

**Input:**  $(V, i, j)$

**Output:**  $V, \quad V[i] < V[i + 1], 0 \leq i < V.SIZE() - 1$

```
1 if(  $i \geq j$  )  
2   | return  $V[i, j]$ ;  
3  $m \leftarrow (i + j)/2$   
4  $V_1 \leftarrow \text{MERGESORT}(V, i, m)$   
5  $V_2 \leftarrow \text{MERGESORT}(V, m + 1, j)$   
6  $V \leftarrow \text{MERGE}(V_1, V_2)$   
7 return  $V$ 
```

---



# Mergesort

---





# Mergesort

---

---

## Function Merge

---

**Input:**  $V, V_1, V_2$

**Output:**  $V, \quad V[i] < V[i + 1], 0 \leq i < n - 1$

```
1 for(  $i \leftarrow 0, j \leftarrow 0, k \leftarrow 0; j < V_1.SIZE() \wedge k < V_2.SIZE(); i++$  )
2   |   if(  $V_1[j] \leq V_2[k]$  )
3   |     |  $V[i] \leftarrow V_1[j++]$ 
4   |   else
5   |     |  $V[i] \leftarrow V_2[k++]$ 
6 while  $j < V_1.SIZE()$  do  $V[i++] \leftarrow V_1[j++]$ 
7 while  $k < V_2.SIZE()$  do  $V[i++] \leftarrow V_2[k++]$ 
```

---





# Mergesort

## Análise

A relação de recorrência do Mergesort corresponde à:

$$T(n) = 2 \cdot T(n/2) + O(n) \in \Theta(n \lg n)$$

In-place	Estável
✗	✓

## Observação

- Requer uma quantidade de memória superior a  $O(1)$  (vetores auxiliares).
- Recursivo!



# Sumário

---

## 2 Ordenação

- Bubblesort
- Insertionsort
- Mergesort
- Quicksort
- Heapsort



# Quicksort

---

## Quicksort

O Quicksort se baseia na escolha de um pivô. Após escolhido este pivô, a sequência original é particionada em três partes:

- 1 Elementos menores que o pivô;
- 2 Pivô;
- 3 Elementos maiores que o pivô;

O procedimento é aplicado recursivamente na primeira e última partes.



# Quicksort

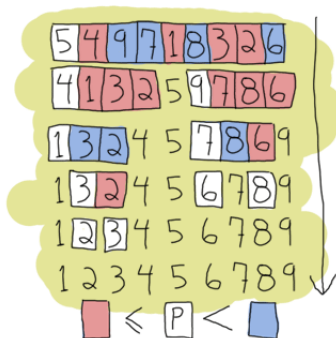
---

## Exemplo



# Quicksort

exampleblock





# Quicksort

---

---

## Function Quicksort

---

**Input:**  $V, i, j$

**Output:**  $V, \quad V[i] < V[i + 1], 0 \leq i < n - 2$

```
1 if(  $i < j$  )  
2    $p \leftarrow \text{PARTITION}(V, i, j)$   
3    $\text{QUICKSORT}(V, i, p - 1)$   
4    $\text{QUICKSORT}(V, p + 1, j)$ 
```

---



# Quicksort

---





# Quicksort





# Quicksort: Análise

---

## Análise

A relação de recorrência do Mergesort, no pior caso, corresponde à:

$$T(n) = T(n - 1) + O(n) \in \Theta(n^2)$$

Contanto, no caso médio, o Quicksort divide as partições de modo em que a primeira e a última partição tenham tamanhos similares, o que leva a uma relação de recorrência que se aproxima de:

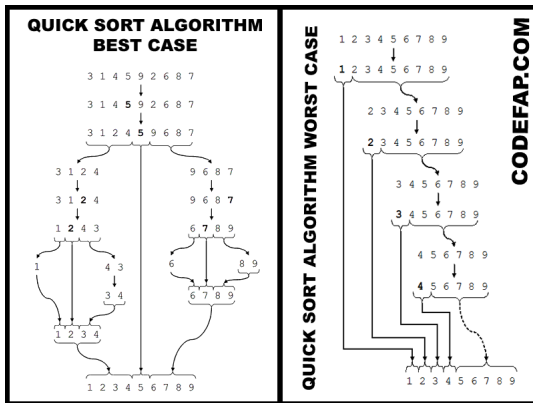
$$T(n) = 2 \cdot T(n/2) + O(n) \in \Theta(n \lg n)$$

In-place	Estável
X	X



# Quicksort

## Análise





# Sumário

---

## 2 Ordenação

- Bubblesort
- Insertionsort
- Mergesort
- Quicksort
- Heapsort



# Heapsort

---

## Heap

A chave do heapsort é uma estrutura denominada heap. Uma heap binária é uma estrutura de natureza recursiva e tem as seguinte propriedades:

- i) O elemento pai é  $\geq$  do que os seus filhos.
- ii) O filho da esquerda é uma heap.
- iii) O filho da direita também é uma heap.



