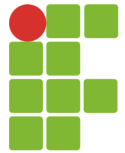
Métodos de Ordenação Pseudolineares

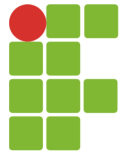
Análise de Algoritmos – Ciência da Computação

Prof. Daniel Saad Nogueira Nunes

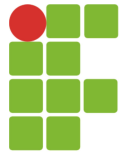
IFB – Instituto Federal de Brasília,
Campus Taguatinga



Sumário



Sumário

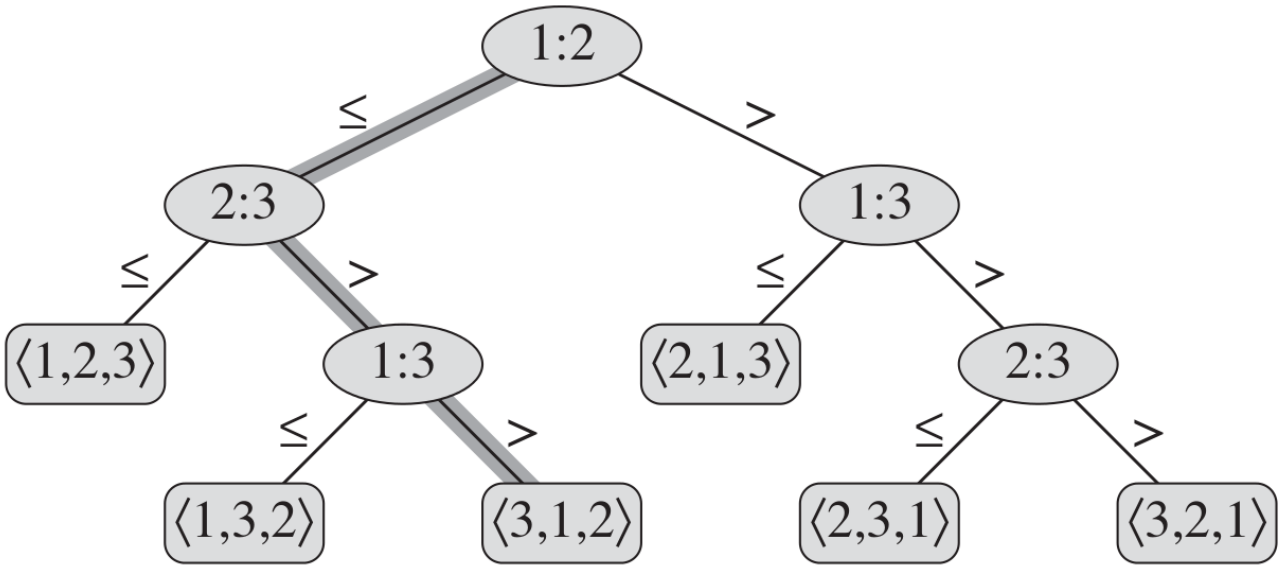


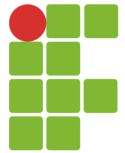
Ordenação por Comparações

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 prolema 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?

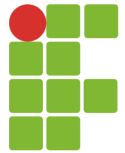
Ordenação por Comparação





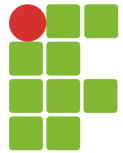
Ordenação por Comparação

- Durante o seu trajeto, o algoritmo de ordenação, faz escolhas baseadas nessa árvore de decisão de modo a obter uma permutação da sequência original em ordem crescente.
- Isso corresponde de um percurso da raiz até uma folha.
- Qual a cota inferior para o número de folhas? E para altura da árvore?



Ordenação por Comparação

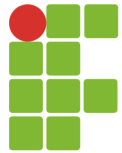
- Durante o seu trajeto, o algoritmo de ordenação, faz escolhas baseadas nessa árvore de decisão de modo a obter uma permutação da sequência original em ordem crescente.
- Isso corresponde de um percurso da raiz até uma folha.
- Qual a cota inferior para o número de folhas? E para altura da árvore?
- $n_{\text{folhas}} \geq n!$ folhas e $h \geq \lg n!$.



