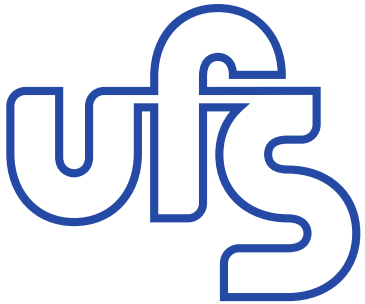
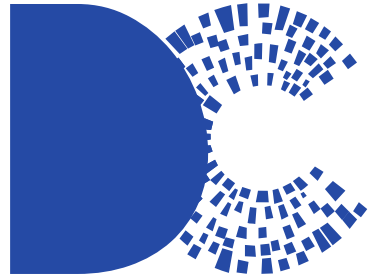


UNIVERSIDADE FEDERAL DE SERGIPE



DEPARTAMENTO DE COMPUTAÇÃO

Ordenação linear

Projeto e Análise de Algoritmos

Bruno Prado

Departamento de Computação / UFS

Introdução

- Algoritmos clássicos de ordenação
  - Mergesort
    - $\Omega(n \log_2 n)$
    - $O(n \log_2 n)$

Introdução

- Algoritmos clássicos de ordenação
  - Mergesort
    - $\Omega(n \log_2 n)$
    - $O(n \log_2 n)$
  - Quicksort
    - $\Omega(n \log_2 n)$
    - $O(n^2)$

Introdução

- Algoritmos clássicos de ordenação
  - Mergesort
    - $\Omega(n \log_2 n)$
    - $O(n \log_2 n)$
  - Quicksort
    - $\Omega(n \log_2 n)$
    - $O(n^2)$
  - Heapsort
    - $\Omega(n \log_2 n)$
    - $O(n \log_2 n)$

Introdução

- Algoritmos clássicos de ordenação
  - Mergesort
    - $\Omega(n \log_2 n)$
    - $O(n \log_2 n)$
  - Quicksort
    - $\Omega(n \log_2 n)$
    - $O(n^2)$
  - Heapsort
    - $\Omega(n \log_2 n)$
    - $O(n \log_2 n)$

Todos executam em pelo menos $\Omega(n \log_2 n)$ passos

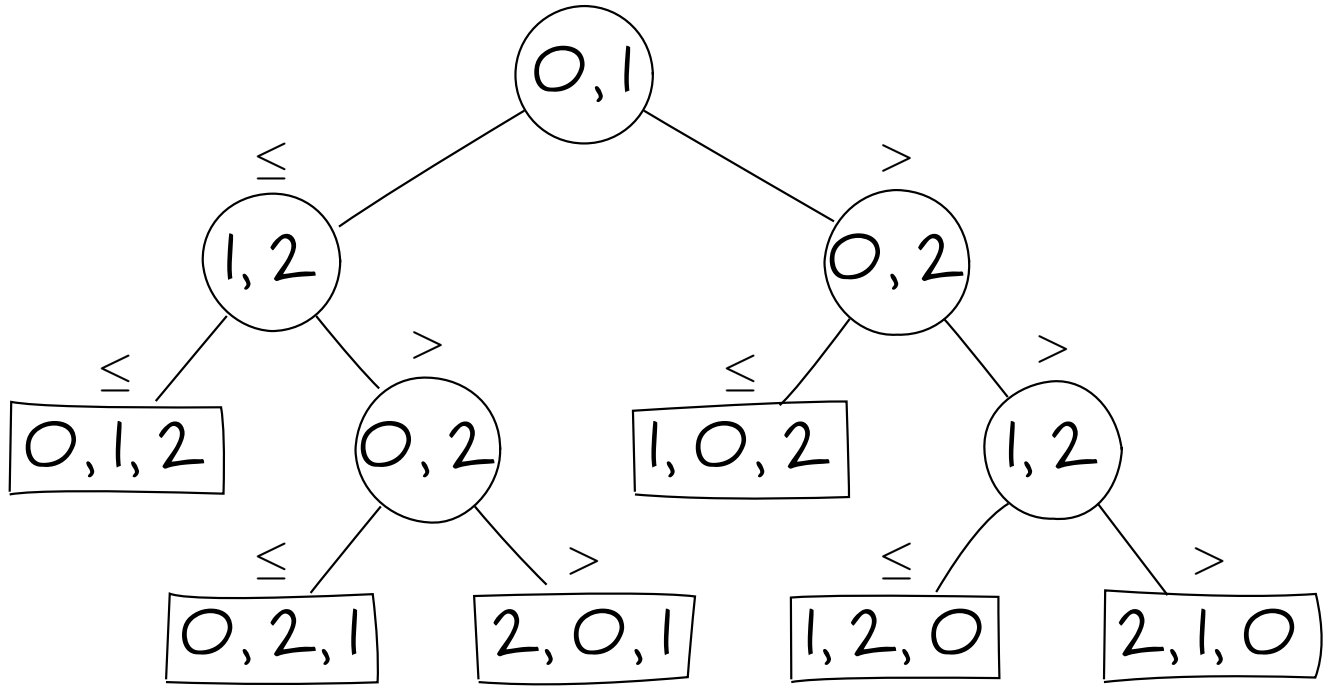
Introdução

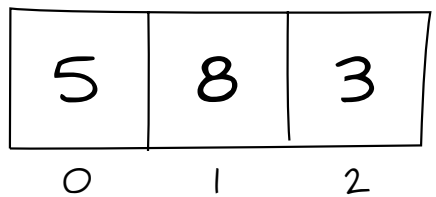
- Algoritmos clássicos de ordenação
  - Mergesort
    - $\Omega(n \log_2 n)$
    - $O(n \log_2 n)$
  - Quicksort
    - $\Omega(n \log_2 n)$
    - $O(n^2)$
  - Heapsort
    - $\Omega(n \log_2 n)$
    - $O(n \log_2 n)$

Este é o limite inferior de ordenação?

Introdução

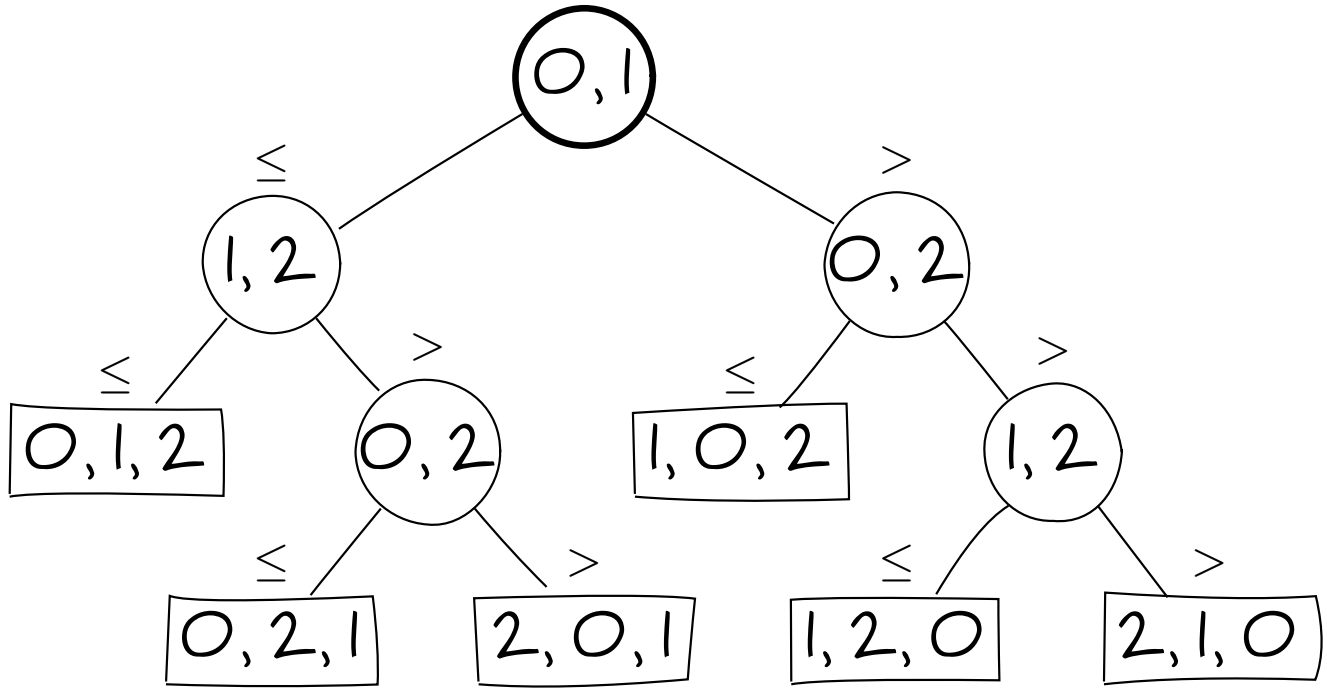
- Ordenação baseada em comparação
  - Árvore de decisão

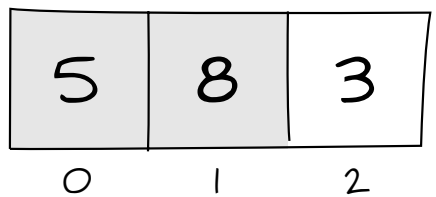




Introdução

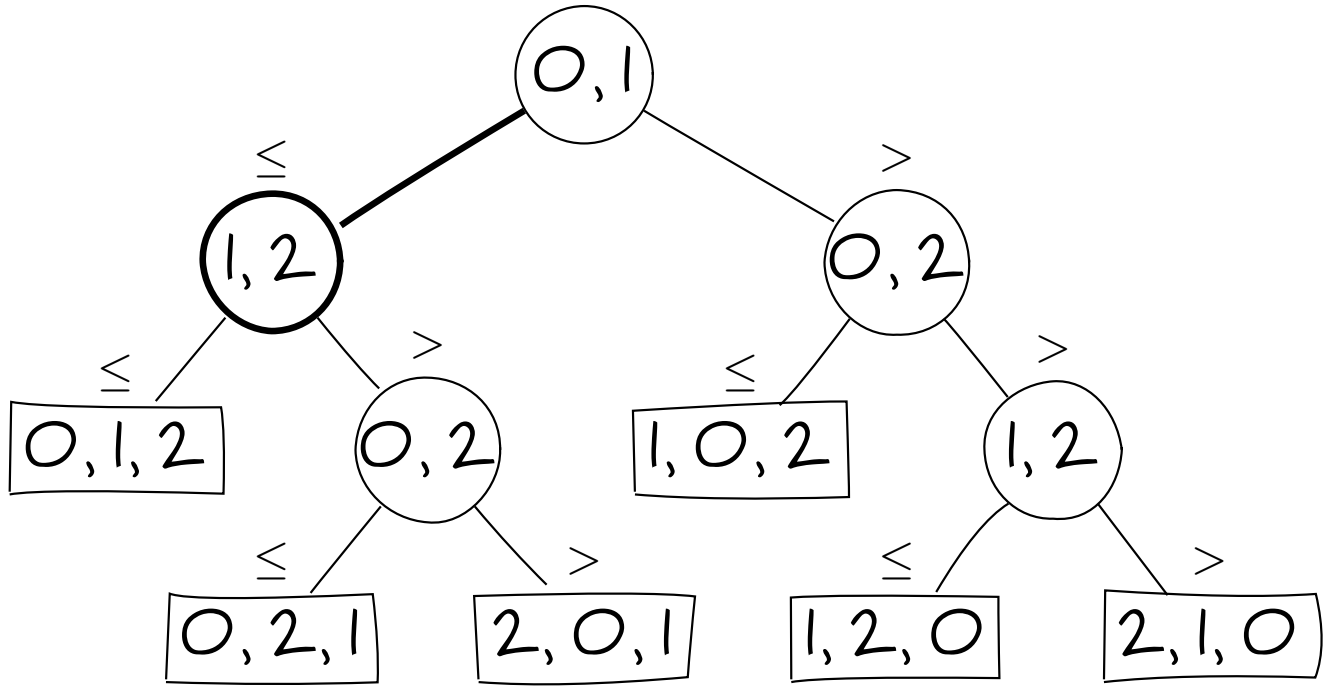
- Ordenação baseada em comparação
  - Árvore de decisão

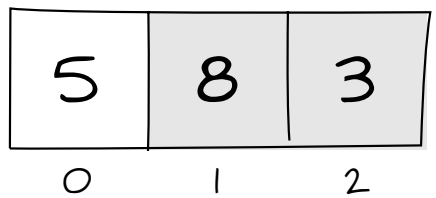




Introdução

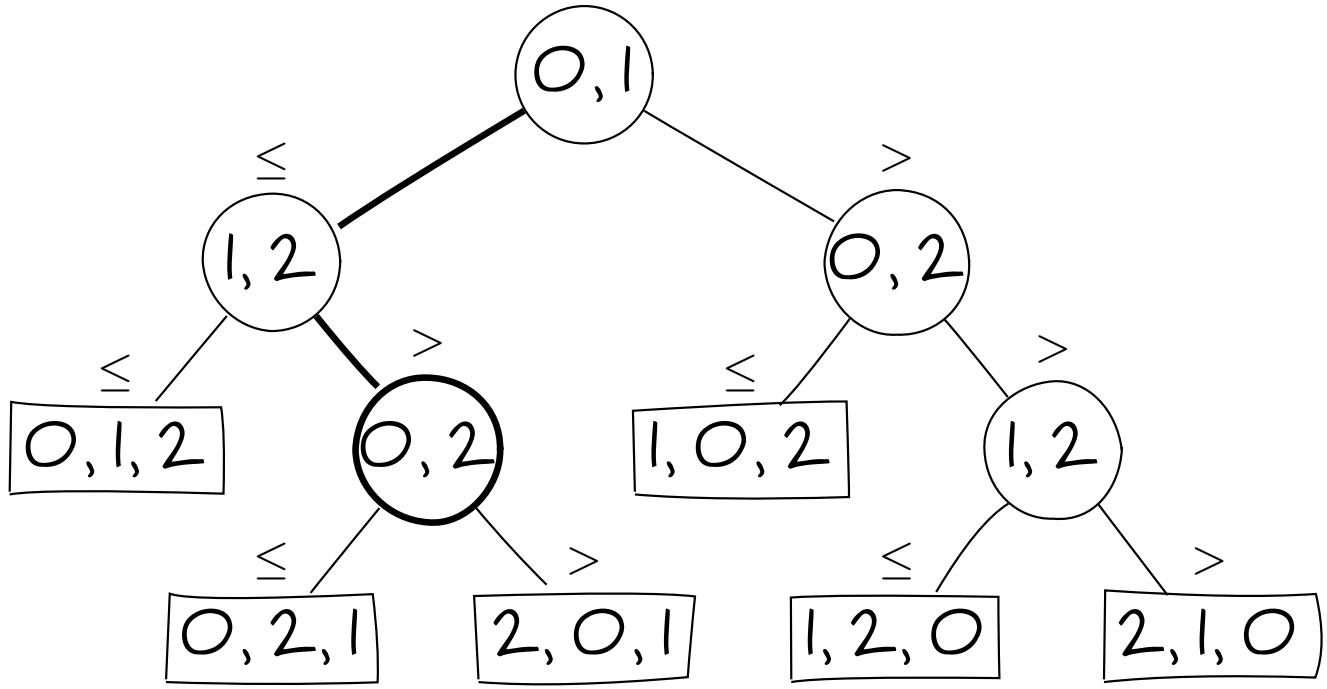
- Ordenação baseada em comparação
  - Árvore de decisão

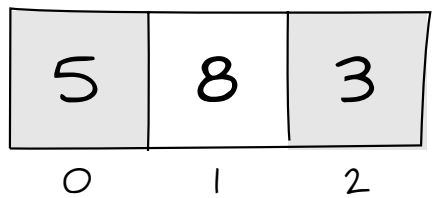




Introdução

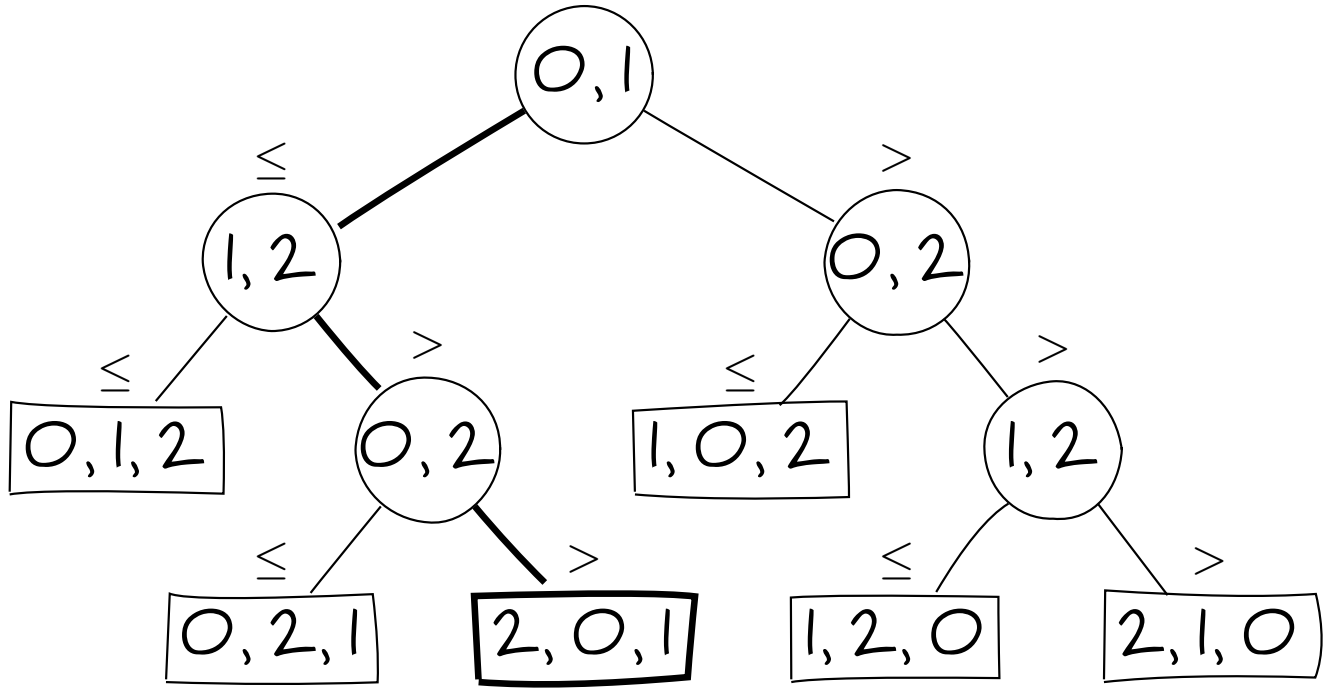
- Ordenação baseada em comparação
  - Árvore de decisão