# Heapsort

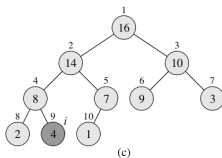
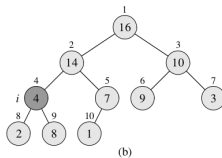
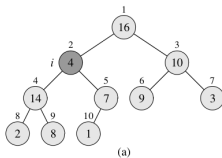
---



# Heapsort

## Heapify

Para construir uma Heap, devemos aplicar o procedimento de **heapify** nos nós que não apresentam a propriedade de Heap.

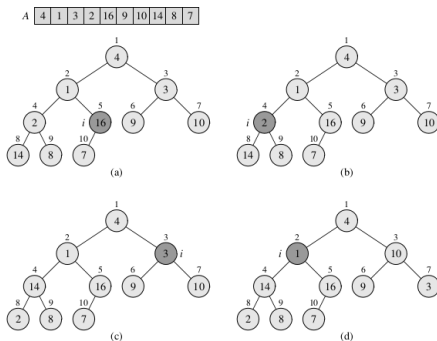




# Heapsort

## Heap

Note que os nós folha, já são heaps (por vacuidade). Logo, o **heapify** só necessita ser aplicado aos nós acima dos nós folhas.





# Heapsort

---

## Heapsort

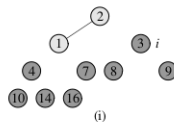
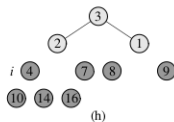
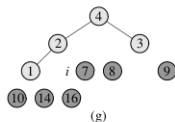
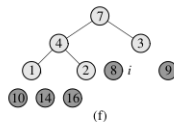
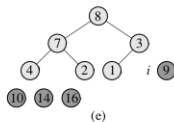
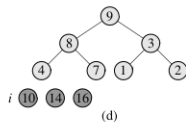
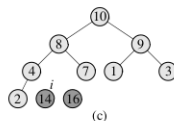
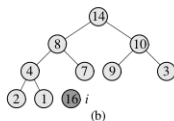
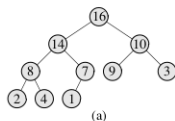
- Uma vez que a Heap está contruída, sabemos que o elemento raiz (primeiro elemento) é o maior de todos, logo podemos retirá-lo e colocá-lo no fim da sequência.
- Escolhemos o último nó folha para ser a raiz (primeiro elemento da sequência) e aplicamos **heapify** para manter a estrutura da heap.
- O procedimento é repetido até que tenhamos a sequência ordenada.





# Heapsort

## Exemplo





# Heapsort

---

---

## Function Heapsort

---

**Input:**  $V$

**Output:**  $V, \quad V[i] < V[i + 1], 0 \leq i < n - 1$

```
1 MAKEHEAP( $V$ )
2 for(  $i \leftarrow V.SIZE() - 1; i > 0; i --$  )
3   | SWAP( $V[0], V[i]$ )
4   | HEAPIFY( $V, 0, i$ )
```

---



# Heapsort

---

---

## Function MakeHeap

---

**Input:**  $V$

**Output:**  $V$ , com propriedade de **Heap**

```
1 for(  $i \leftarrow V.SIZE()/2; i \geq 0; i--$  )  
2   | HEAPIFY( $V, i, V.size()$ )
```

---



# Heapsort

---

## Function Heapify

---

**Input:**  $V, i, heapSize$

---

```
1  $l \leftarrow 2 \cdot i + 1$ 
2  $r \leftarrow 2 \cdot i + 2$ 
3  $largest \leftarrow i$ 
4 if(  $l < heapSize \wedge V[l] > V[i]$  )
5    $largest \leftarrow l$ 
6 if(  $r < heapSize \wedge V[r] > V[largest]$  )
7    $largest \leftarrow r$ 
8 if(  $largest \neq i$  )
9   SWAP( $V[i], V[largest]$ )
10  HEAPIFY( $V, largest, heapSize$ )
```