Ordenação por Comparação

- Durante o seu trajeto, o algoritmo de ordenação, faz escolhas baseadas nessa árvore de decisão de modo a obter uma permutação da sequência original em ordem crescente.
- Isso corresponde de um percurso da raiz até uma folha.
- Qual a cota inferior para o número de folhas? E para altura da árvore?
- $n_{\text{folhas}} \geq n!$ folhas e $h \geq \lg n!$.

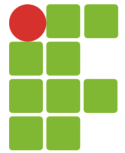
O problema de ordenações usando comparações possui cota $\Omega(n \lg n)$.



Demonstração da cota inferior

Queremos mostrar que $\lg n! \in \Omega(n \lg n)$

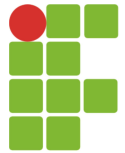
$$
\begin{aligned}
\lg n! &= \sum_{i=1}^{n} \lg(i) \\
&\geq \sum_{i=\lfloor \frac{n}{2} \rfloor + 1}^{n} \lg(i) \\
&\geq \sum_{i=\lfloor \frac{n}{2} \rfloor + 1}^{n} \lg\left(\frac{n}{2}\right) \\
&\geq \frac{n}{2} \lg(n/2) \\
&= \frac{n}{2}(\lg(n) - 1) = \frac{n \lg n}{2} - \frac{n}{2} \geq cn \lg n, \quad \diamond\ c < 2 \text{ e } \forall n > n_0 \text{ sfg}
\end{aligned}
$$



Algoritmos Ótimos

Algoritmos Ótimos

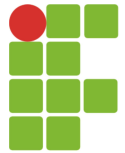
- O Heapsort e o Mergesort são **algoritmos ótimos** de ordenação por **comparações**.
- No pior caso, eles levam o mesmo tempo que o melhor algoritmo possível para o problema da ordenação por comparações, que por sua vez, possui uma cota inferior de $\Omega(n \lg n)$.



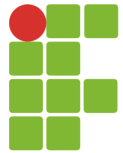
Ordenação em Tempo "Linear"

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;



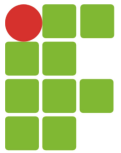
Sumário



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 ao varrer a sequência da direita para esquerda e, utilizando os contadores de cada valor, posicioná-los na posição correta da sequência resultado.