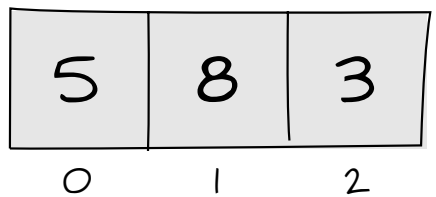




Introdução

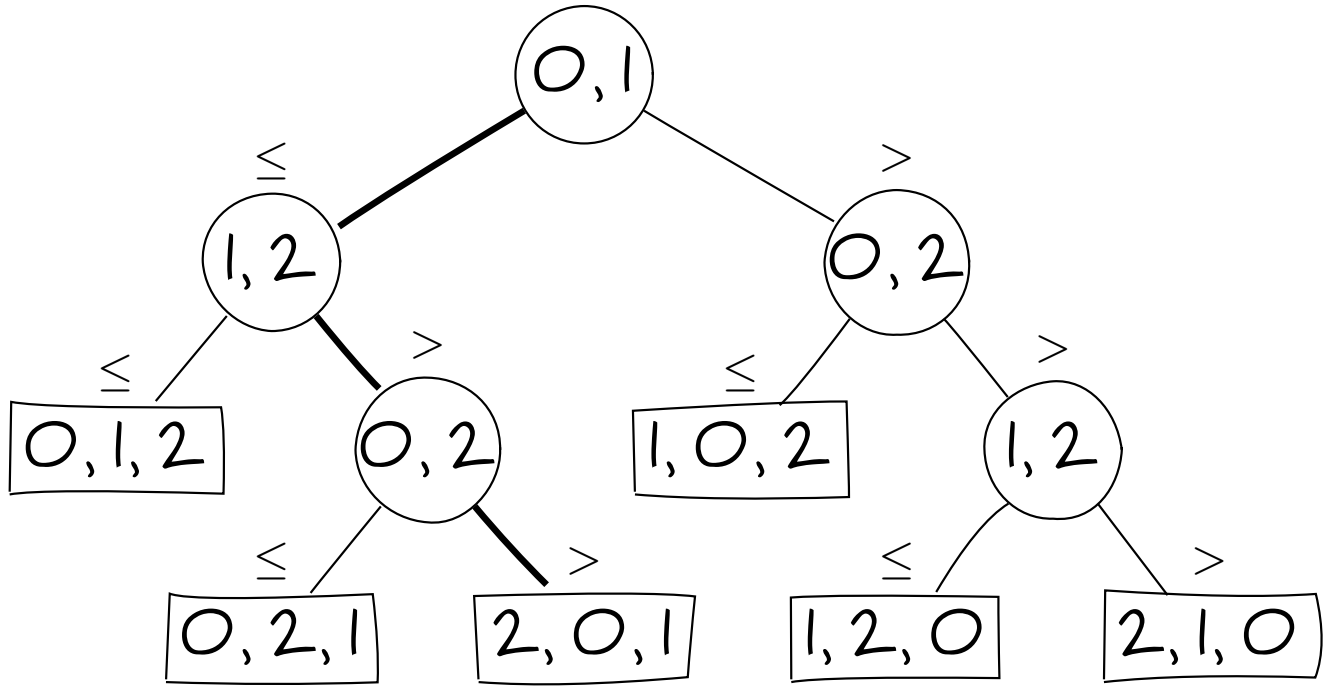
- Ordenação baseada em comparação
  - Árvore de decisão

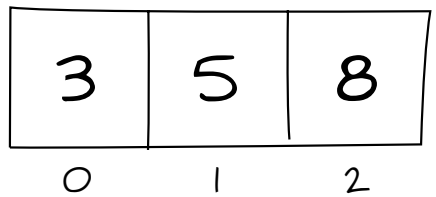




Introdução

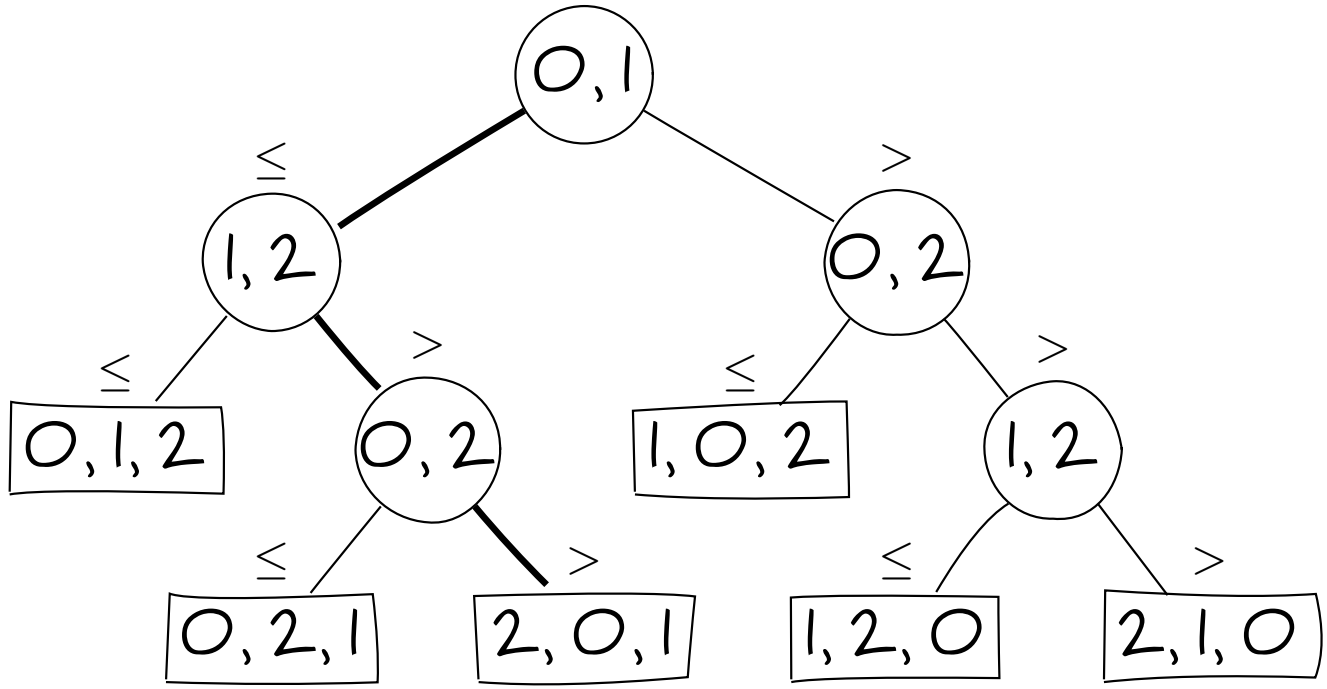
- Ordenação baseada em comparação
  - Árvore de decisão

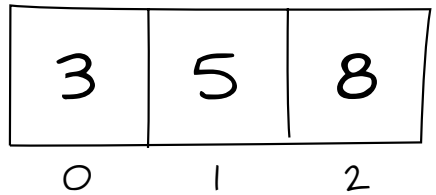




Introdução

- Ordenação baseada em comparação
  - Árvore de decisão

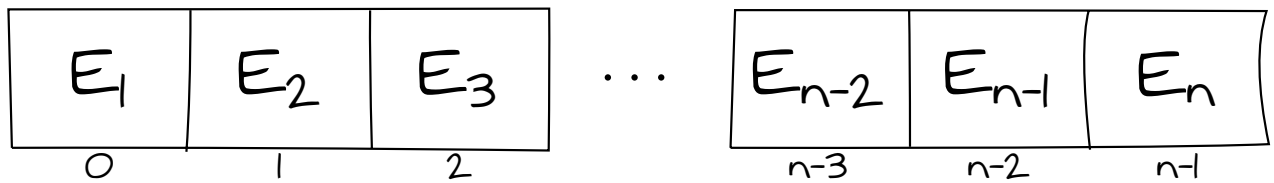




A quantidade de comparações é a altura da árvore

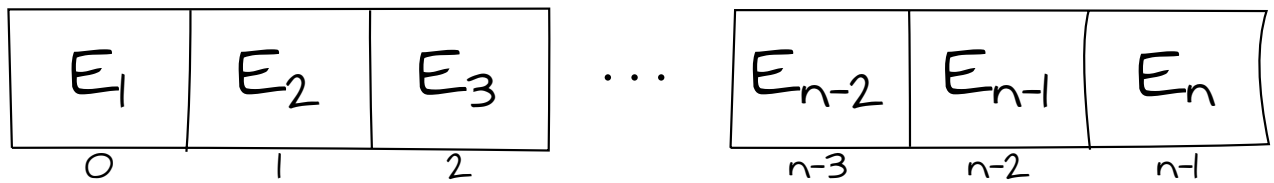
Introdução

- Ordenação baseada em comparação
  - Todos os n elementos precisam ser comparados por operações relacionais binárias ($\leq$, $<$, $\geq$ ou $>$)



Introdução

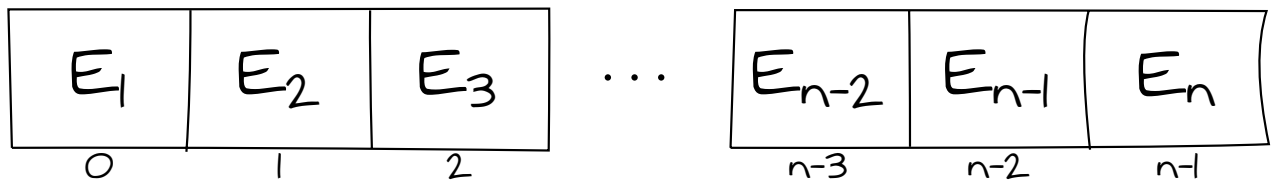
- Ordenação baseada em comparação
  - Todos os n elementos precisam ser comparados por operações relacionais binárias ($\leq$, $<$, $\geq$ ou $>$)



  - O número mínimo de comparações é descrito por $h = f(n)$, que é a altura da árvore

Introdução

- Ordenação baseada em comparação
  - Todos os n elementos precisam ser comparados por operações relacionais binárias ($\leq$, $<$, $\geq$ ou $>$)



  - O número mínimo de comparações é descrito por $h = f(n)$, que é a altura da árvore
  - Nas folhas da árvore existem $n!$ combinações possíveis para ordenação da sequência

Introdução

- Ordenação baseada em comparação
  - Como uma árvore binária com altura $h = f(n)$ possui no máximo $2^{f(n)}$ folhas, temos que $2^{f(n)} \geq n!$

Introdução

- Ordenação baseada em comparação
  - Como uma árvore binária com altura $h = f(n)$ possui no máximo $2^{f(n)}$ folhas, temos que $2^{f(n)} \geq n!$
  - Aplicando a operação de logaritmo em base 2 em ambos os lados da expressão

$$\begin{aligned} \log_2 2^{f(n)} &\geq \log_2 n! \\ f(n) &\geq \log_2 n! \end{aligned}$$

Introdução

- Ordenação baseada em comparação
  - Como uma árvore binária com altura $h = f(n)$ possui no máximo $2^{f(n)}$ folhas, temos que $2^{f(n)} \geq n!$
  - Aplicando a operação de logaritmo em base 2 em ambos os lados da expressão

$$\begin{aligned} \log_2 2^{f(n)} &\geq \log_2 n! \\ f(n) &\geq \log_2 n! \end{aligned}$$

  - Utilizando a aproximação de Stirling

$$\begin{aligned} O(\log_2 n!) &= n \log_2 n - n + O(\log_2 n) \\ O(\log_2 n!) &= O(n \log_2 n) \end{aligned}$$

Introdução

- Ordenação baseada em comparação
  - Como uma árvore binária com altura $h = f(n)$ possui no máximo $2^{f(n)}$ folhas, temos que $2^{f(n)} \geq n!$
  - Aplicando a operação de logaritmo em base 2 em ambos os lados da expressão

$$\begin{aligned} \log_2 2^{f(n)} &\geq \log_2 n! \\ f(n) &\geq \log_2 n! \end{aligned}$$

  - Utilizando a aproximação de Stirling

$$\begin{aligned} O(\log_2 n!) &= n \log_2 n - n + O(\log_2 n) \\ O(\log_2 n!) &= O(n \log_2 n) \\ &\downarrow \\ f(n) &\geq n \log_2 n \end{aligned}$$

Introdução

- Ordenação baseada em comparação
  - Como uma árvore binária com altura $h = f(n)$ possui no máximo $2^{f(n)}$ folhas, temos que $2^{f(n)} \geq n!$
  - Aplicando a operação de logaritmo em base 2 em ambos os lados da expressão

$$\begin{aligned} \log_2 2^{f(n)} &\geq \log_2 n! \\ f(n) &\geq \log_2 n! \end{aligned}$$

  - Utilizando a aproximação de Stirling

$$\begin{aligned} O(\log_2 n!) &= n \log_2 n - n + O(\log_2 n) \\ O(\log_2 n!) &= O(n \log_2 n) \\ &\downarrow \\ f(n) &\geq n \log_2 n \\ &= \Omega(n \log_2 n) \end{aligned}$$

Ordenação linear

- Conjunto de entrada com tamanho n

			↓				
n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10^1	$3,3$	10^1	$3,3 \times 10^1$	10^2	10^3	10^3	$3,6 \times 10^6$
10^2	$6,6$	10^2	$6,6 \times 10^2$	10^4	10^6	$1,3 \times 10^{30}$	$9,3 \times 10^{157}$
10^3	10	10^3	$1,0 \times 10^4$	10^6	10^9	-	-
10^4	13	10^4	$1,3 \times 10^5$	10^8	10^{12}	-	-
10^5	17	10^5	$1,7 \times 10^6$	10^{10}	10^{15}	-	-
10^6	20	10^6	$2,0 \times 10^7$	10^{12}	10^{18}	-	-

Ordenação linear

- Conjunto de entrada com tamanho n

n	$\log_2 n$	n	$n \log_2 n$ ↓	n^2	n^3	2^n	$n!$
10^1	$3,3$	10^1	$3,3 \times 10^1$	10^2	10^3	10^3	$3,6 \times 10^6$
10^2	$6,6$	10^2	$6,6 \times 10^2$	10^4	10^6	$1,3 \times 10^{30}$	$9,3 \times 10^{157}$
10^3	10	10^3	$1,0 \times 10^4$	10^6	10^9	-	-
10^4	13	10^4	$1,3 \times 10^5$	10^8	10^{12}	-	-
10^5	17	10^5	$1,7 \times 10^6$	10^{10}	10^{15}	-	-
10^6	20	10^6	$2,0 \times 10^7$	10^{12}	10^{18}	-	-

É possível ordenar com um número inferior de passos?

Ordenação linear

- É possível ordenar em tempo linear $O(n)$, desde que o paradigma de ordenação utilizado não seja baseado em comparações dos números
  - Counting Sort
  - Radix Sort
  - Bucket Sort

Ordenação linear

- É possível ordenar em tempo linear $O(n)$, desde que o paradigma de ordenação utilizado não seja baseado em comparações dos números
  - Counting Sort
  - Radix Sort
  - Bucket Sort
- A ordenação é feita pela contagem e indexação dos elementos, entretanto, esta classe de algoritmos precisa de um alfabeto reduzido para entrada, o que gera limitações para suas aplicações

Counting sort

- Etapas de execução do algoritmo
  1. Cálculo do histograma do vetor de entrada

Counting sort

- Etapas de execução do algoritmo
  1. Cálculo do histograma do vetor de entrada
  2. Determinação dos índices de cada símbolo

Counting sort

- Etapas de execução do algoritmo
  1. Cálculo do histograma do vetor de entrada
  2. Determinação dos índices de cada símbolo
  3. Ordenação do vetor de saída

Counting sort

- Etapas de execução do algoritmo
  1. Cálculo do histograma do vetor de entrada
  2. Determinação dos índices de cada símbolo
  3. Ordenação do vetor de saída
- Limitação: o alfabeto (número de símbolos) não pode ser grande, porque é necessário um vetor auxiliar para indexação do tamanho deste alfabeto

Counting sort

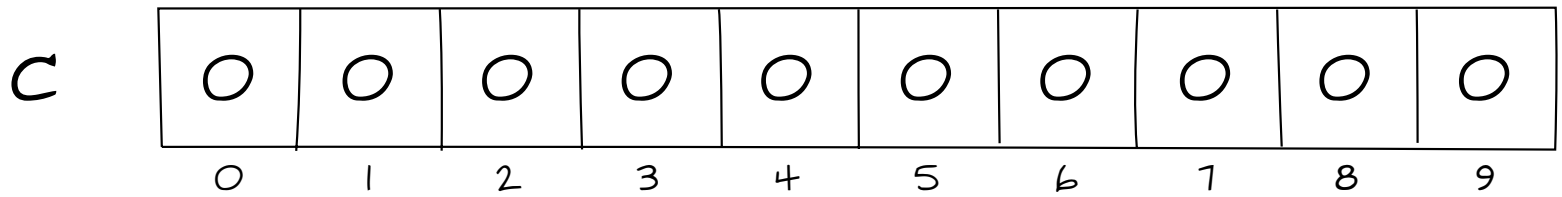
```
1  // Padrão de tipos por tamanho
2  #include <stdint.h>
3  // Procedimento counting sort
4  void counting_sort(int32_t A[], int32_t B[], uint32_t
       n, uint32_t k) {
5      // Vetor de contagem e indexação
6      uint32_t C[k] = { 0 };
7      // Histograma
8      for(uint32_t i = 0; i < n; i++)
9          C[A[i]] = C[A[i]] + 1;
10     // Indexação
11     for(i = 1; i < k; i++)
12         C[i] = C[i] + C[i - 1];
13     // Ordenação
14     for(i = n - 1; i >= 0; i--) {
15         B[C[A[i]] - 1] = A[i];
16         C[A[i]] = C[A[i]] - 1;
17     }
18 }
```

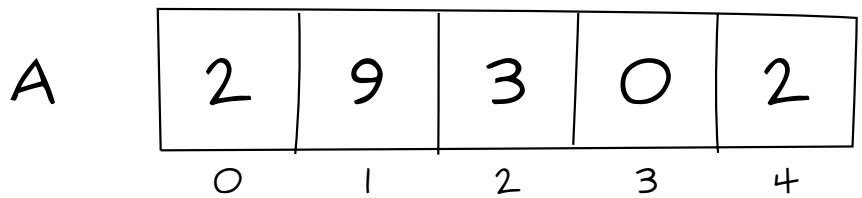
Counting sort

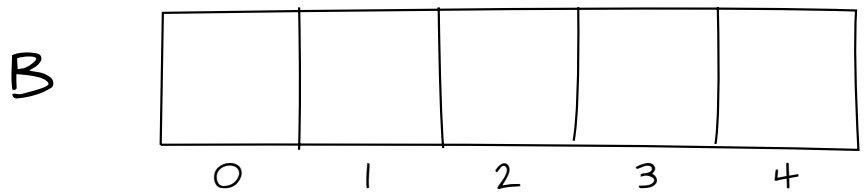
```
1  // Padrão de tipos por tamanho
2  #include <stdint.h>
3  // Procedimento counting sort
4  void counting_sort(int32_t A[], int32_t B[], uint32_t
       n, uint32_t k) {
5      // Vetor de contagem e indexação
6      uint32_t C[k] = { 0 };
7      // Histograma
8      for(uint32_t i = 0; i < n; i++)
9          C[A[i]] = C[A[i]] + 1;
10     // Indexação
11     for(i = 1; i < k; i++)
12         C[i] = C[i] + C[i - 1];
13     // Ordenação
14     for(i = n - 1; i >= 0; i--) {
15         B[C[A[i]] - 1] = A[i];
16         C[A[i]] = C[A[i]] - 1;
17     }
18 }
```

Counting sort

- Cálculo do histograma do vetor de entrada A
  - O alfabeto está definido como um dígito da base decimal, com os símbolos 0, 1, 2, 3, 4, 5, 6, 7, 8 e 9