---



# Heapsort

---





# Heapsort

---

## Análise

- Para construir a Heap, leva-se tempo  $O(n \lg n)$ , uma vez que é necessário manter a propriedade de Heap para todos os nós, e cada nó tem altura  $O(\lg n)$ .
- Apesar de ser um limite superior, uma análise mais detalhada mostra que a construção da Heap é feita em tempo  $\Theta(n)$ .
- Uma vez que a Heap é construída, a retira do nó raiz e a manutenção da propriedade da Heap levam tempo  $\Theta(\lg n)$ .
- Como esse procedimento é repetido para todos os nós, temos que o Heapsort leva tempo  $\Theta(n \lg n)$ .



# Heapsort

---

In-place	Estável
✓	✗



# Heapsort

---

## Teorema

MAKEHEAP( $V$ ) leva tempo  $O(n)$ .





# Heapsort

## Demonstração

O procedimento `HEAPIFY()` quando chamado de um nó de altura  $h$  leva tempo  $O(h)$ .

Logo, `MAKEHEAP(V)` leva tempo:

$$\begin{aligned}
 \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) &\in \diamond \text{ cada nível de altura } h \text{ tem essa quantidade de folhas} \\
 O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) &\leq \diamond \text{ Isola o termo } n \\
 O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) &= \diamond \text{ Majoração} \\
 O\left(n \frac{1/2}{(1 - 1/2)^2}\right) &= \diamond \text{ Equivalência} \\
 O(2n) &\in O(n)
 \end{aligned}$$





# Sumário

---

## 3 Ordenação por Comparações



# Ordenação por Comparações

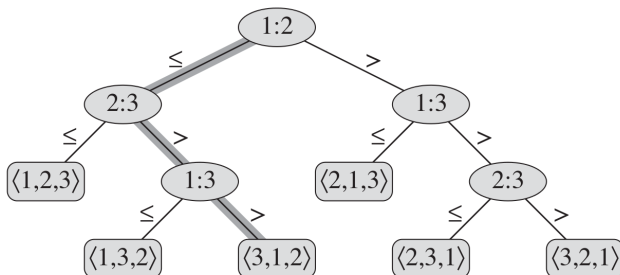
---

## Cota Inferior para Algoritmos de Ordenação por Comparações

- Vimos até o presente momento, diversos algoritmos de ordenação que se baseiam em comparações de chaves para resolver o problema da Ordenação.
- Até agora, sabemos que uma cota superior para o problema da ordenação é de  $O(n \lg n)$ , obtidas por algoritmos como o **Heapsort** e o **Mergesort**.
- Existe uma cota mínima para algoritmos de ordenação por comparação?



# Ordenação por Comparação





## Ordenação por Comparação

---

- Durante o seu trajeto, um algoritmo de ordenação faz comparações de modo a obter uma permutação da sequência original em ordem crescente.
- Isso corresponde de um percurso da raiz até uma folha.
- Quantas folhas temos? Qual a altura da árvore de decisões? Se o algoritmo deixar de explorar um caminho para todas as instâncias do problema o que acontece?



## Ordenação por Comparação

---

- Durante o seu trajeto, um algoritmo de ordenação faz comparações de modo a obter uma permutação da sequência original em ordem crescente.
- Isso corresponde de um percurso da raiz até uma folha.
- Quantas folhas temos? Qual a altura da árvore de decisões? Se o algoritmo deixar de explorar um caminho para todas as instâncias do problema o que acontece?

O problema de ordenações usando comparações possui cota  $\Omega(n \lg n)$ .



# Algoritmos Ótimos

---

## Algoritmos Ótimos

- O Heapsort e o Mergesort são **algoritmos ótimos** de ordenação por **comparações**.
- No pior caso, eles levam o mesmo tempo que o melhor algoritmo possível para o problema da ordenação por comparações, que por sua vez, possui uma cota inferior de  $\Omega(n \lg n)$ .