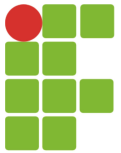


Countingsort

Exemplo

	0	1	2	3	4	5	6	7	8
V	3	6	4	1	3	4	1	0	4

	0	1	2	3	4	5	6
C	0	0	0	0	0	0	0

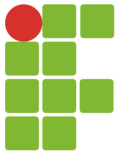


Countingsort

Exemplo

	0	1	2	3	4	5	6	7	8
V	3	6	4	1	3	4	1	0	4

	0	1	2	3	4	5	6
C	1	2	0	2	3	0	1

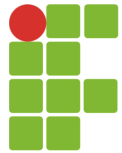


Countingsort

Exemplo

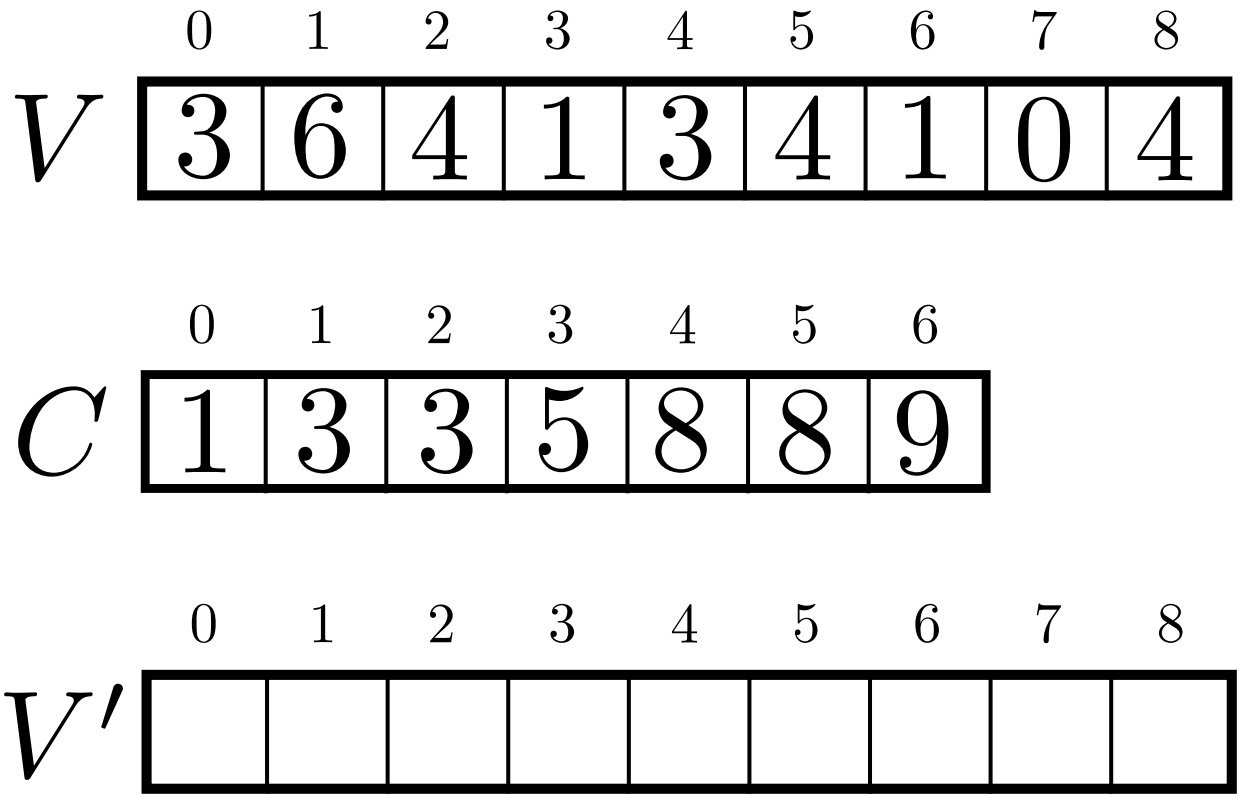
	0	1	2	3	4	5	6	7	8
V	3	6	4	1	3	4	1	0	4

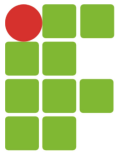
	0	1	2	3	4	5	6
C	1	3	3	5	8	8	9