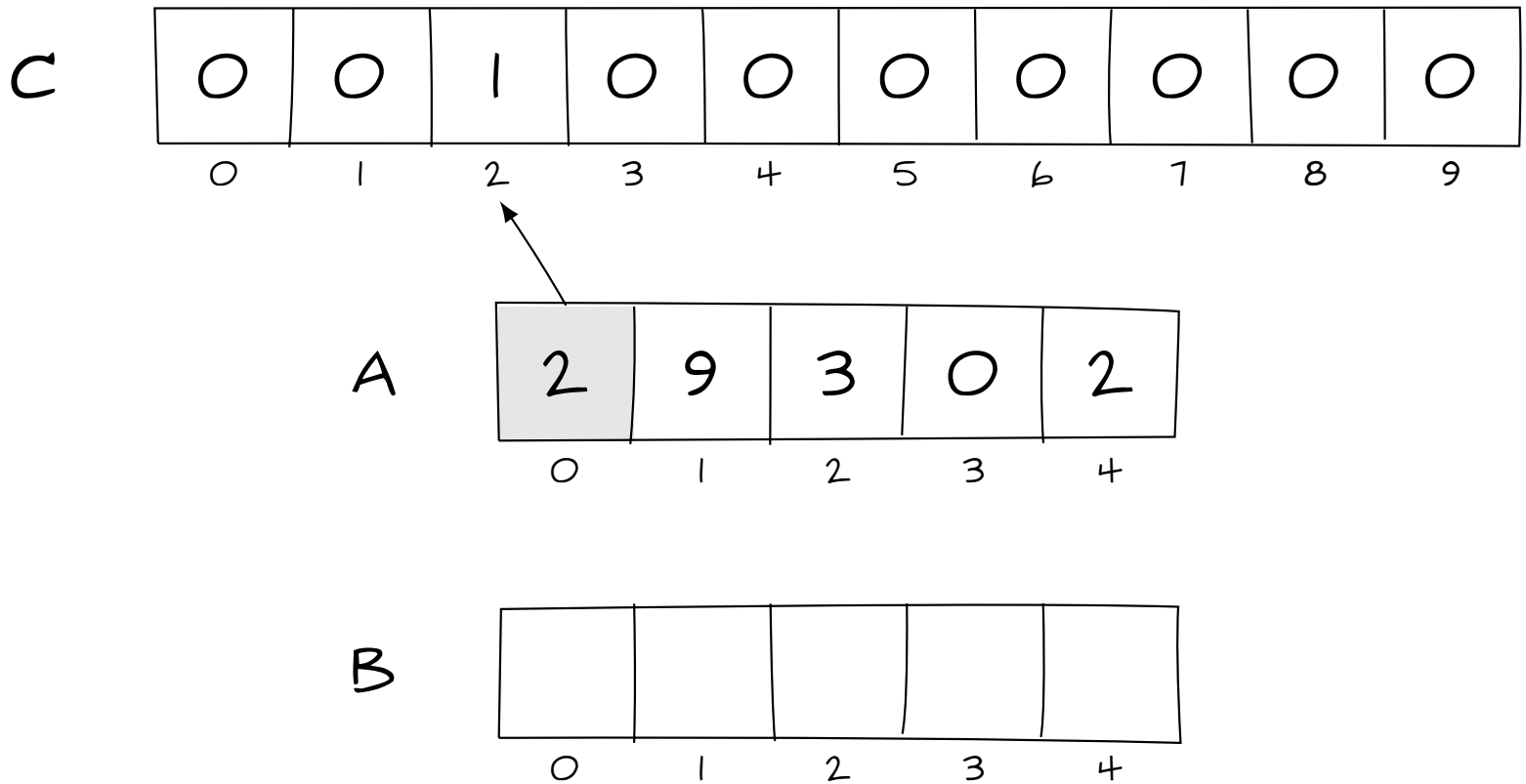






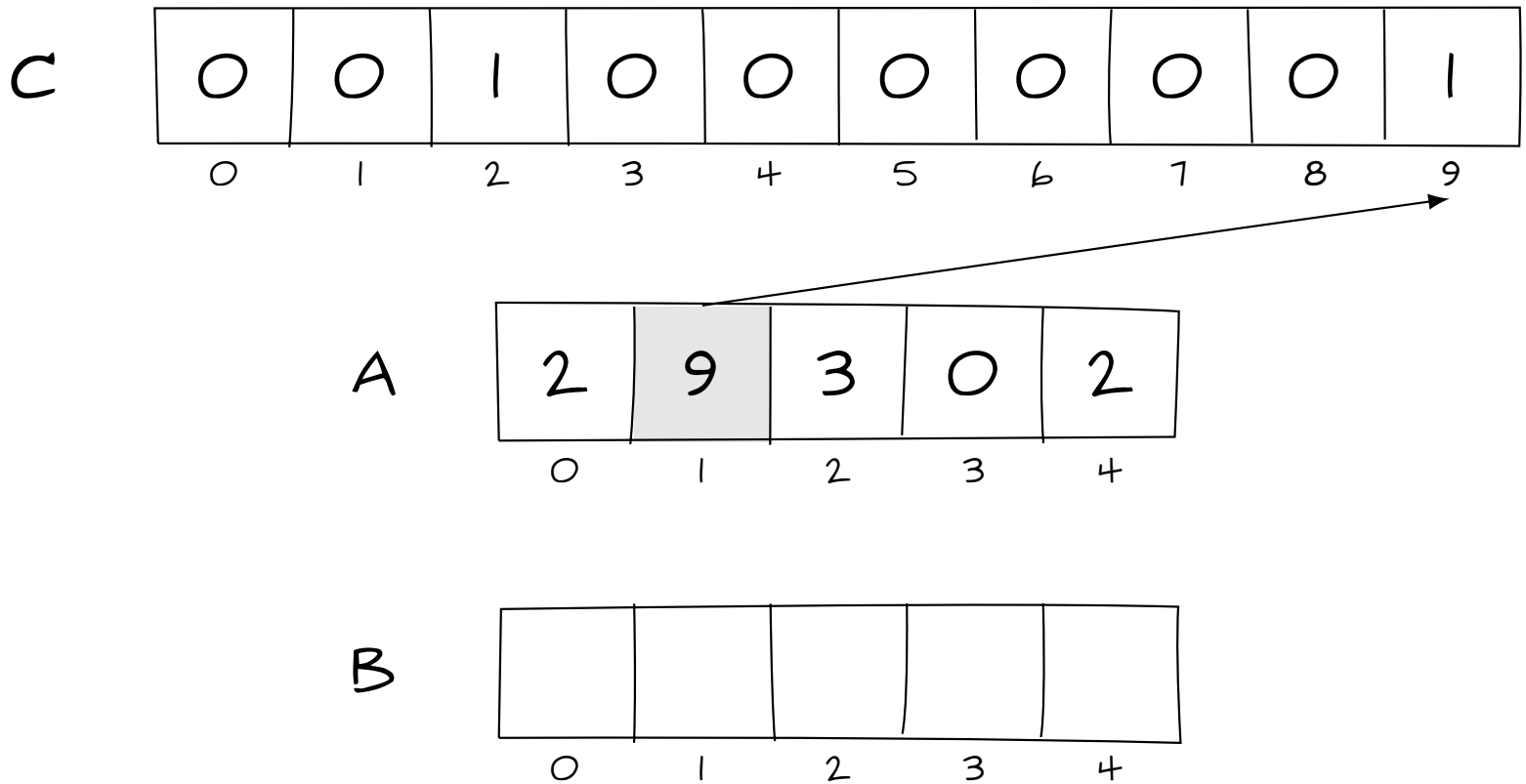
Counting sort

- Cálculo do histograma do vetor de entrada A
  - O alfabeto está definido como um dígito da base decimal, com os símbolos 0, 1, 2, 3, 4, 5, 6, 7, 8 e 9



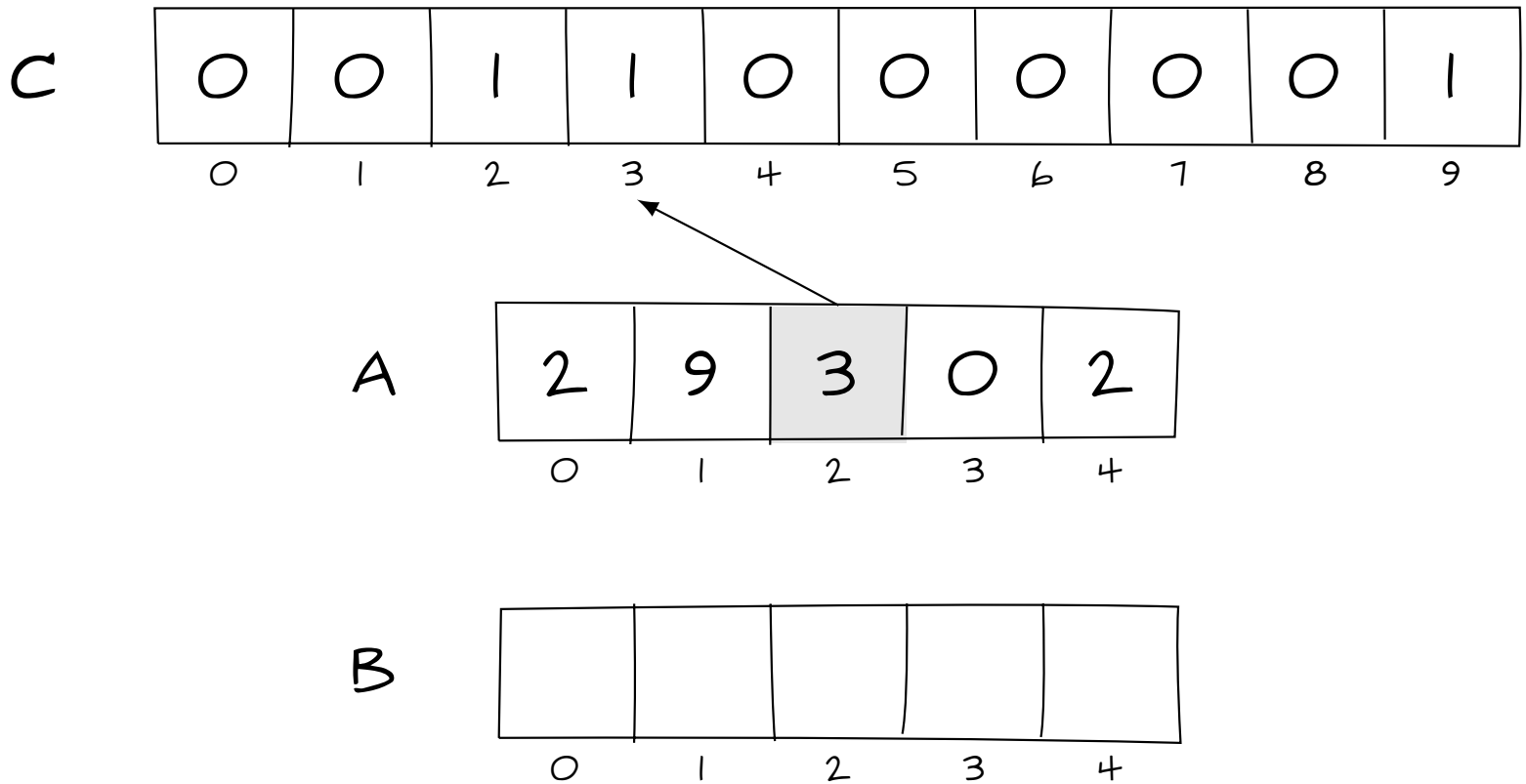
Counting sort

- Cálculo do histograma do vetor de entrada A
  - O alfabeto está definido como um dígito da base decimal, com os símbolos 0, 1, 2, 3, 4, 5, 6, 7, 8 e 9



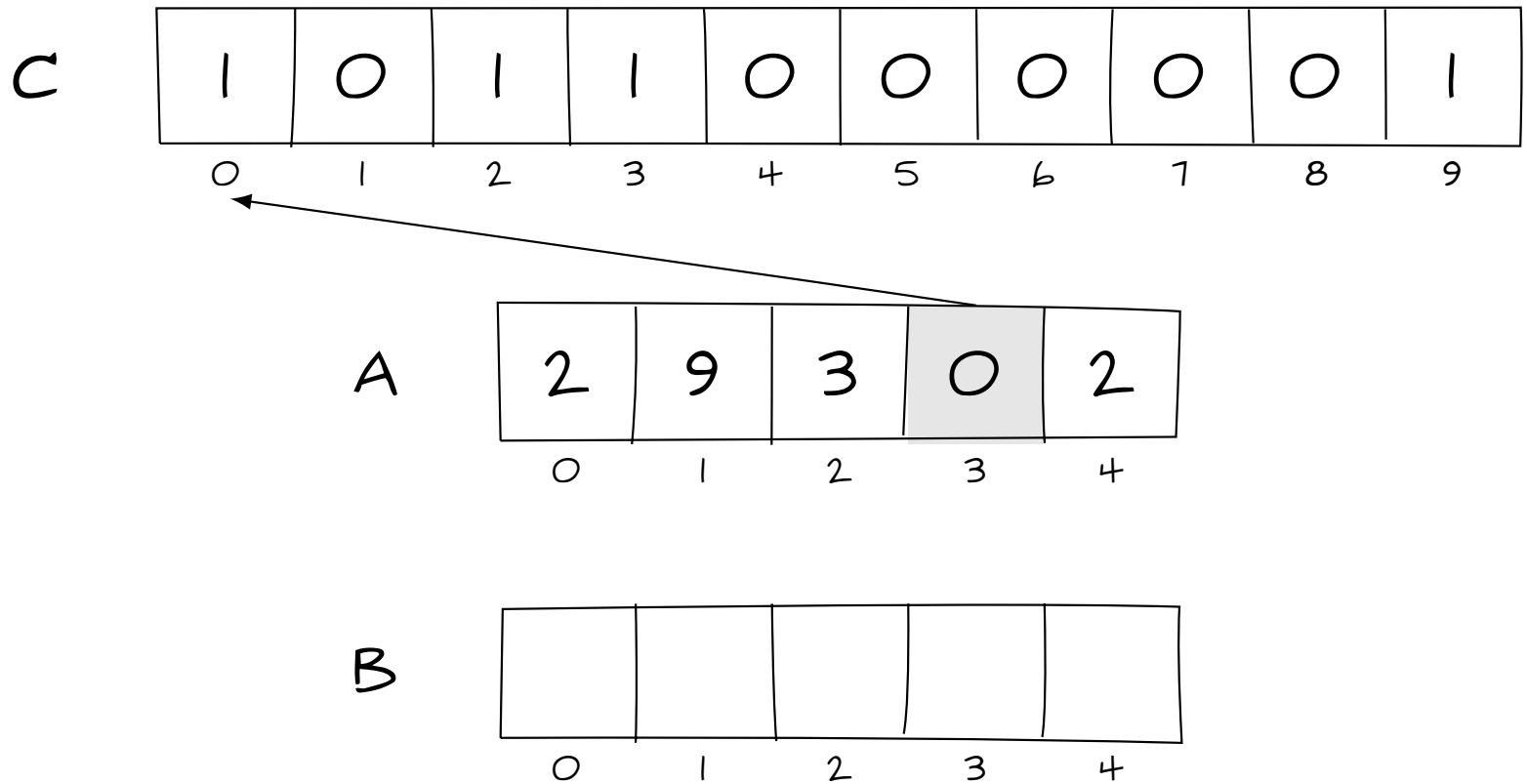
Counting sort

- Cálculo do histograma do vetor de entrada A
  - O alfabeto está definido como um dígito da base decimal, com os símbolos 0, 1, 2, 3, 4, 5, 6, 7, 8 e 9



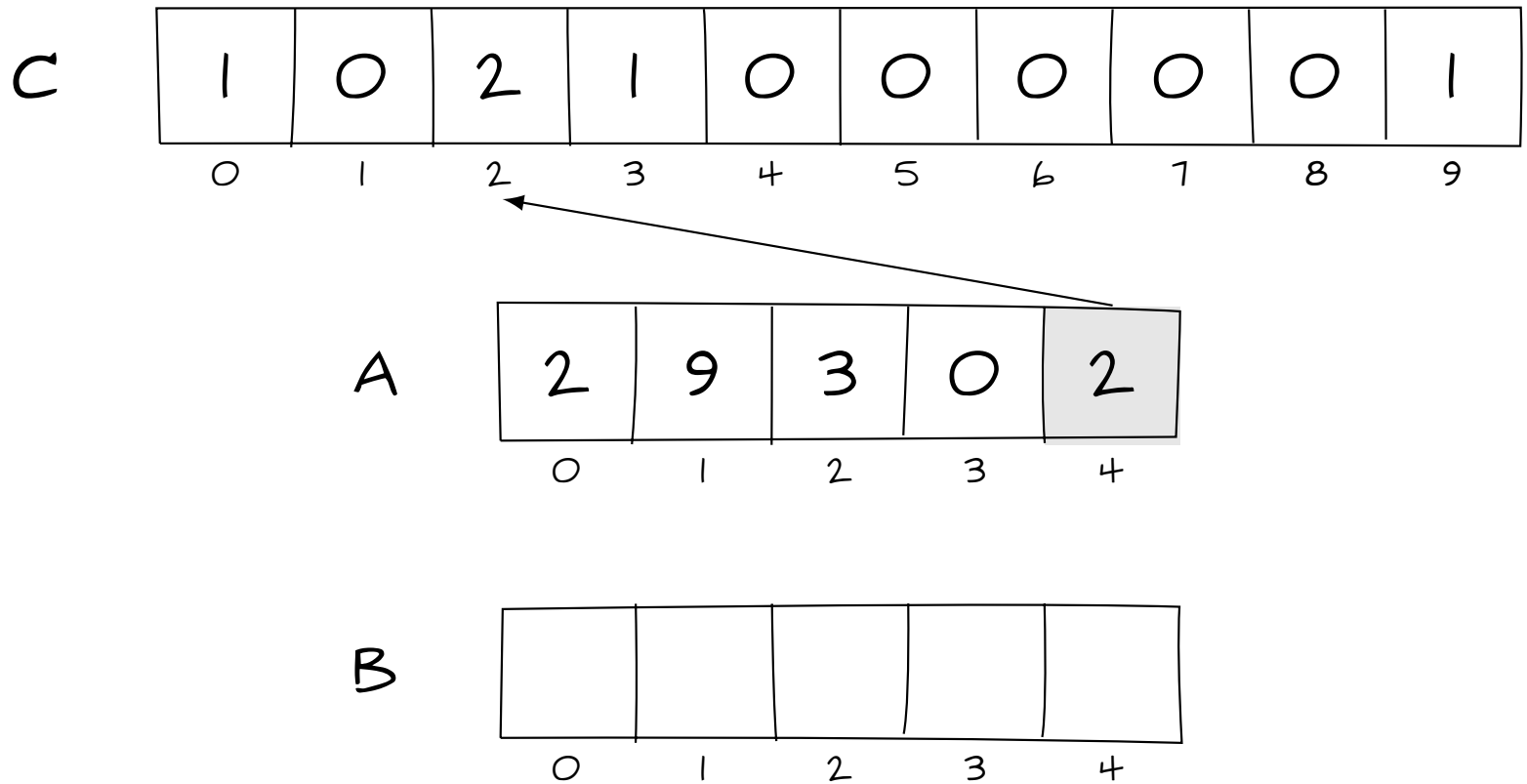
Counting sort

- Cálculo do histograma do vetor de entrada A
  - O alfabeto está definido como um dígito da base decimal, com os símbolos 0, 1, 2, 3, 4, 5, 6, 7, 8 e 9



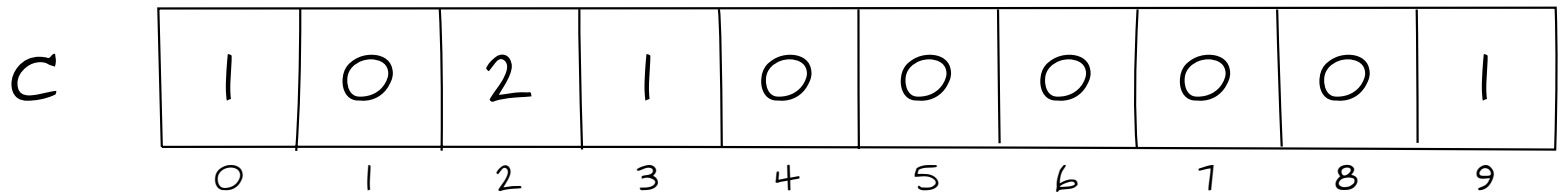
Counting sort

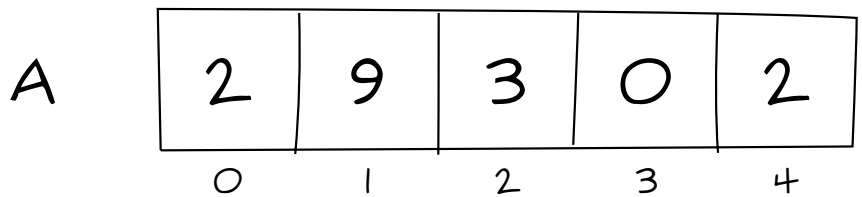
- Cálculo do histograma do vetor de entrada A
  - O alfabeto está definido como um dígito da base decimal, com os símbolos 0, 1, 2, 3, 4, 5, 6, 7, 8 e 9