# Ordenação em Tempo “Linear”

---

## Ordenação em Tempo “Linear”

- É possível resolver o problema da ordenação em tempo linear ao explorar propriedades de algumas instâncias e resolver este problema reduzido em tempo  $o(n \lg n)$ , desde que não se use comparações.
- Dois métodos de ordenação em tempo “pseudolinear” são:
  - 1 Countingsort;
  - 2 Radixsort;





# Sumário

---

## 4 Ordenação em $O(n)$



# Sumário

---

## 4 Ordenação em $O(n)$

- Countingsort
- Radixsort



# Countingsort

---

## Countingsort

- O Countingsort conta as ocorrências de cada elemento na sequência original.
- Uma vez computada essa informação, o Countingsort calcula o número de elementos menor ou igual a um elemento  $i$  qualquer.
- A partir disso, o Countingsort consegue ordenar a sequência original.



# Countingsort

---

exampleblock



# Countingsort

---

## Function Countingsort

---

**Input:**  $V, i$

**Output:**  $V, \quad V[i] < V[i + 1], 0 \leq i < n - 2$

```
1  $V' \leftarrow V$ 
2 for(  $i \leftarrow 0; i \leq k; i++$  )
3    $C[i] \leftarrow 0$ 
4 for(  $i \leftarrow 0; i < V.SIZE(); i++$  )
5    $C[V'[i]]++$ 
6 for(  $i \leftarrow 1; i \leq k; i++$  )
7    $C[i] \leftarrow C[i] + C[i - 1]$ 
8 for(  $i \leftarrow V.SIZE() - 1; i \geq 0; i--$  )
9    $V[i - C[V'[i]]] \leftarrow V'[i]$ 
```

---



# Countingsort

---





# Countingsort: Análise

## Análise

- Primeiramente, é necessário contar a ocorrência de cada elemento, o que leva tempo  $\Theta(n)$ .
- Depois, é preciso computar a quantidade de elementos menores ou iguais a um outro determinado elemento, o que leva tempo  $\Theta(k)$ , onde  $k$  é o valor do maior elemento possível na sequência.
- Por fim, uma inspeção no vetor é necessária para executar a ordenação, logo, é necessário tempo  $\Theta(n)$ .
- Portanto, o custo total é de  $\Theta(n + k)$ .

In-place	Estável
✗	✓



# Sumário

---

## 4 Ordenação em $O(n)$

- Countingsort
- Radixsort





# Radixsort

---

## Radixsort

- A ideia do Radixsort é olhar, para cada iteração  $i$ , olhar para o  $i$ -ésimo dígito menos significativo e ordenar a sequência original baseado na ordem dos dígitos e na informação da iteração anterior.
- Ele pode usar o Countingsort para ordenar os dígitos na  $i$ -ésima iteração.
- Além disso, para ser um método estável, ele deve utilizar um método estável de ordenação para ordenar os dígitos em cada iteração.



# Radixsort

---

## exampleblock

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



# Radixsort

---

---

**Function** Radixsort

---

**Input:**  $V$

**Output:**  $V, \quad V[i] < V[i + 1], 0 \leq i < n - 1$

- 1 **for**(  $i \leftarrow 0; i < d; i++$  )
  - 2     Use um método de ordenação estável considerando apenas o  $i$ -ésimo dígito menos significativo
-



# Radixsort

---





# Radixsort

---

## Análise

- Suponha que o maior elemento possua  $d$  dígitos.
- Se em cada iteração utilizarmos o Countingsort para ordenar os dígitos, levaremos tempo  $O(n + k)$  para a iteração, onde  $k$  corresponde ao maior dígito.
- Como são necessárias  $d$  iterações, temos que o tempo total do Algoritmo corresponde à  $\Theta(d \cdot (n + k))$ .

In-place	Estável
✗	✓



# Sumário

---

## 5 Comparações



## Comparações

---

Método	Complexidade	In-place	Estável
Bubblesort	$\Theta(n^2)$	✓	✓
Insertionsort	$\Theta(n^2)$	✓	✓
Mergesort	$\Theta(n \lg n)$	✗	✓
Quicksort	$\Theta(n^2)$	✗	✗
Heapsort	$\Theta(n \lg n)$	✓	✗
Countingsort	$\Theta(n + k)$	✗	✓
Radixsort	$\Theta(d \cdot (n + k))$	✗	✓



# Sumário

---

## 6 Links





## Links

---

- Sonorização de algoritmos de ordenação:  
<http://youtu.be/t8g-iYGHpEA>
- Sonorização de algoritmos de ordenação 2:  
<http://youtu.be/kPRA0W1kECg>
- Grupo AlgoRythmics de dança:  
<http://www.youtube.com/user/AlgoRythmics>
- Animações de algoritmos de ordenação:  
<http://www.sorting-algorithms.com/>