Countingsort

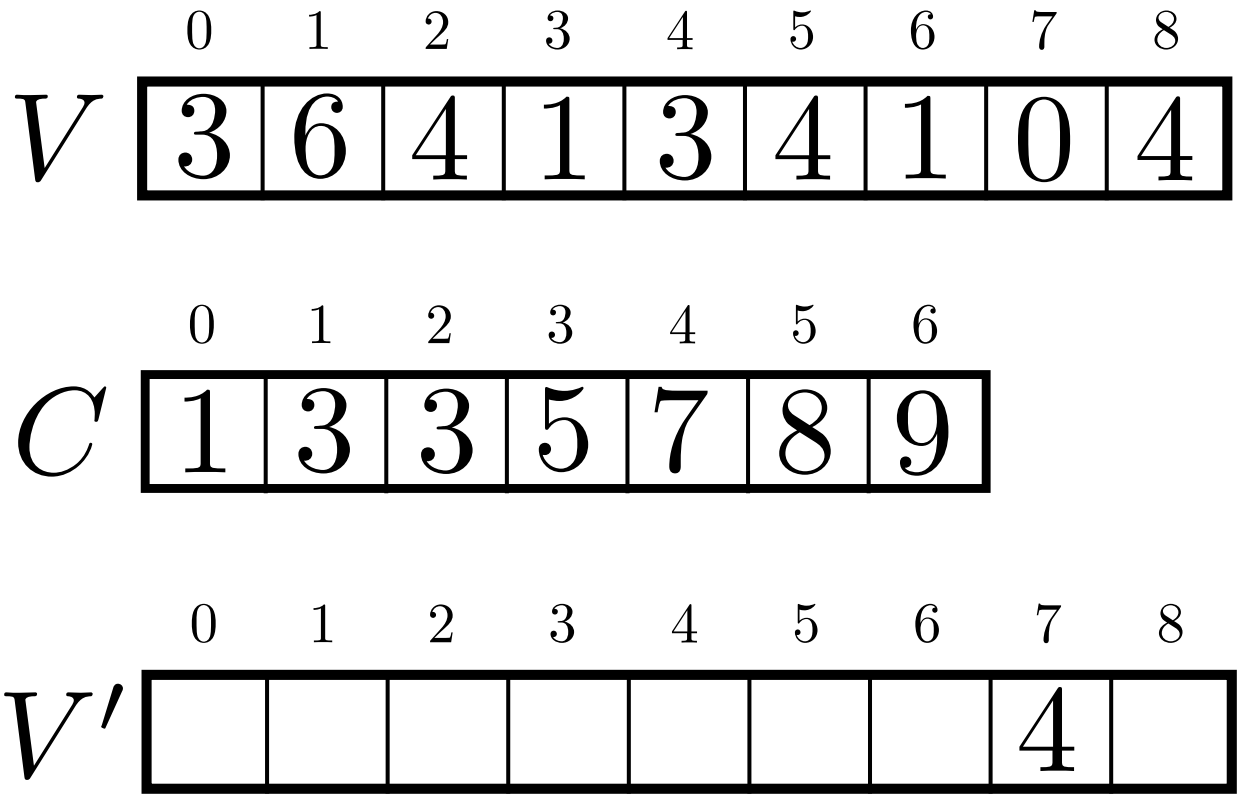
Exemplo

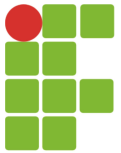




Countingsort

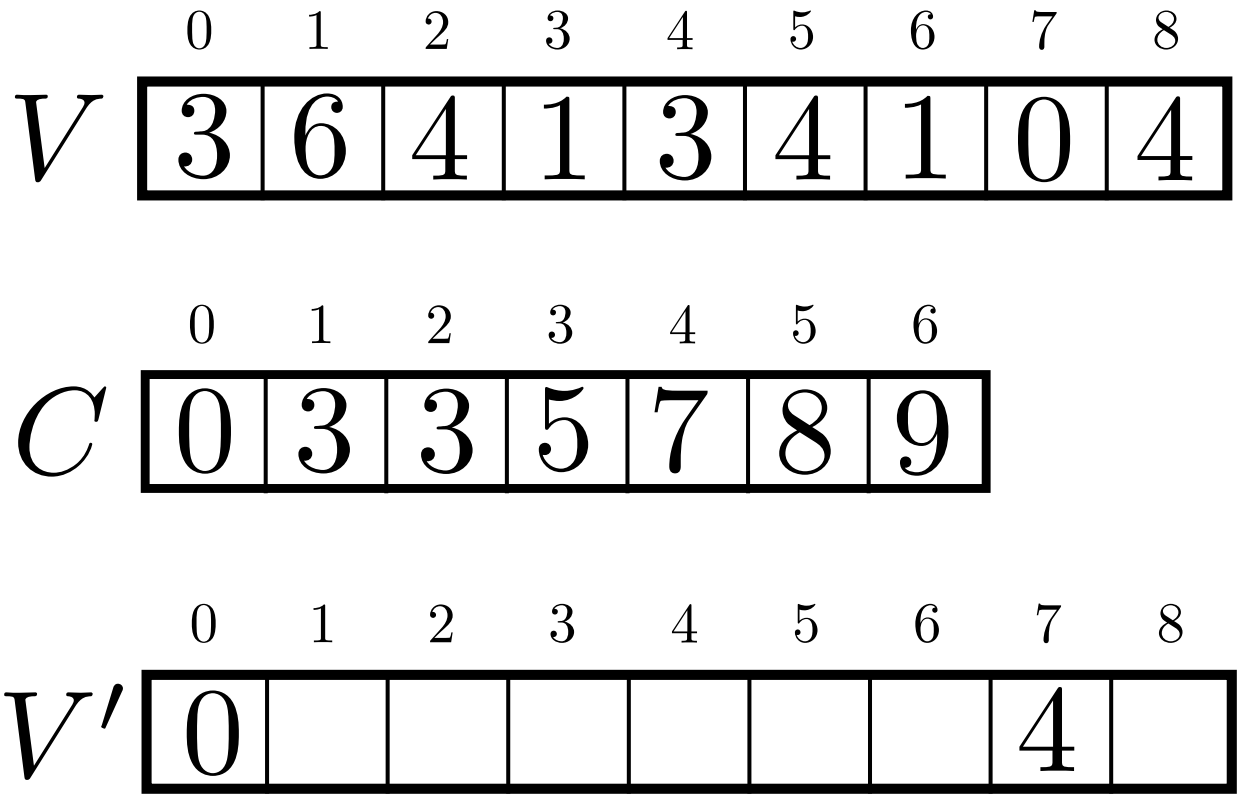
Exemplo





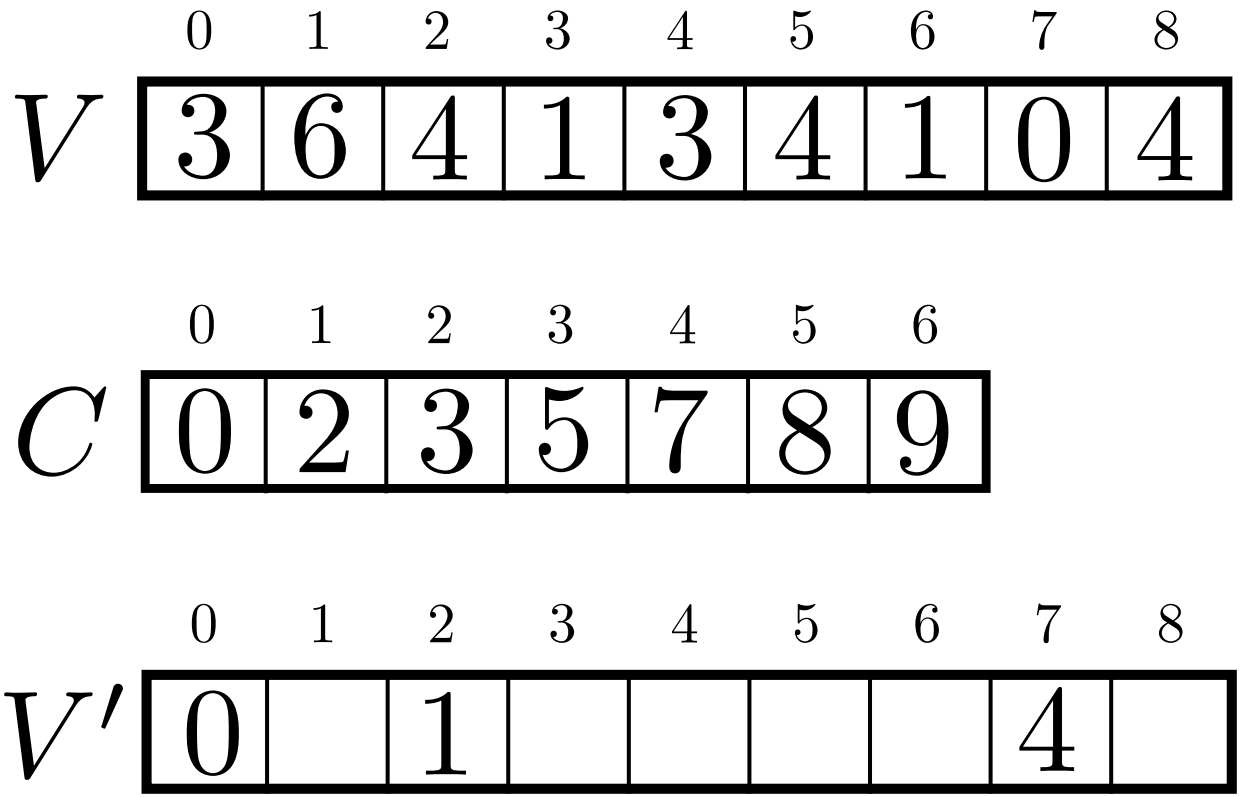
Countingsort

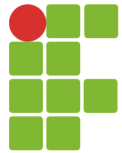
Exemplo



Countingsort

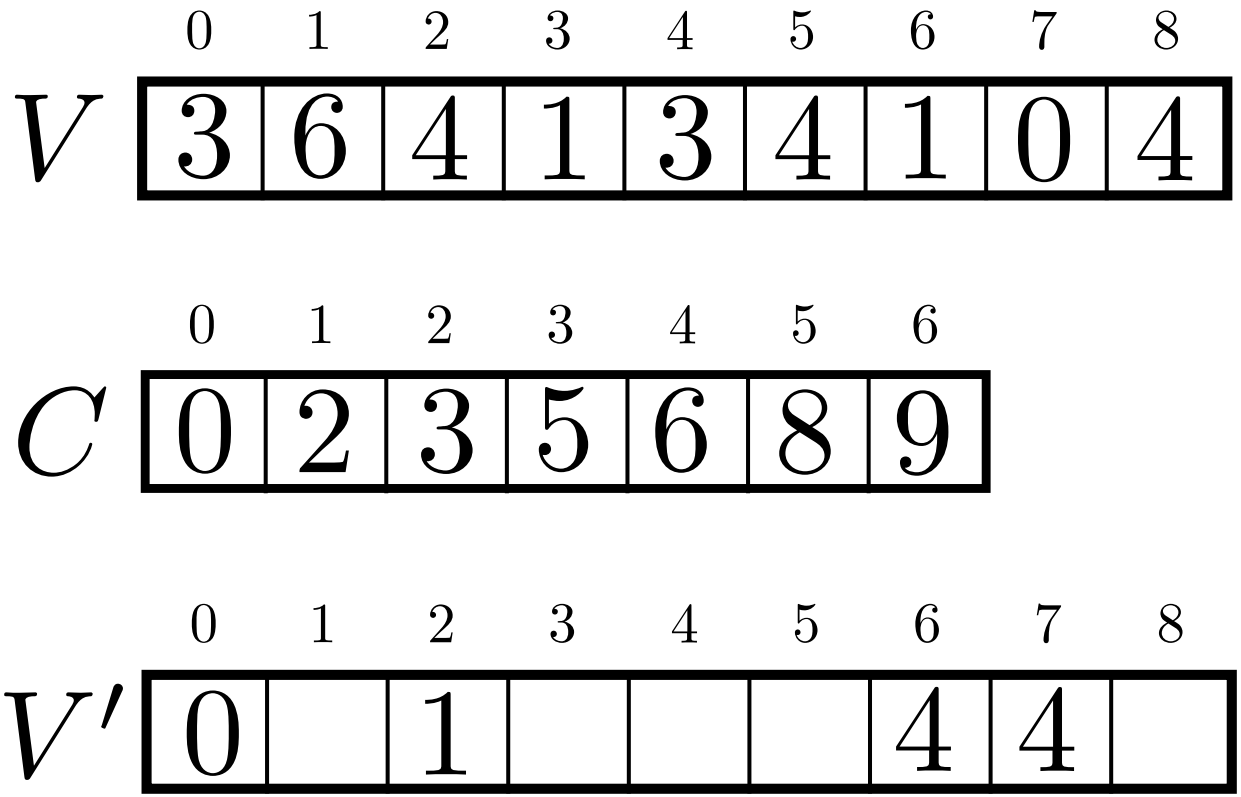
Exemplo





Countingsort

Exemplo



Countingsort

Exemplo

	0	1	2	3	4	5	6	7	8
V	3	6	4	1	3	4	1	0	4

	0	1	2	3	4	5	6
C	0	2	3	4	6	8	9

	0	1	2	3	4	5	6	7	8