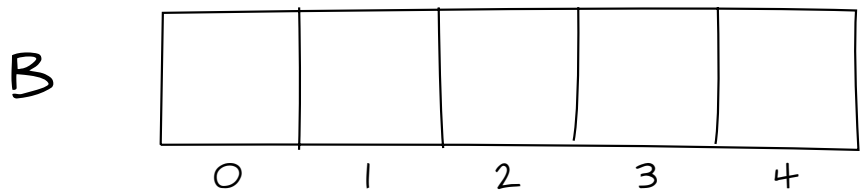


Counting sort

- Cálculo do histograma do vetor de entrada A
  - O alfabeto está definido como um dígito da base decimal, com os símbolos 0, 1, 2, 3, 4, 5, 6, 7, 8 e 9







Cada posição do vetor C armazena
a frequência de ocorrências dos símbolos

Counting sort

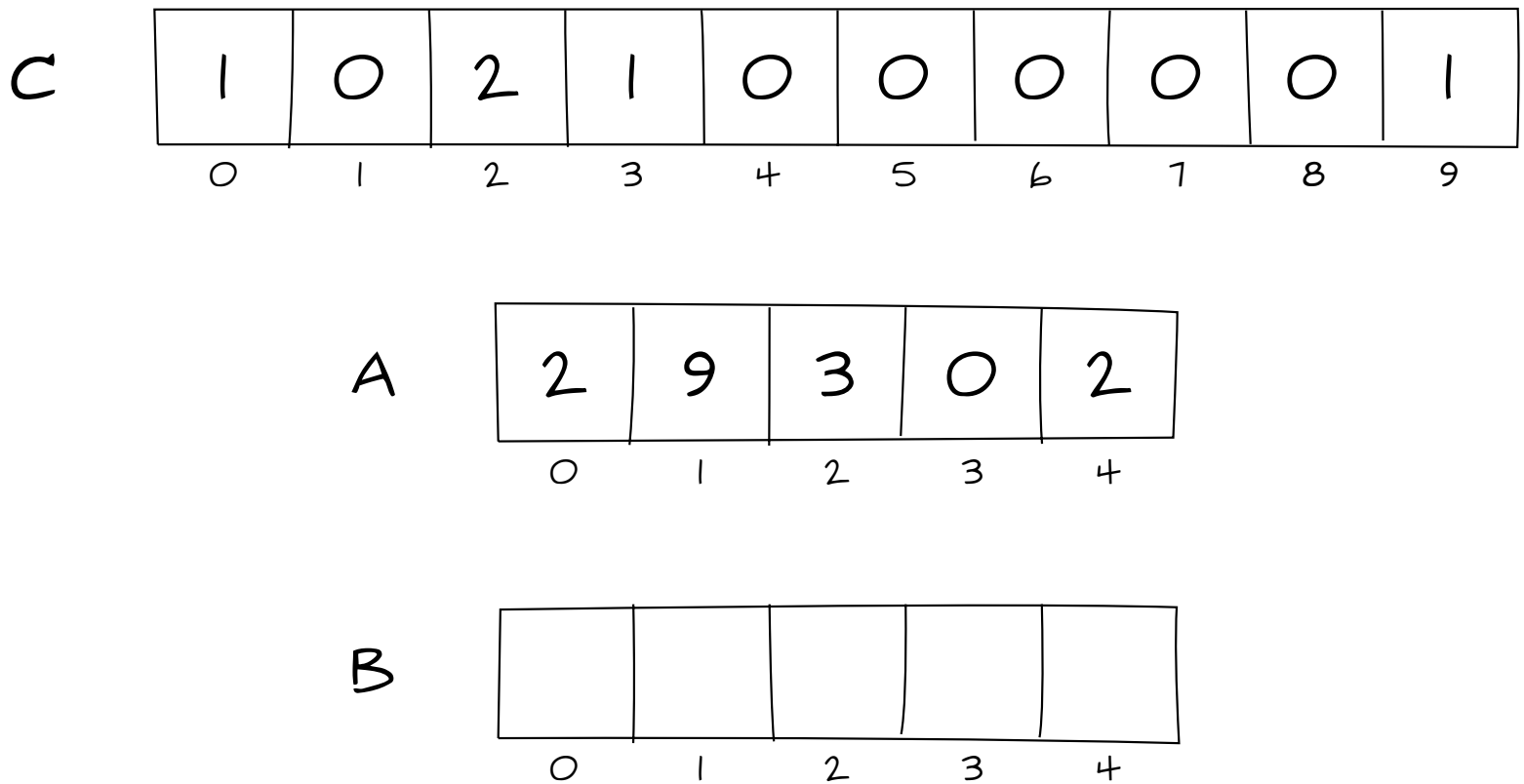
```
1  // Padrão de tipos por tamanho
2  #include <stdint.h>
3  // Procedimento counting sort
4  void counting_sort(int32_t A[], int32_t B[], uint32_t
       n, uint32_t k) {
5      // Vetor de contagem e indexação
6      uint32_t C[k] = { 0 };
7      // Histograma
8      for(uint32_t i = 0; i < n; i++)
9          C[A[i]] = C[A[i]] + 1;
10     // Indexação
11     for(i = 1; i < k; i++)
12         C[i] = C[i] + C[i - 1];
13     // Ordenação
14     for(i = n - 1; i >= 0; i--) {
15         B[C[A[i]] - 1] = A[i];
16         C[A[i]] = C[A[i]] - 1;
17     }
18 }
```

Counting sort

```
1  // Padrão de tipos por tamanho
2  #include <stdint.h>
3  // Procedimento counting sort
4  void counting_sort(int32_t A[], int32_t B[], uint32_t
       n, uint32_t k) {
5      // Vetor de contagem e indexação
6      uint32_t C[k] = { 0 };
7      // Histograma
8      for(uint32_t i = 0; i < n; i++)
9          C[A[i]] = C[A[i]] + 1;
10     // Indexação
11     for(i = 1; i < k; i++)
12         C[i] = C[i] + C[i - 1];
13     // Ordenação
14     for(i = n - 1; i >= 0; i--) {
15         B[C[A[i]] - 1] = A[i];
16         C[A[i]] = C[A[i]] - 1;
17     }
18 }
```

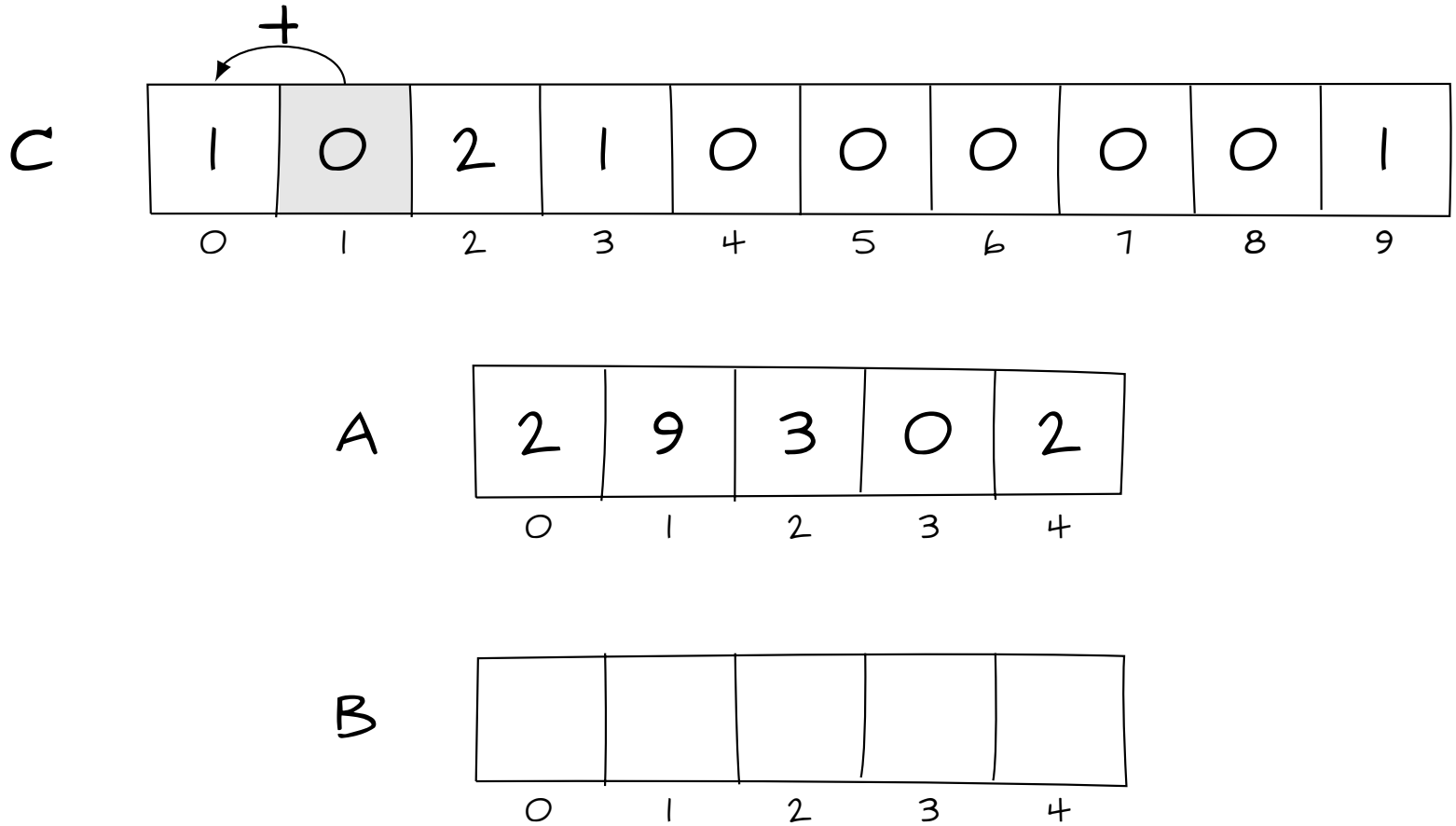
Counting sort

- Indexação do histograma do vetor de contagem C



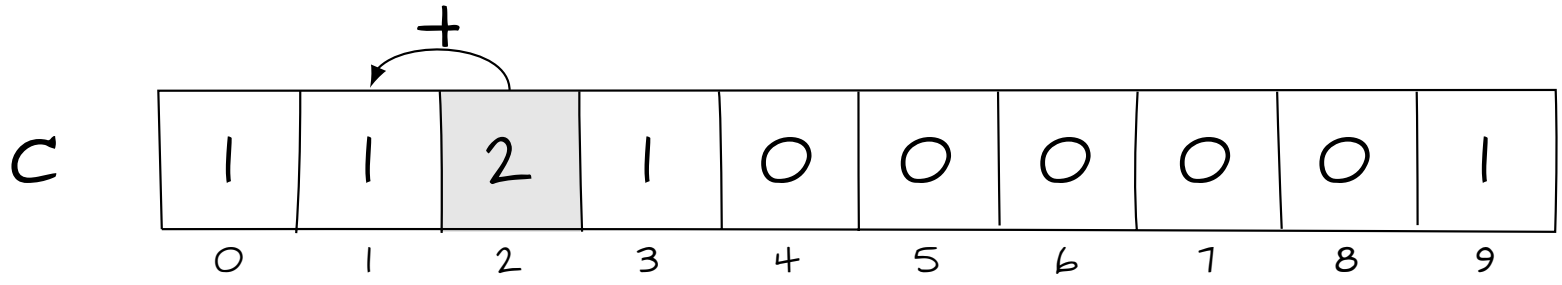
Counting sort

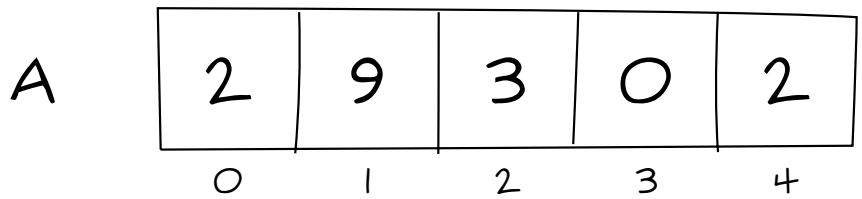
- Indexação do histograma do vetor de contagem C

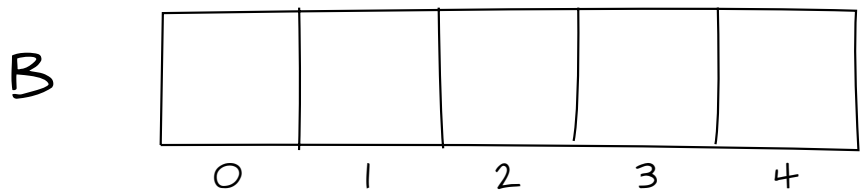


Counting sort

- Indexação do histograma do vetor de contagem C

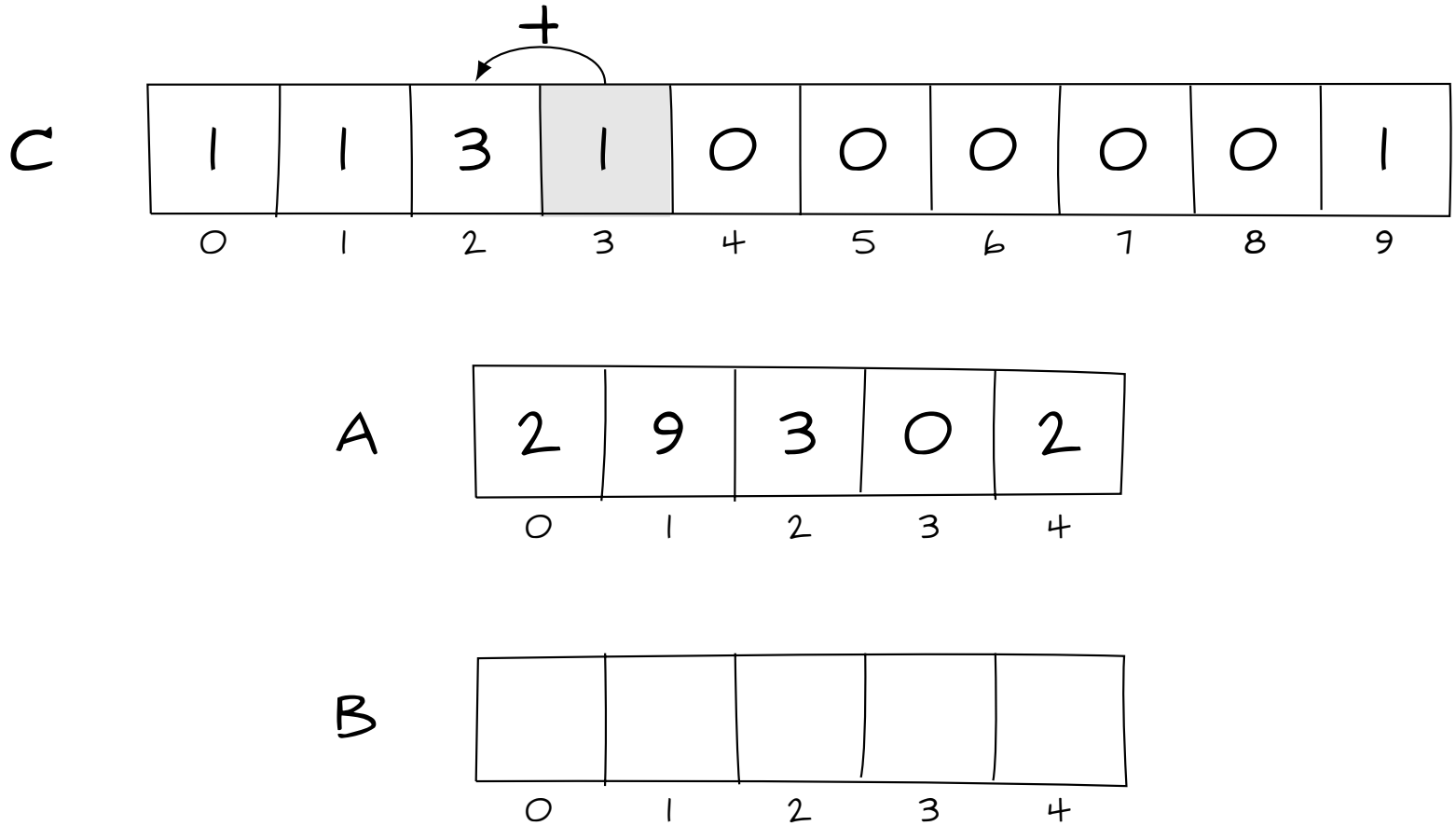






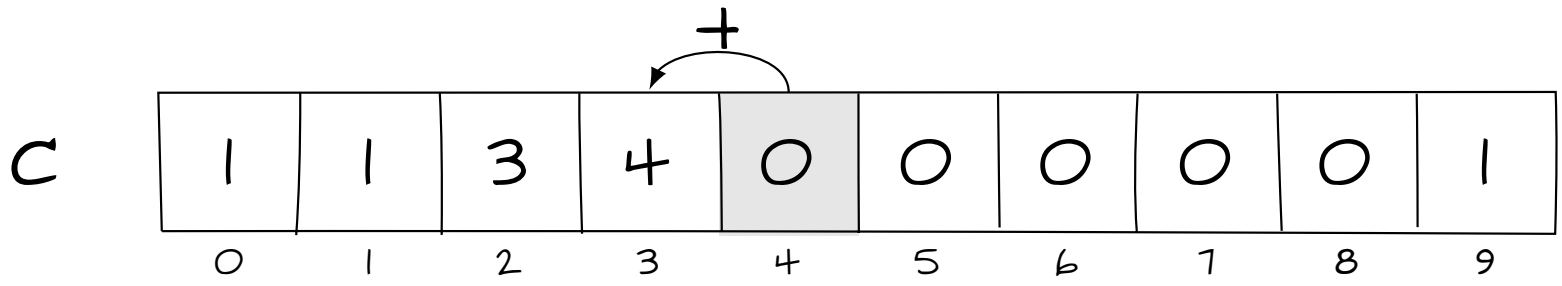
Counting sort

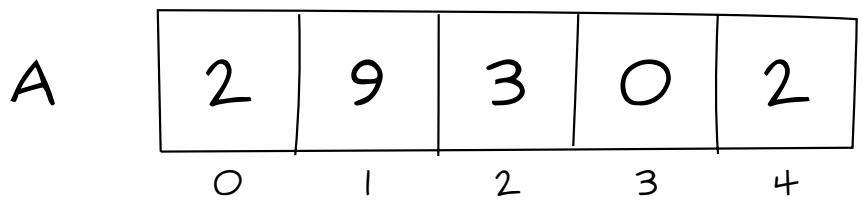
- Indexação do histograma do vetor de contagem C

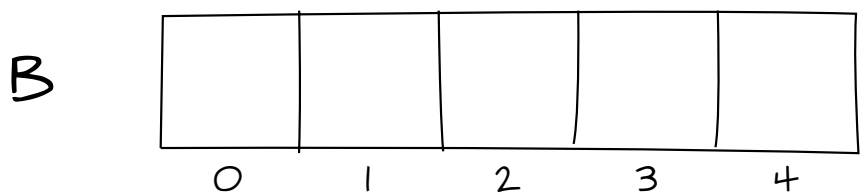


Counting sort

- Indexação do histograma do vetor de contagem C

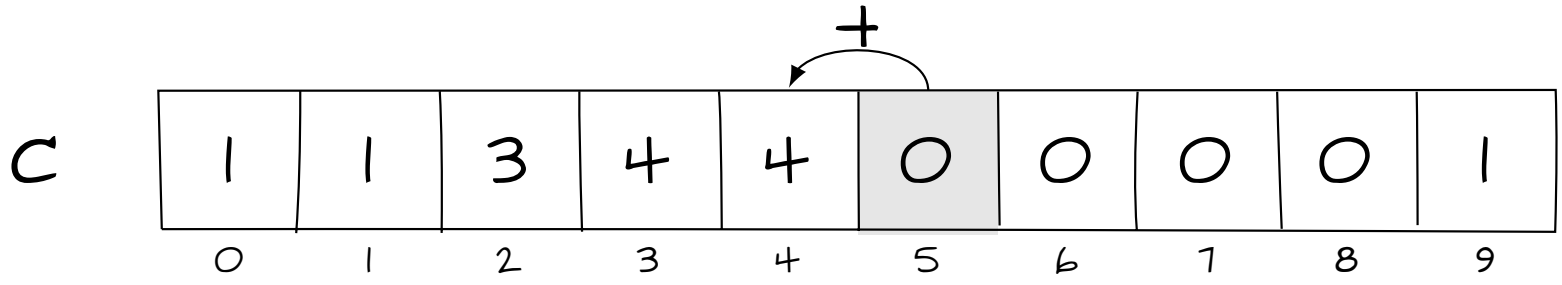


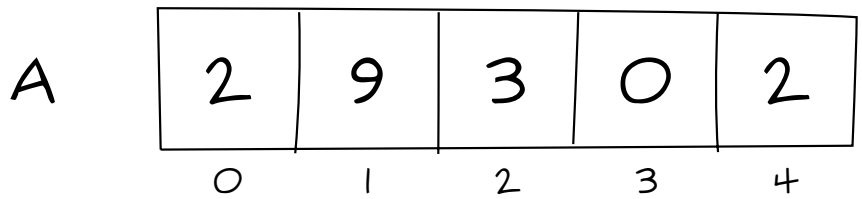


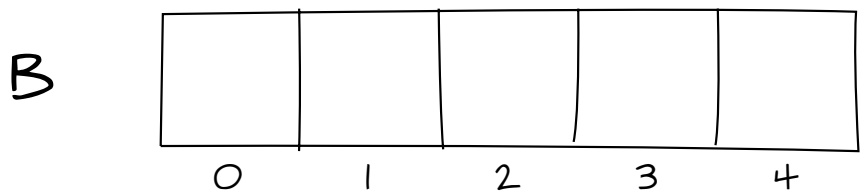


Counting sort

- Indexação do histograma do vetor de contagem C

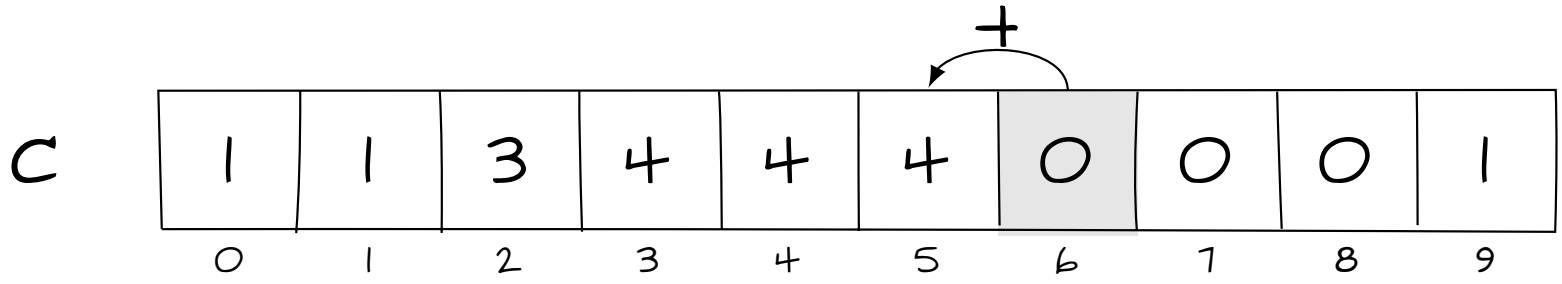


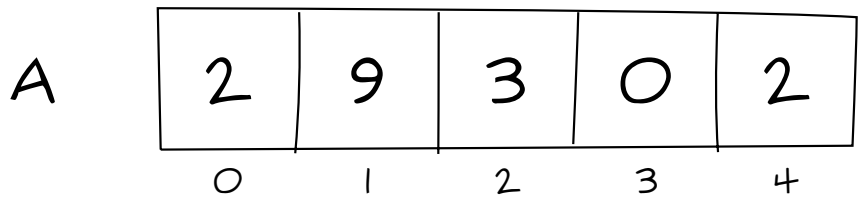


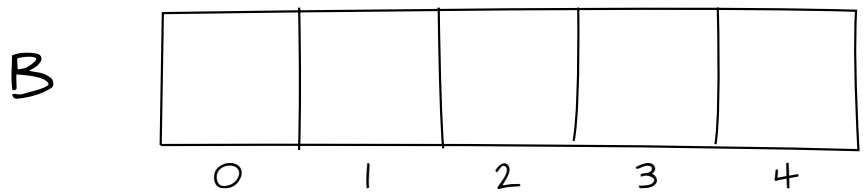


Counting sort

- Indexação do histograma do vetor de contagem C

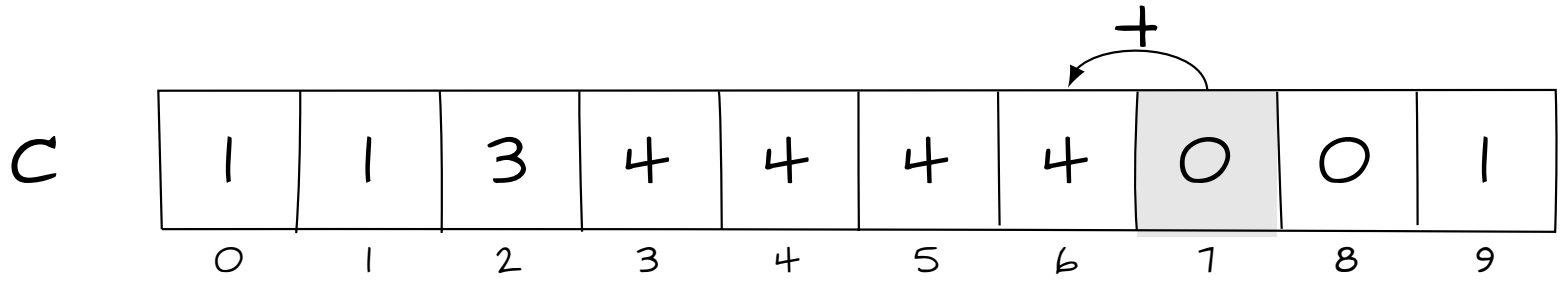


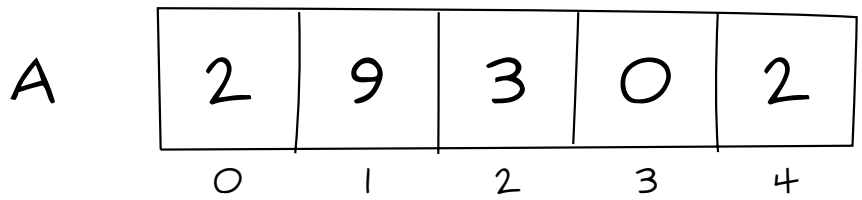


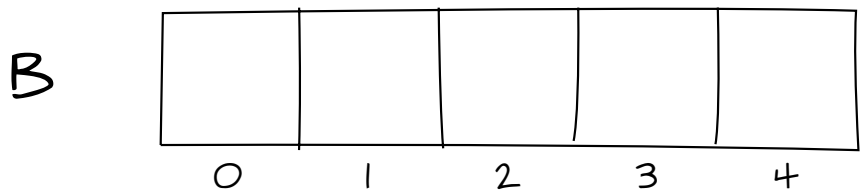


Counting sort

- Indexação do histograma do vetor de contagem C

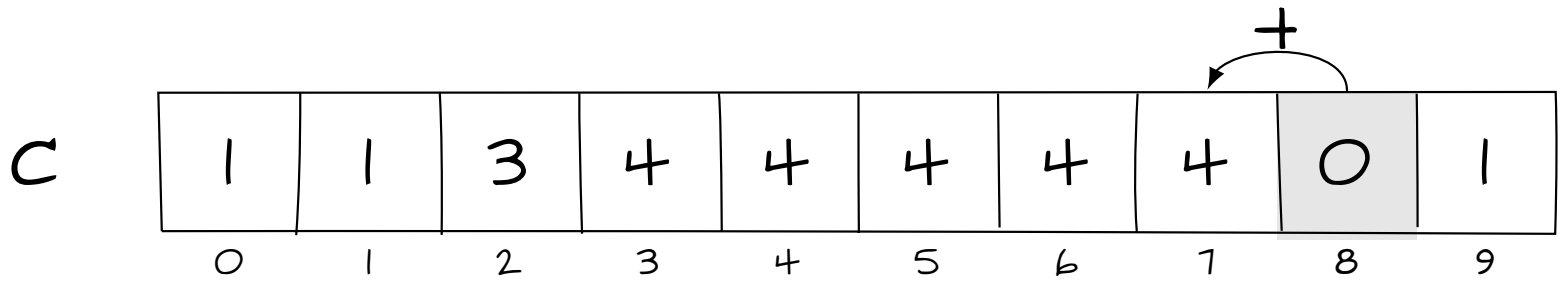


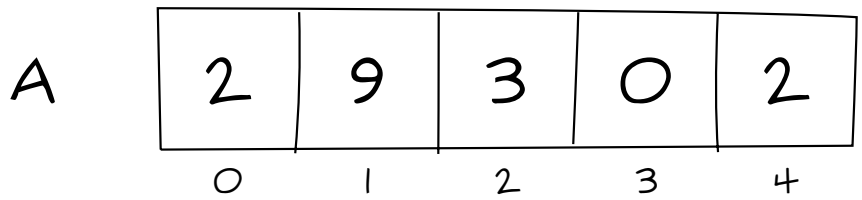


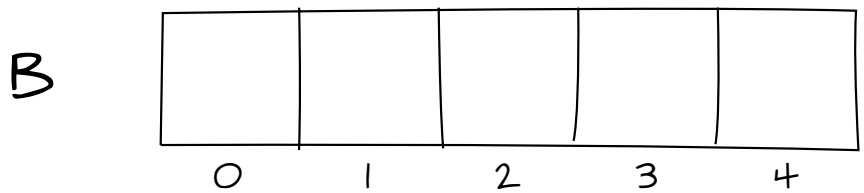


Counting sort

- Indexação do histograma do vetor de contagem C

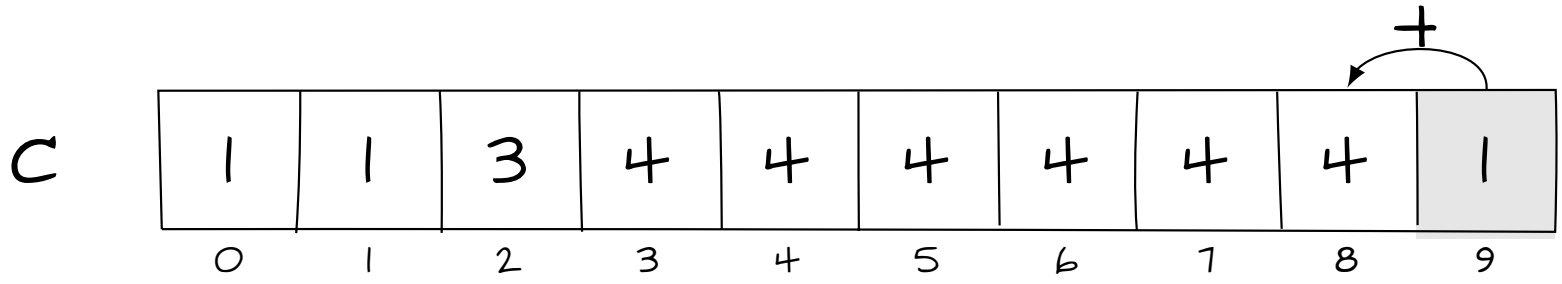


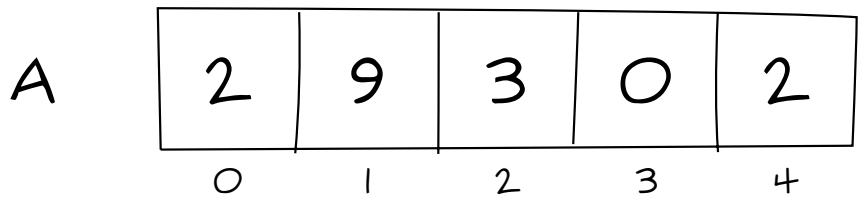


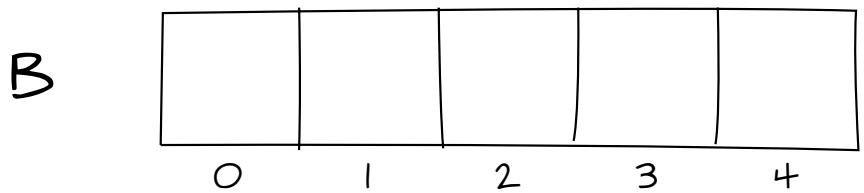


Counting sort

- Indexação do histograma do vetor de contagem C

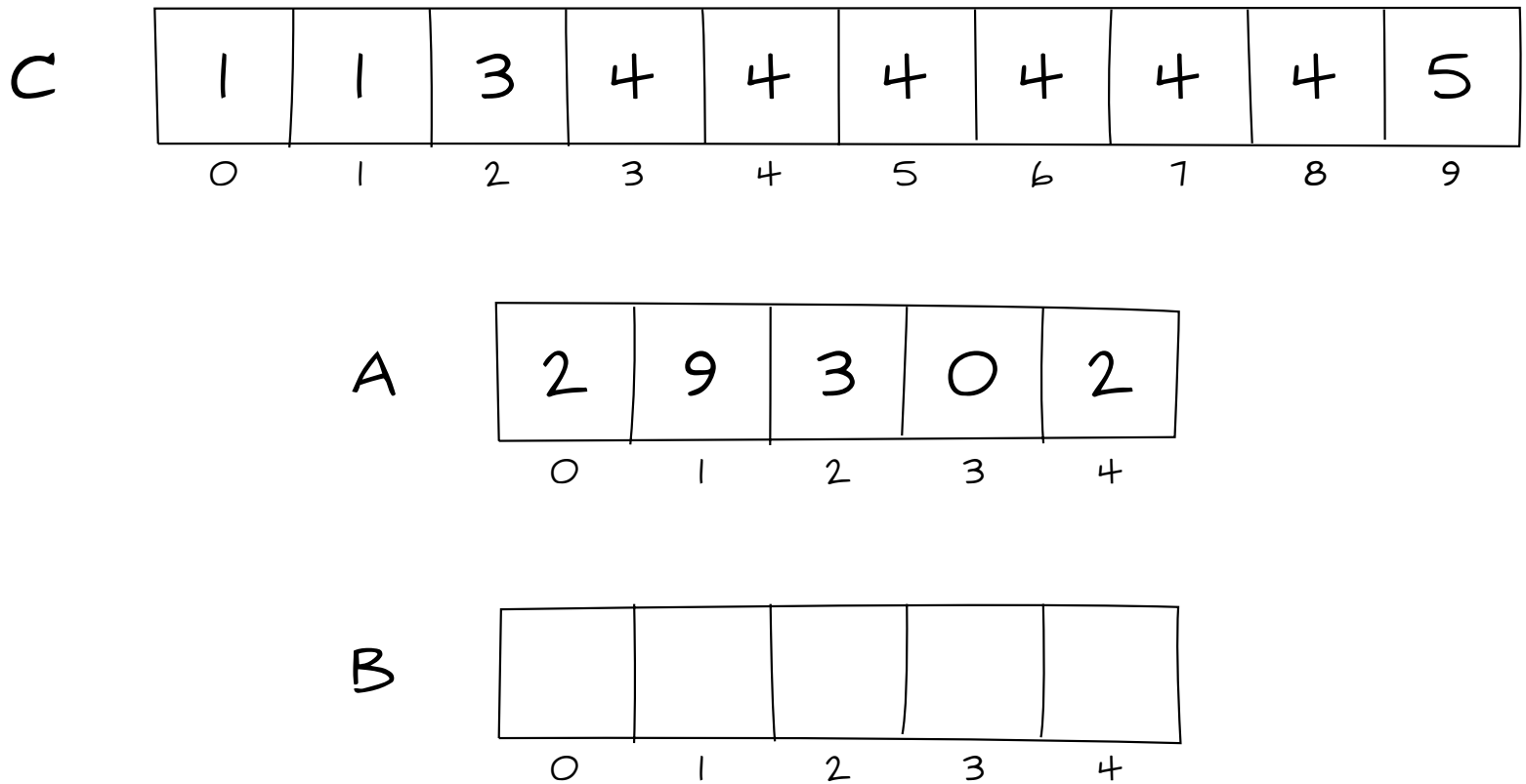






Counting sort

- Indexação do histograma do vetor de contagem C



Cada posição do vetor C armazena
o índice dos símbolos para ordenação

Counting sort

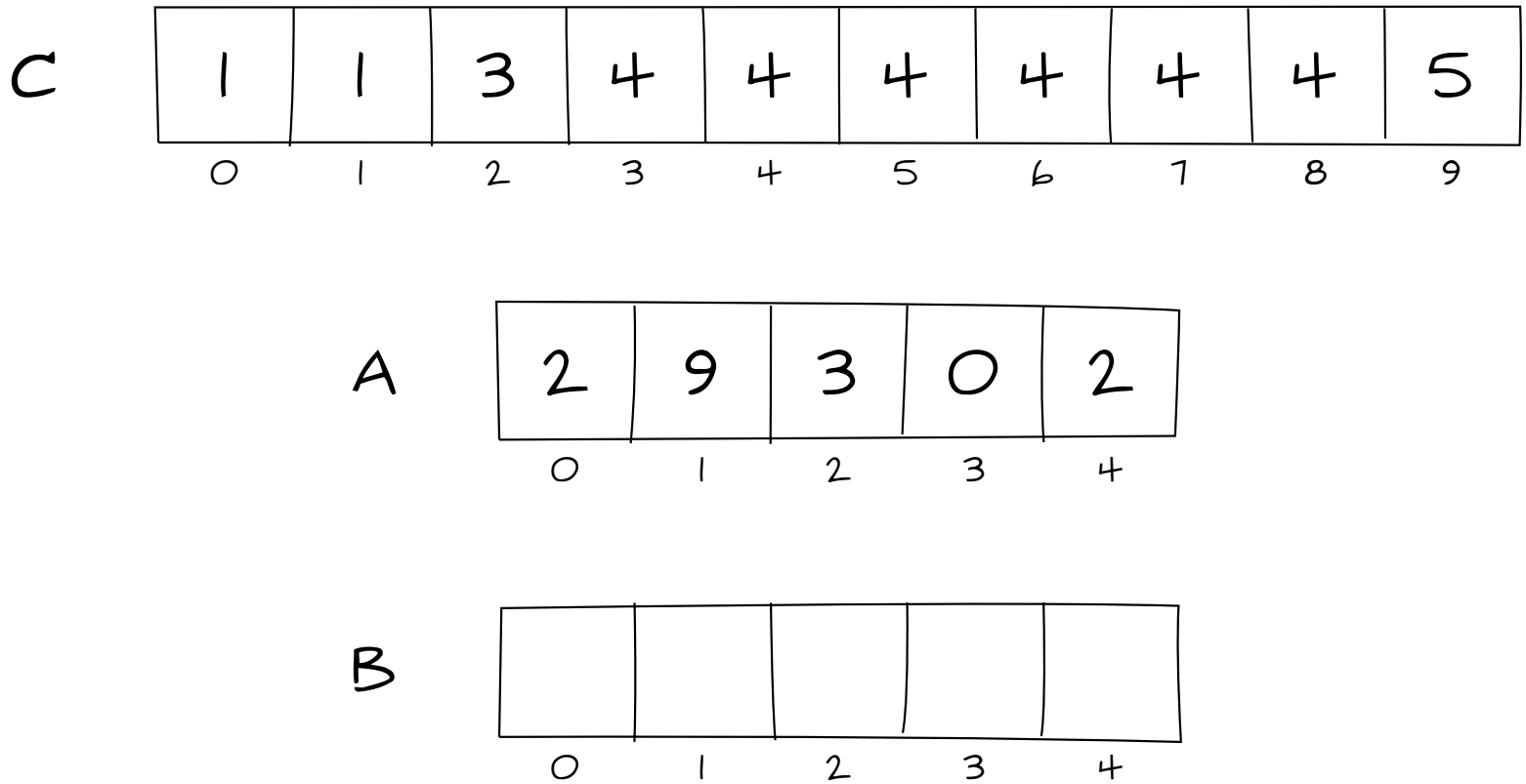
```
1  // Padrão de tipos por tamanho
2  #include <stdint.h>
3  // Procedimento counting sort
4  void counting_sort(int32_t A[], int32_t B[], uint32_t
       n, uint32_t k) {
5      // Vetor de contagem e indexação
6      uint32_t C[k] = { 0 };
7      // Histograma
8      for(uint32_t i = 0; i < n; i++)
9          C[A[i]] = C[A[i]] + 1;
10     // Indexação
11     for(i = 1; i < k; i++)
12         C[i] = C[i] + C[i - 1];
13     // Ordenação
14     for(i = n - 1; i >= 0; i--) {
15         B[C[A[i]] - 1] = A[i];
16         C[A[i]] = C[A[i]] - 1;
17     }
18 }
```