V'	0		1		3		4	4	

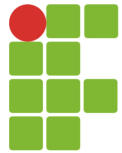
Countingsort

Exemplo

	0	1	2	3	4	5	6	7	8
V	3	6	4	1	3	4	1	0	4

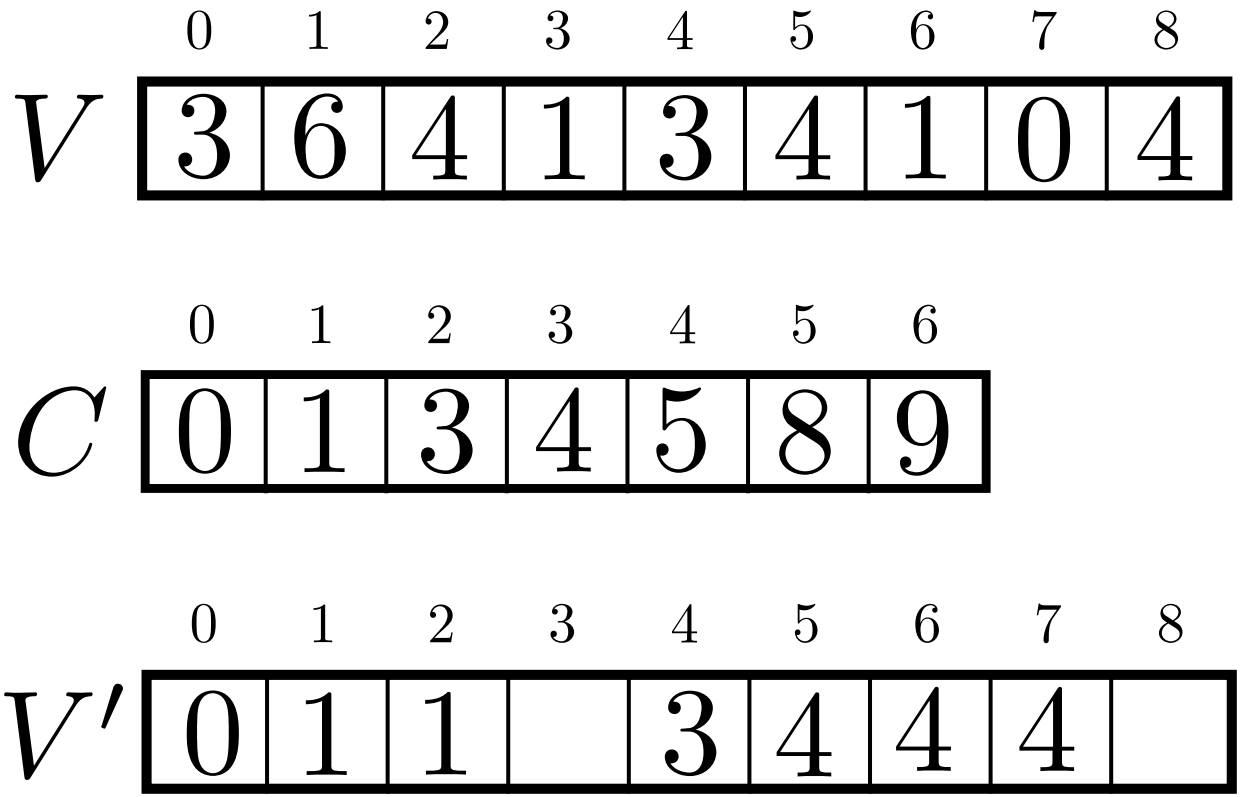
	0	1	2	3	4	5	6
C	0	1	3	4	6	8	9

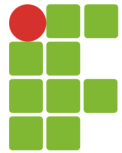
	0	1	2	3	4	5	6	7	8
V'	0	1	1		3		4	4	



Countingsort

Exemplo





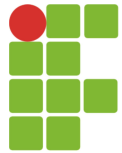
Countingsort

Exemplo

	0	1	2	3	4	5	6	7	8
V	3	6	4	1	3	4	1	0	4

	0	1	2	3	4	5	6
C	0	1	3	4	5	8	8

	0	1	2	3	4	5	6	7	8
V'	0	1	1		3	4	4	4	6



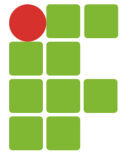