Counting sort

```
1  // Padrão de tipos por tamanho
2  #include <stdint.h>
3  // Procedimento counting sort
4  void counting_sort(int32_t A[], int32_t B[], uint32_t
       n, uint32_t k) {
5      // Vetor de contagem e indexação
6      uint32_t C[k] = { 0 };
7      // Histograma
8      for(uint32_t i = 0; i < n; i++)
9          C[A[i]] = C[A[i]] + 1;
10     // Indexação
11     for(i = 1; i < k; i++)
12         C[i] = C[i] + C[i - 1];
13     // Ordenação
14     for(i = n - 1; i >= 0; i--) {
15         B[C[A[i]] - 1] = A[i];
16         C[A[i]] = C[A[i]] - 1;
17     }
18 }
```

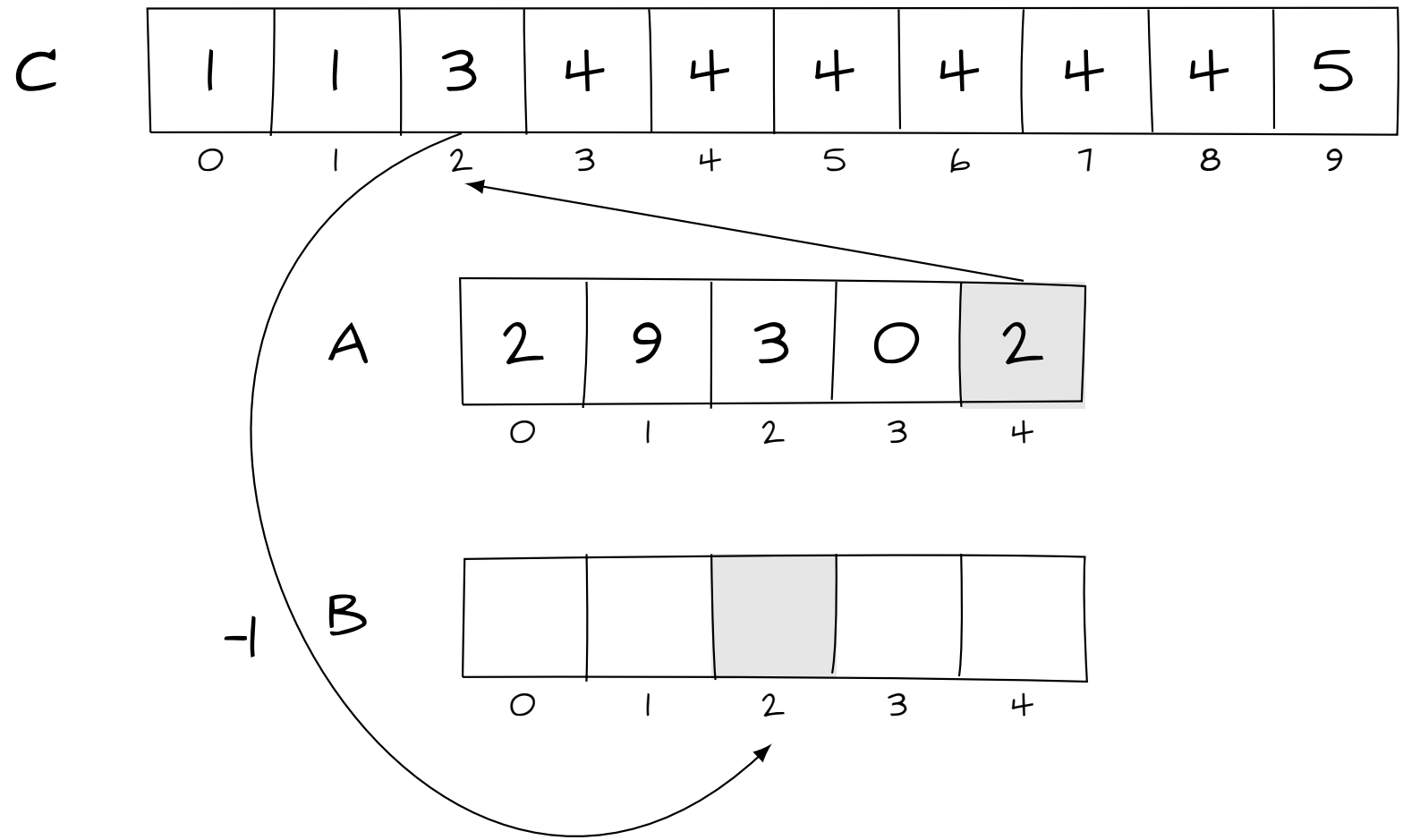
Counting sort

- Ordenação do vetor A com saída no vetor B



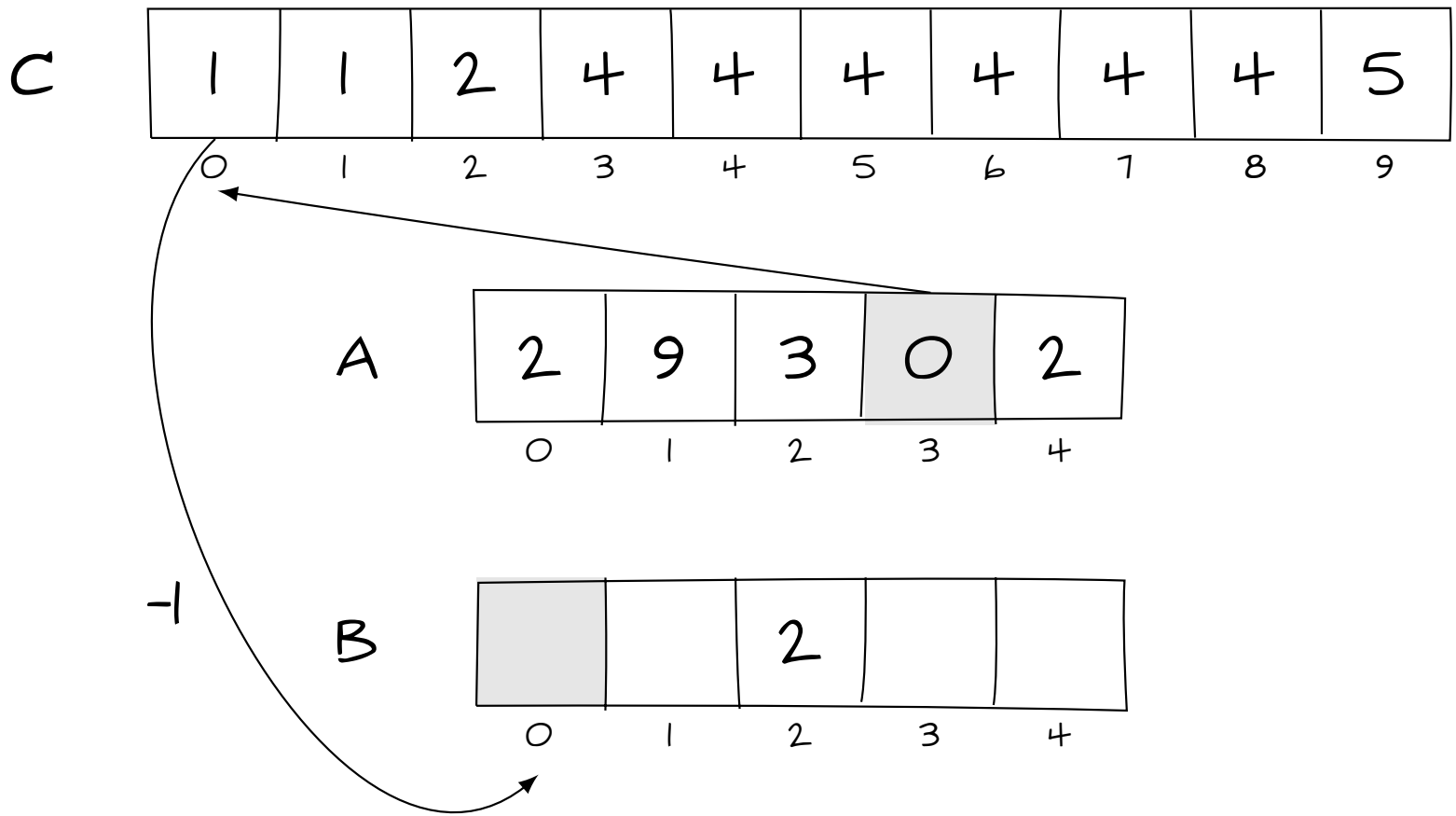
Counting sort

- Ordenação do vetor A com saída no vetor B



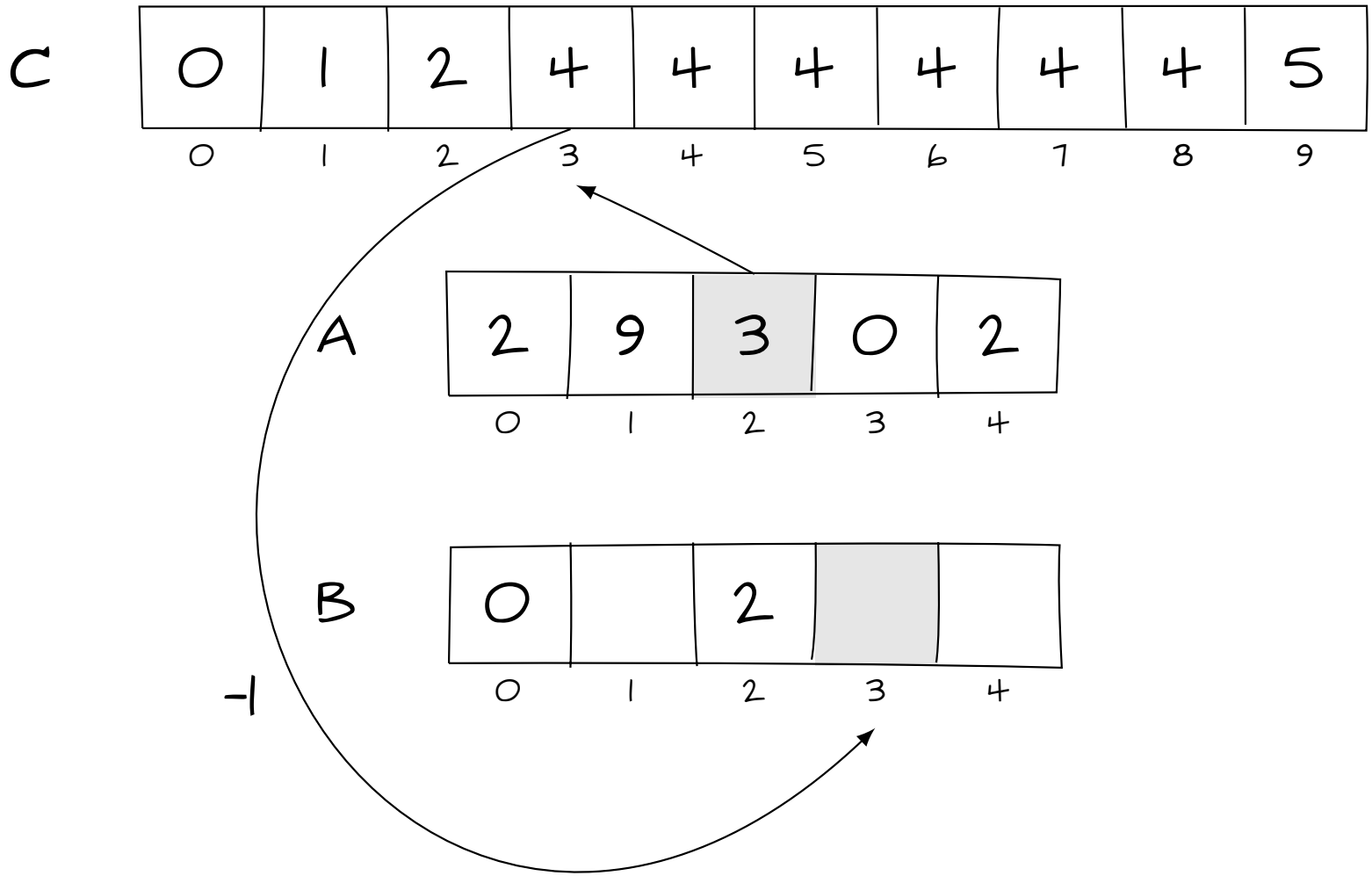
Counting sort

- Ordenação do vetor A com saída no vetor B



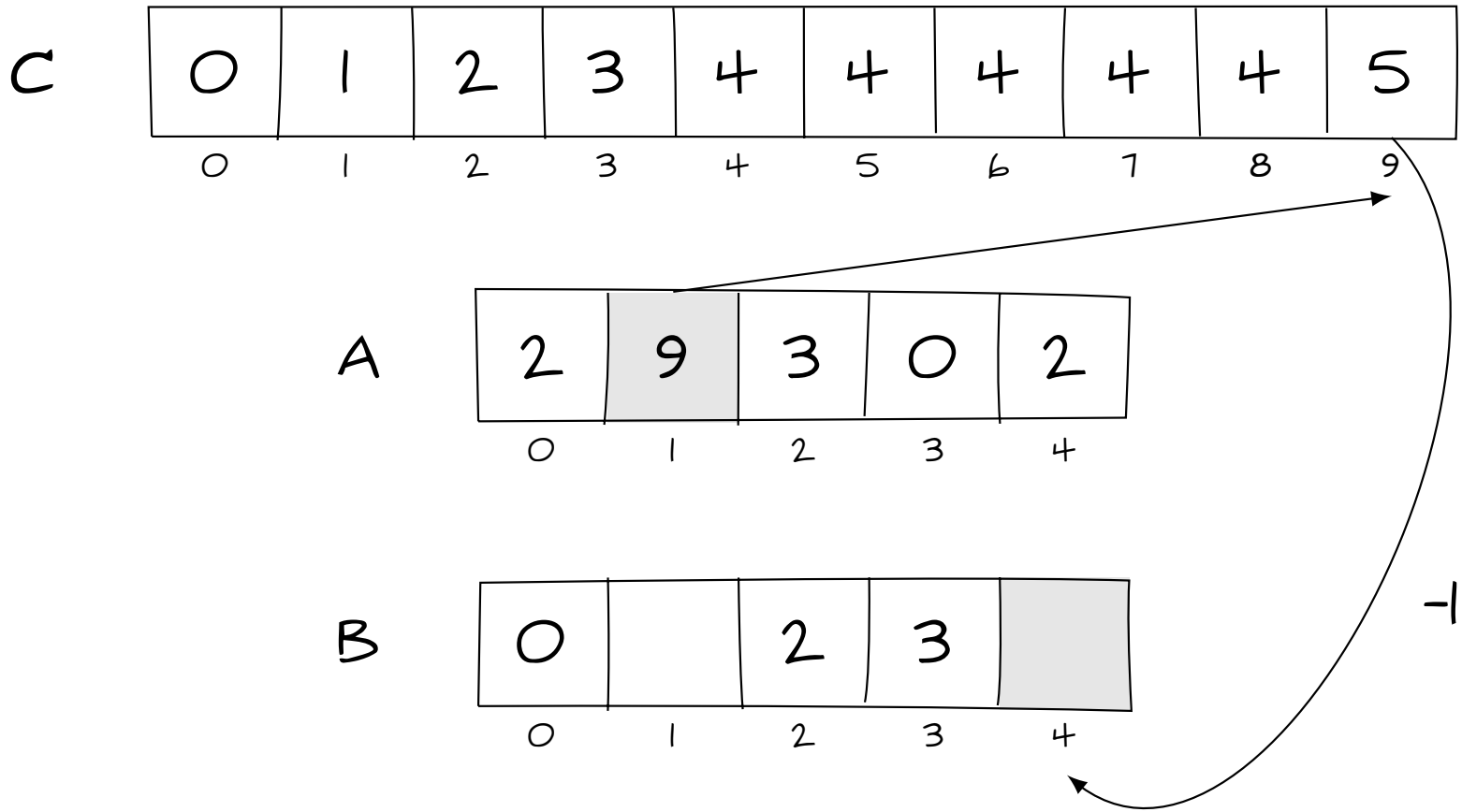
Counting sort

- Ordenação do vetor A com saída no vetor B



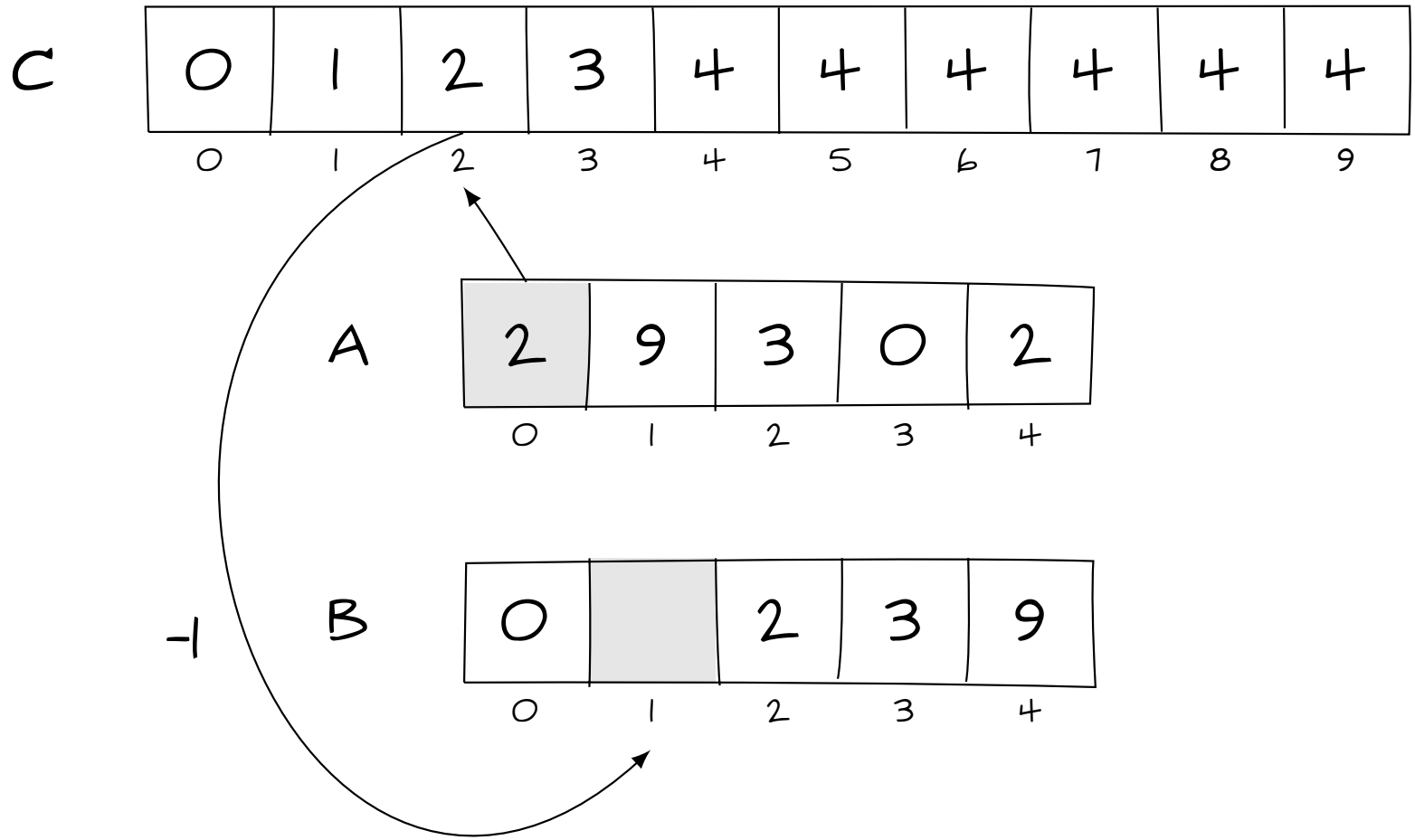
Counting sort

- Ordenação do vetor A com saída no vetor B



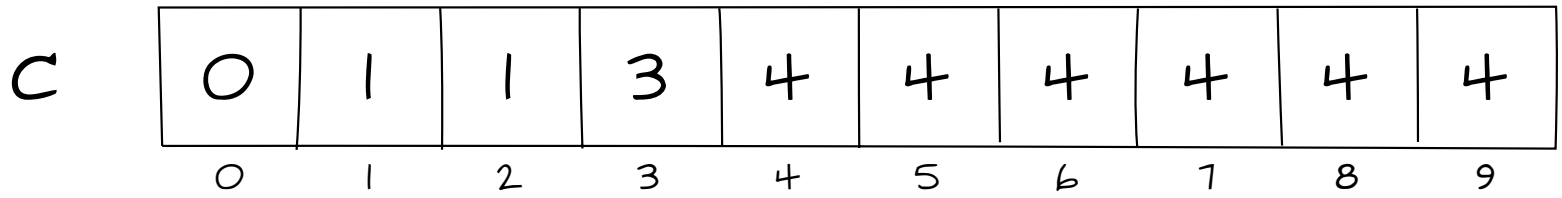
Counting sort

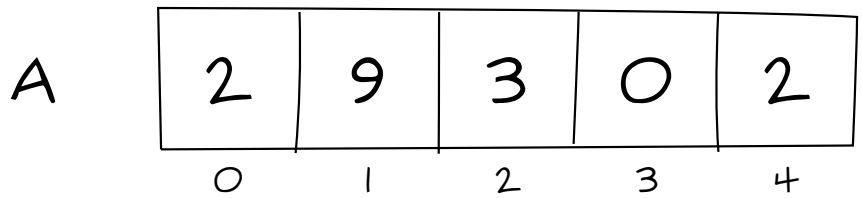
- Ordenação do vetor A com saída no vetor B

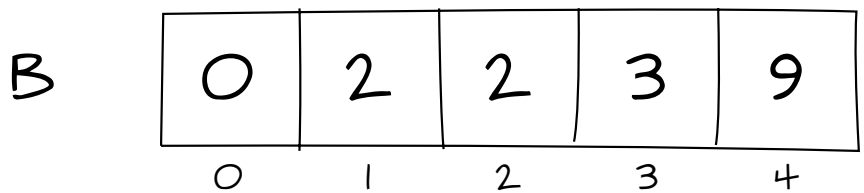


Counting sort

- Ordenação do vetor A com saída no vetor B







Counting sort

- Características do Counting sort
  - ✓ Possui complexidade de espaço e de tempo $\Theta(n + k)$

Counting sort

- Características do Counting sort
  - ✓ Possui complexidade de espaço e de tempo $\Theta(n + k)$
  - ✓ É estável, preservando a ordem relativa dos elementos

Counting sort

- Características do Counting sort
  - ✓ Possui complexidade de espaço e de tempo $\Theta(n + k)$
  - ✓ É estável, preservando a ordem relativa dos elementos
  - ✗ O tamanho do alfabeto deve ser conhecido e limitado

Counting sort

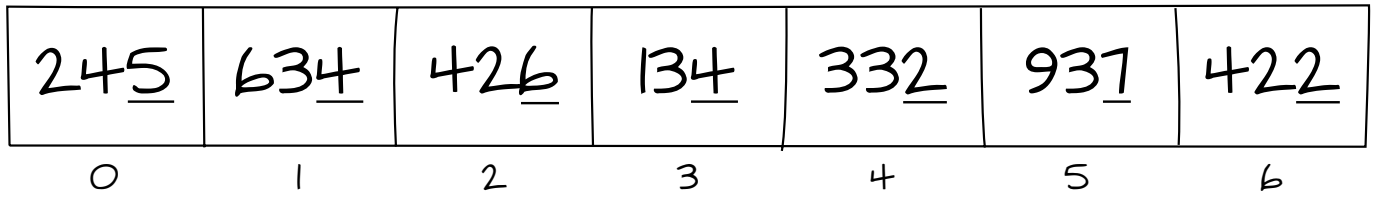
- Características do Counting sort
  - ✓ Possui complexidade de espaço e de tempo $\Theta(n + k)$
  - ✓ É estável, preservando a ordem relativa dos elementos
  - ✗ O tamanho do alfabeto deve ser conhecido e limitado
  - ✗ Não suporta símbolos negativos na entrada

Radix sort

```
1  // Padrão de tipos por tamanho
2  #include <stdint.h>
3  // Procedimento radix sort
4  void radix_sort(int32_t A[], uint32_t d) {
5      // Iterando nos dígitos
6      for(uint32_t i = 0; i < d; i++)
7          // Ordenação estável e linear
8          linearsort(A, i);
9  }
```

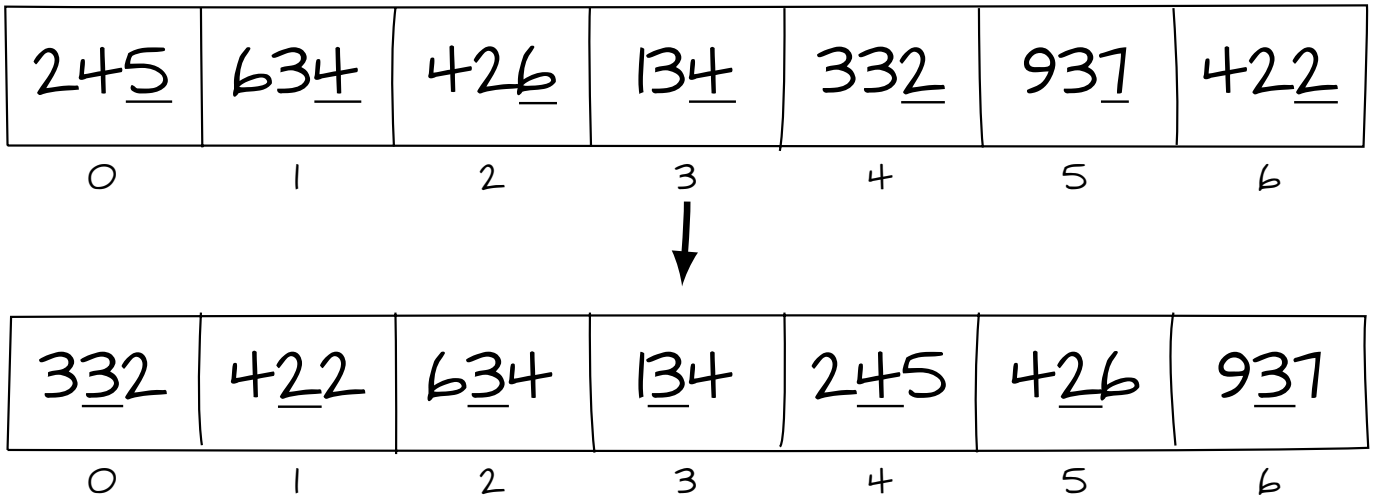
Radix sort

- Processo de ordenação dos dígitos



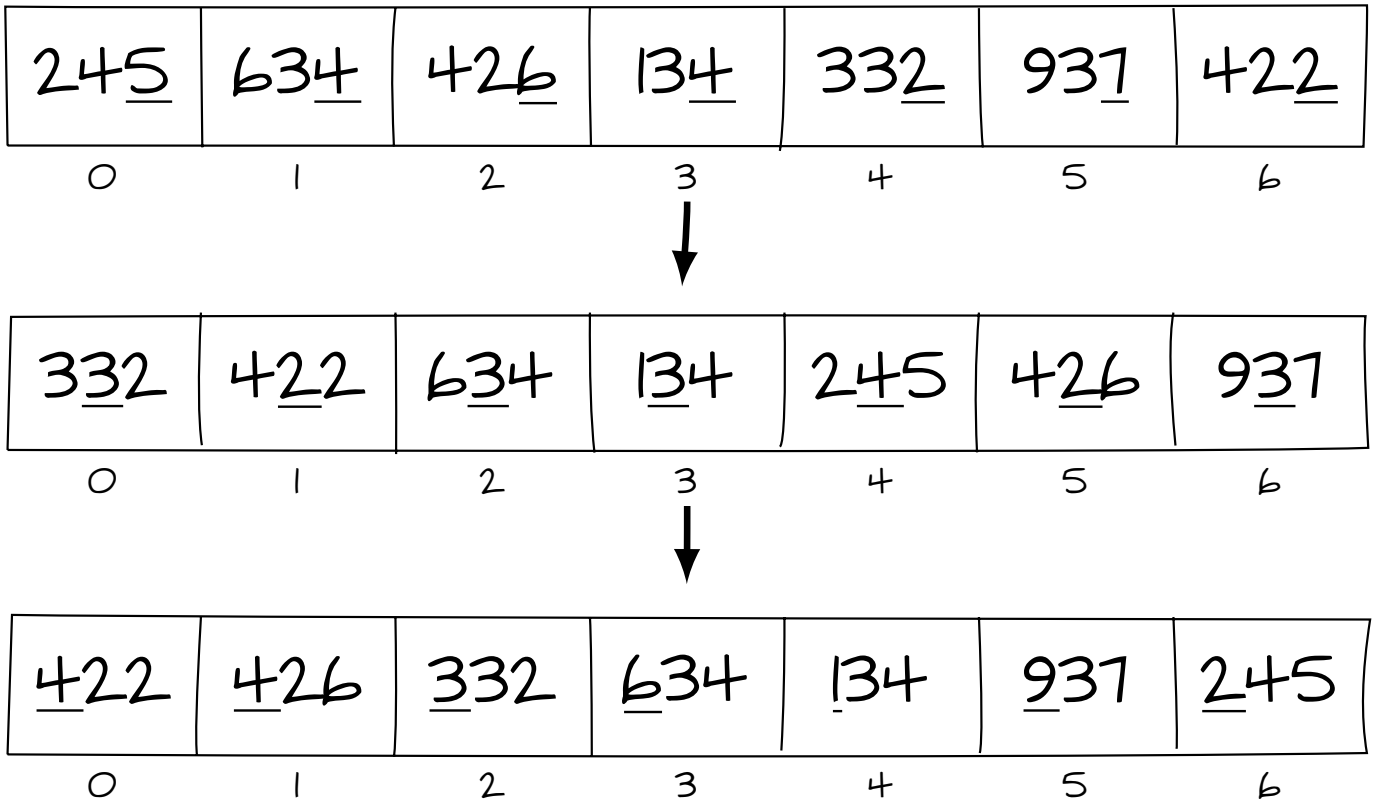
Radix sort

- Processo de ordenação dos dígitos



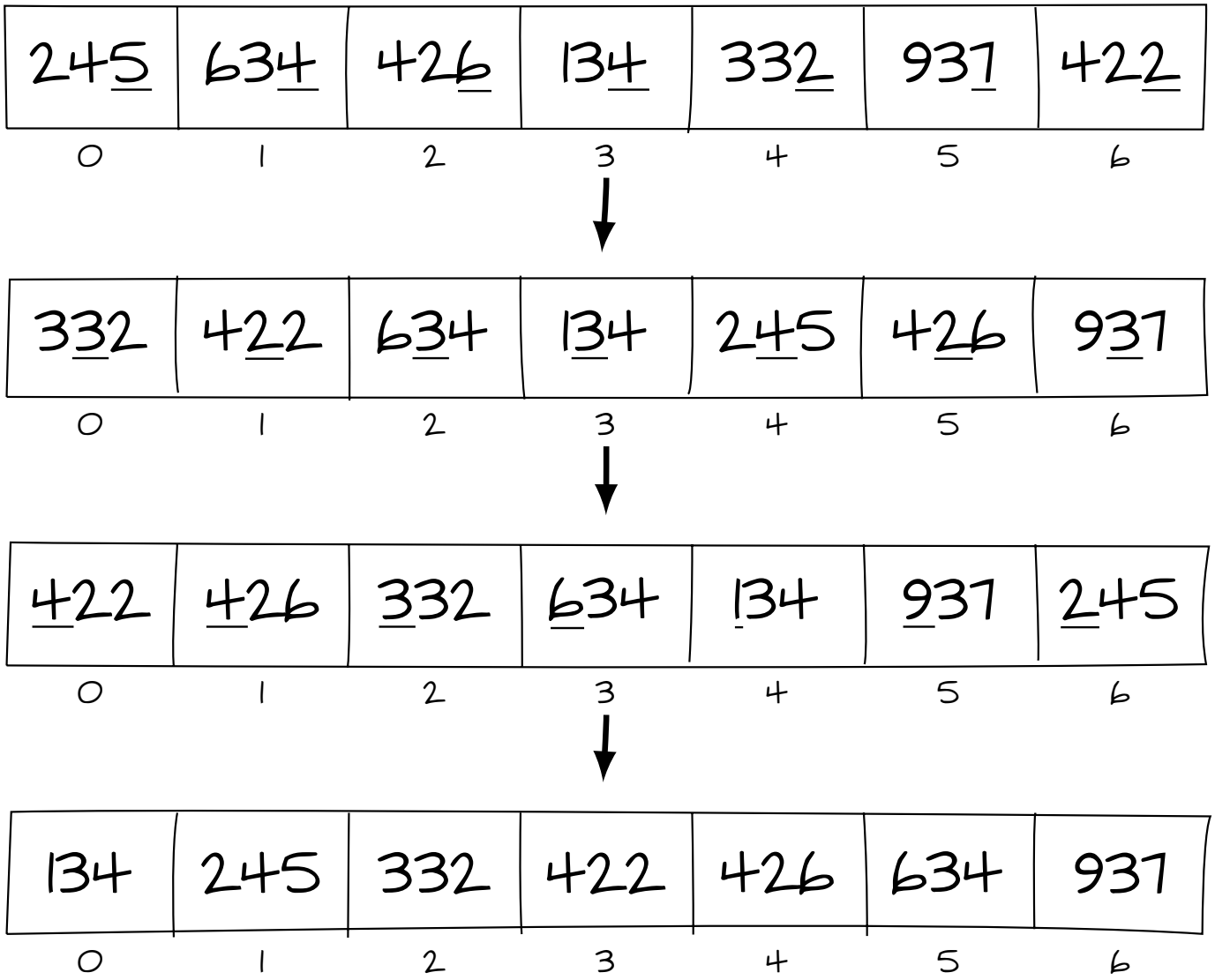
Radix sort

- Processo de ordenação dos dígitos



Radix sort

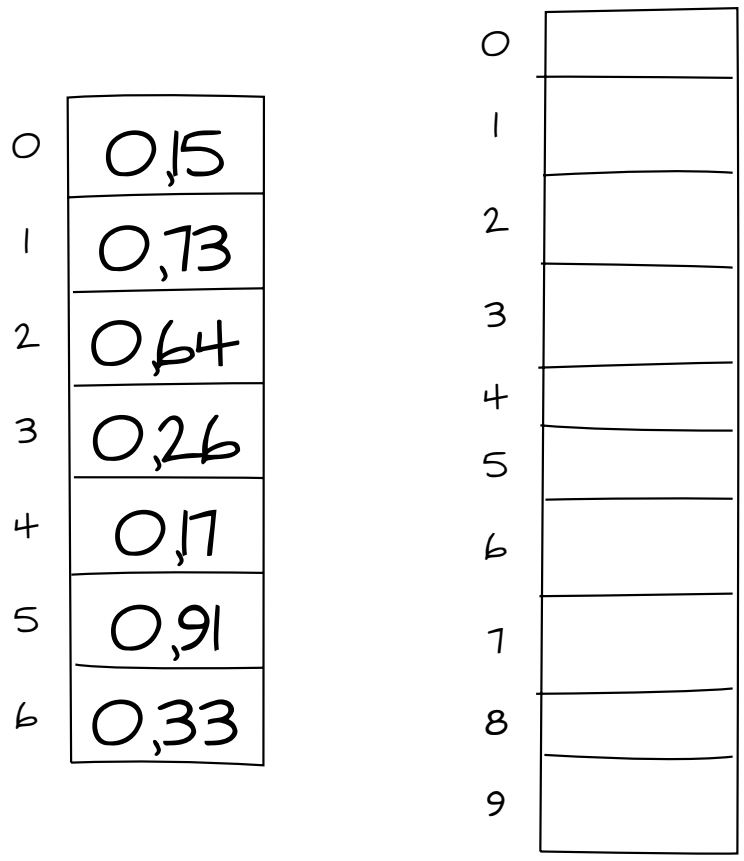
- Processo de ordenação dos dígitos



Espaço $\Theta(n+k)$ e tempo $\Theta(d \times (n+k))$

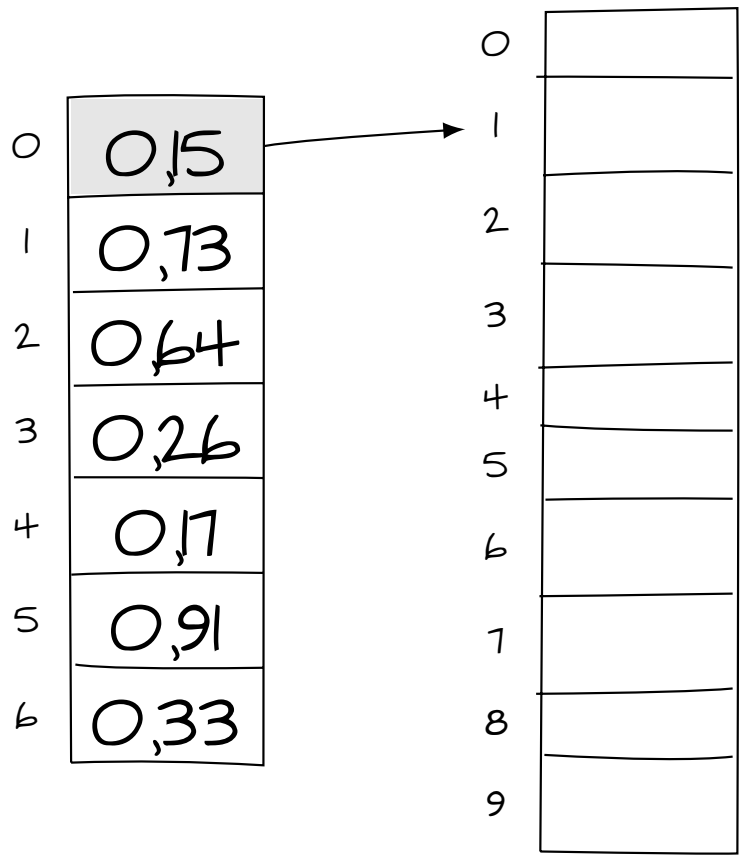
Bucket sort

- O dígito mais significativo é usado para indexação e, caso a posição já esteja em uso, é feita a ordenação por inserção (estável) em lista encadeada (*bucket*)