Countingsort

Exemplo

	0	1	2	3	4	5	6	7	8
V	3	6	4	1	3	4	1	0	4

	0	1	2	3	4	5	6
C	0	1	3	3	5	8	8

	0	1	2	3	4	5	6	7	8
V'	0	1	1	3	3	4	4	4	6



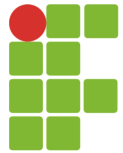
Countingsort

Function Countingsort

Input: $V[0, n-1], k$

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

```
1 V' ← V[0, n − 1]
2 for( i ← 0; i ≤ k; i++ )
3     C[i] ← 0
4 for( i ← 0; i < n; i++ )
5     C[V'[i]]++
6 for( i ← 1; i ≤ k; i++ )
7     C[i] ← C[i] + C[i − 1]
8 for( i ← n − 1; i ≥ 0; i−− )
9     V[−−C[V'[i]] ← V'[i]
```

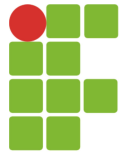


Countingsort: Análise

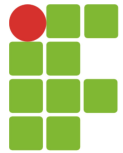
Análise

- 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
✗	✓



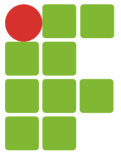
Sumário



Radixsort

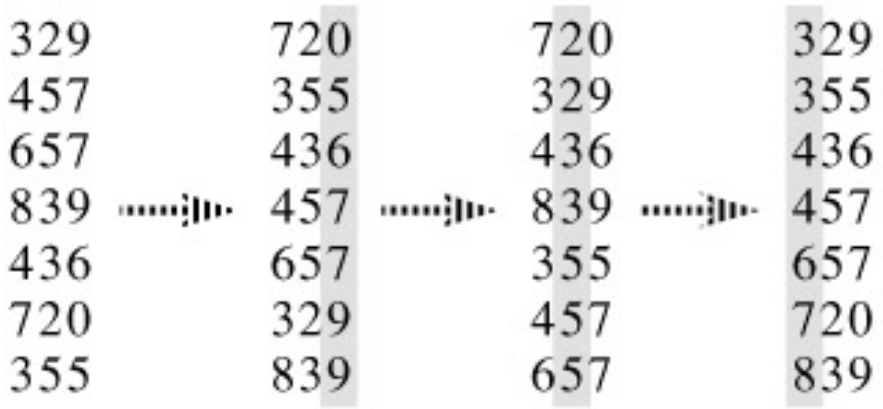
Radixsort

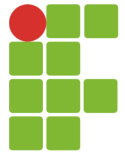
- A ideia do Radixsort é olhar, para cada iteração i, olhar para o i-ésimo dígito menos significativo e ordenar a sequência original baseado na ordem dos dígitos e na informação da iteração anterior.
- Ele pode usar o Countingsort para ordenar os dígitos na i-ésima iteração.
- Ele deve utilizar um método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



Radixsort

Exemplo





Radixsort

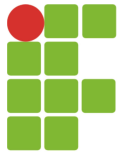
Function Radixsort

Input: $V[0, n-1]$

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

Análise

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

In-place	Estável
✗	✓