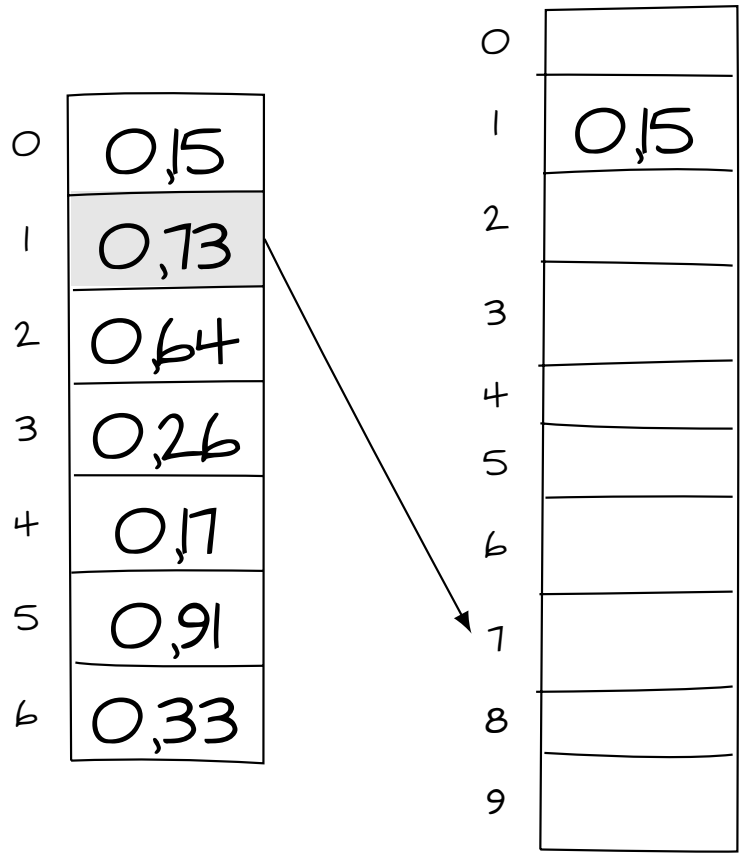
Bucket sort

- O dígito mais significativo é usado para indexação e, caso a posição já esteja em uso, é feita a ordenação por inserção (estável) em lista encadeada (*bucket*)



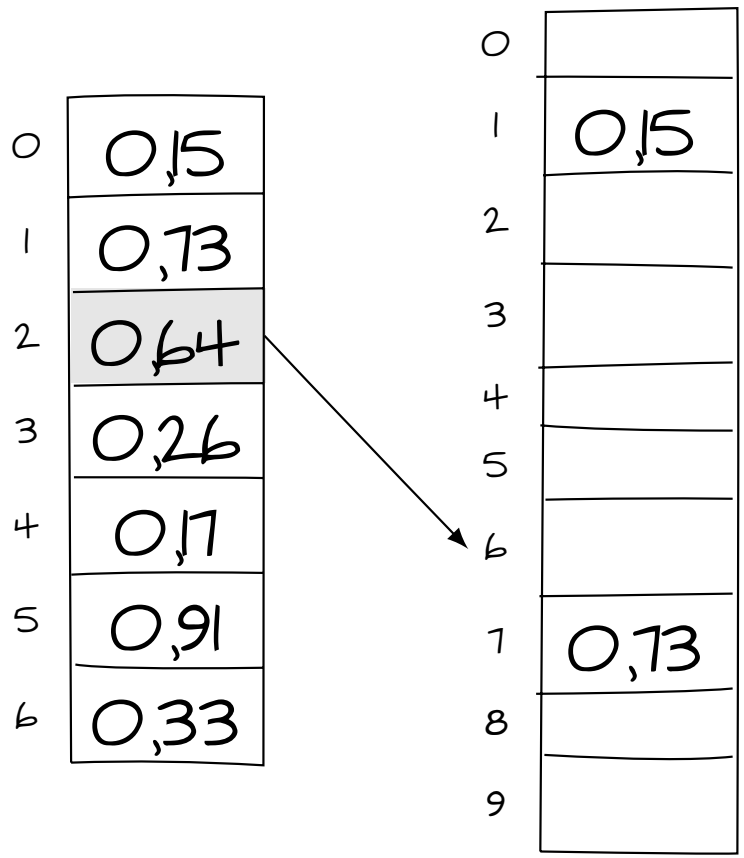
Bucket sort

- O dígito mais significativo é usado para indexação e, caso a posição já esteja em uso, é feita a ordenação por inserção (estável) em lista encadeada (*bucket*)



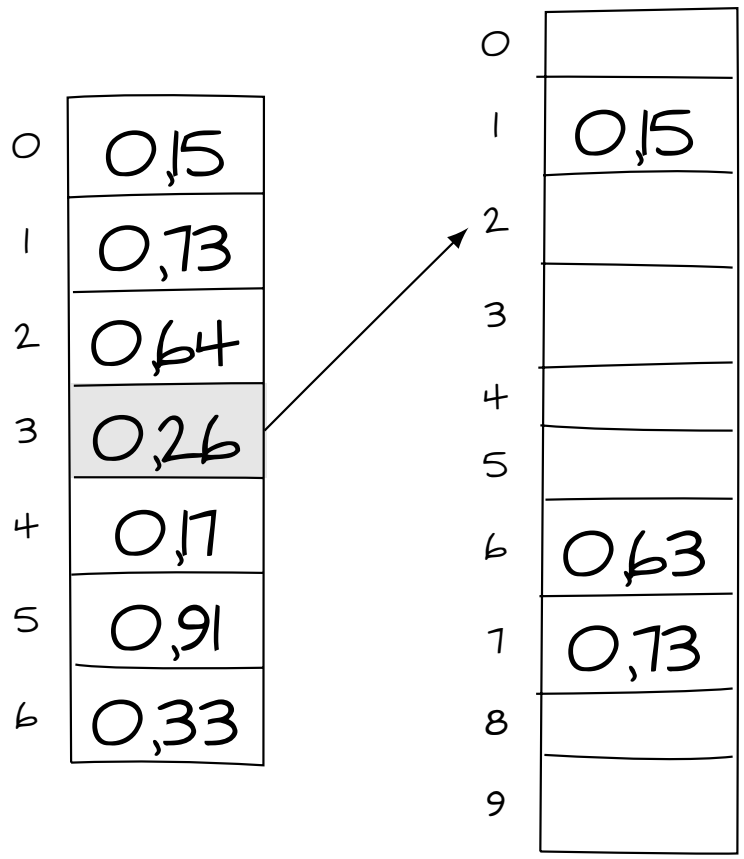
Bucket sort

- O dígito mais significativo é usado para indexação e, caso a posição já esteja em uso, é feita a ordenação por inserção (estável) em lista encadeada (*bucket*)



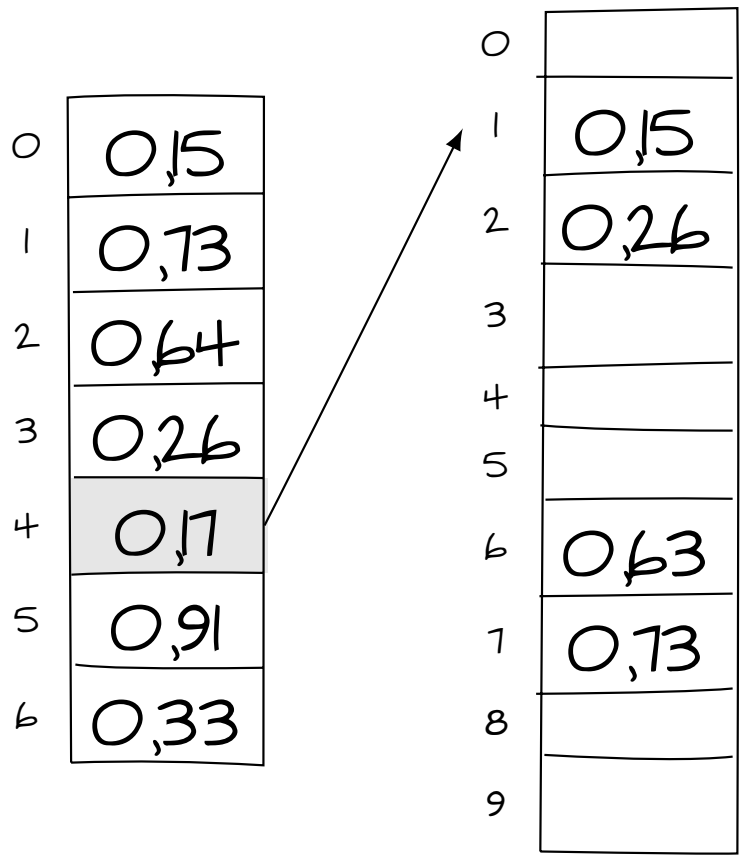
Bucket sort

- O dígito mais significativo é usado para indexação e, caso a posição já esteja em uso, é feita a ordenação por inserção (estável) em lista encadeada (*bucket*)



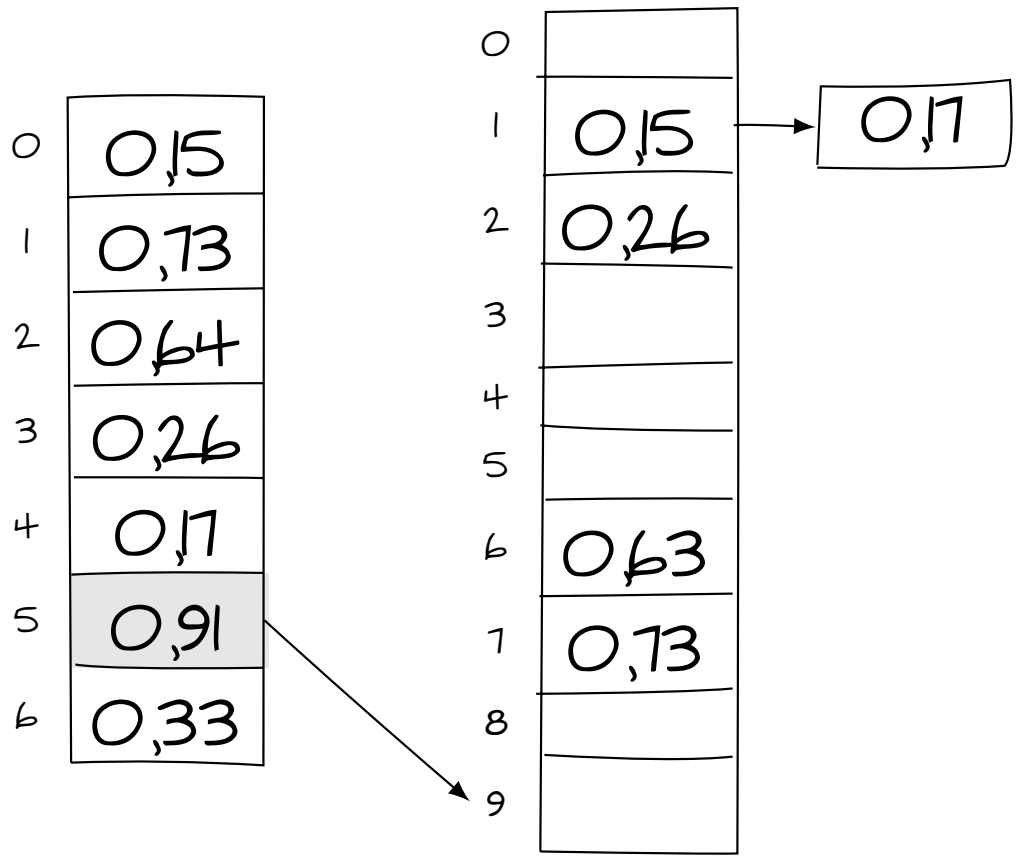
Bucket sort

- O dígito mais significativo é usado para indexação e, caso a posição já esteja em uso, é feita a ordenação por inserção (estável) em lista encadeada (*bucket*)



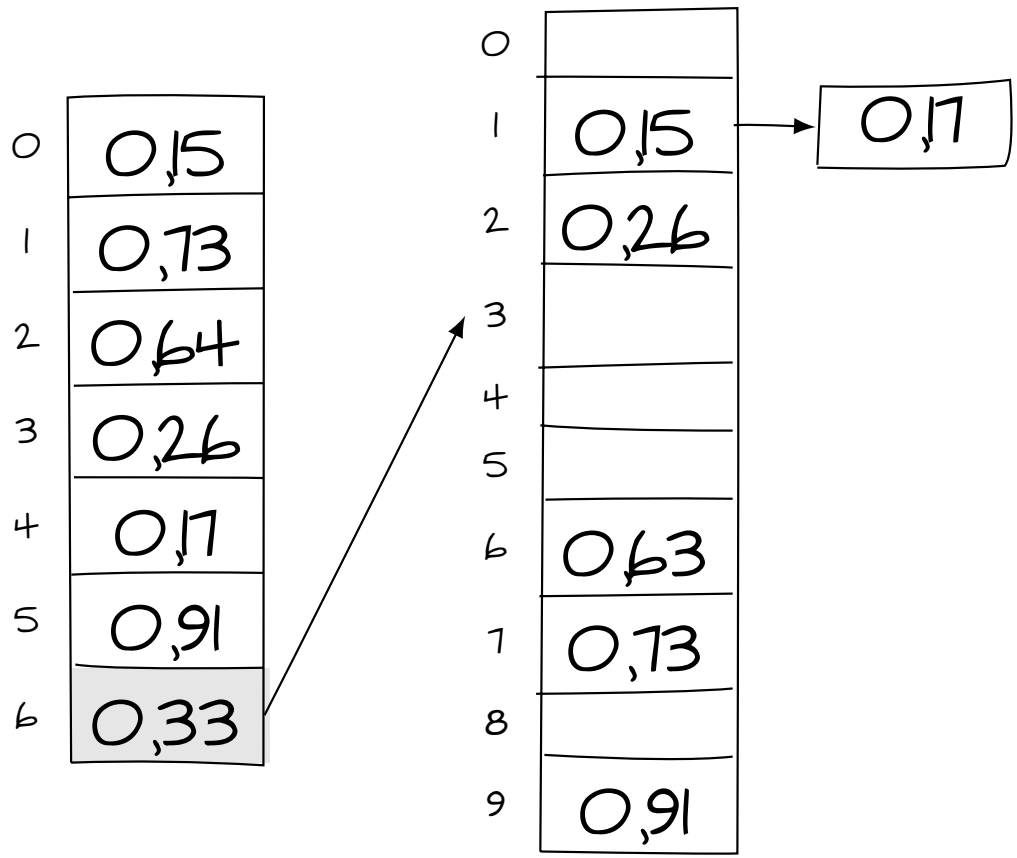
Bucket sort

- O dígito mais significativo é usado para indexação e, caso a posição já esteja em uso, é feita a ordenação por inserção (estável) em lista encadeada (*bucket*)



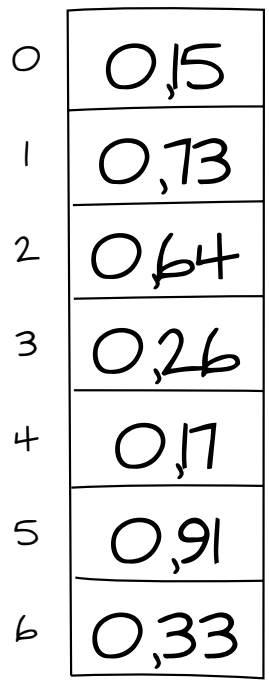
Bucket sort

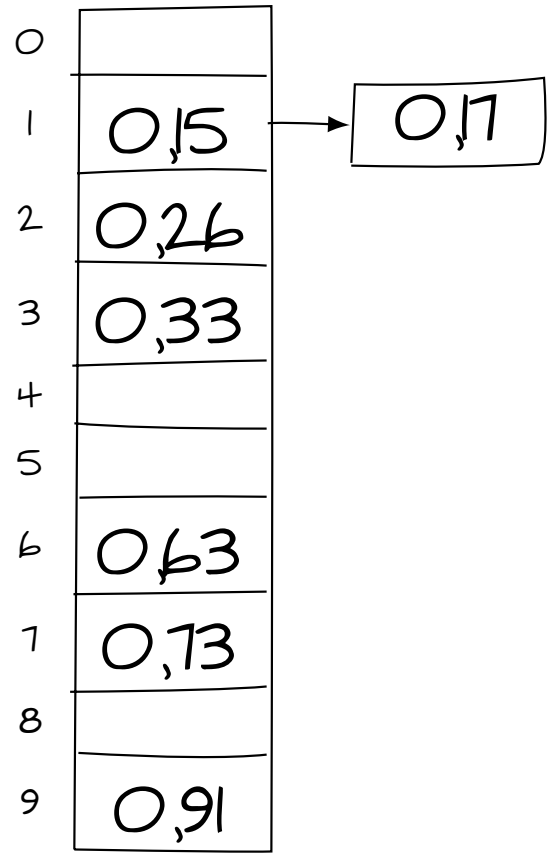
- O dígito mais significativo é usado para indexação e, caso a posição já esteja em uso, é feita a ordenação por inserção (estável) em lista encadeada (*bucket*)



Bucket sort

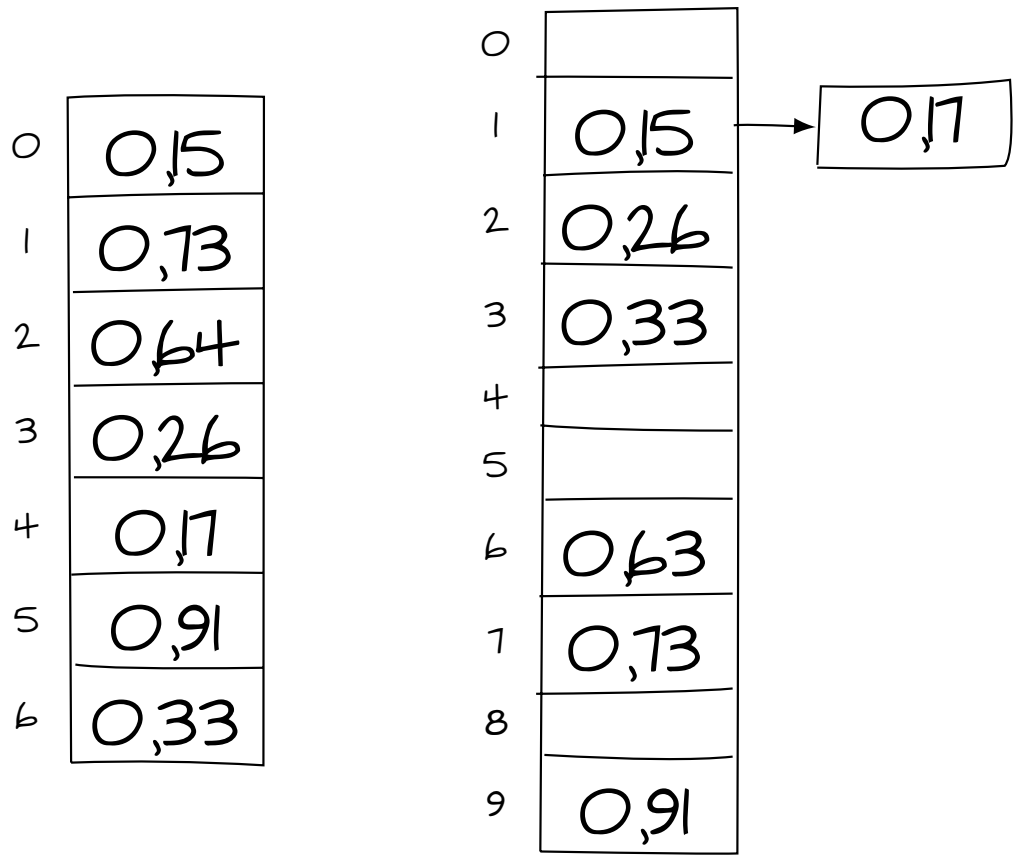
- O dígito mais significativo é usado para indexação e, caso a posição já esteja em uso, é feita a ordenação por inserção (estável) em lista encadeada (*bucket*)





Bucket sort

- O dígito mais significativo é usado para indexação e, caso a posição já esteja em uso, é feita a ordenação por inserção (estável) em lista encadeada (*bucket*)



Espaço $\Theta(n+k)$ e tempo $\Omega(n)$ e $O(n^2)$

Exemplo

- Considerando o algoritmo de ordenação Radix sort, implementado com ordenação linear pelo Counting sort, ordene o vetor 99, 342, 102, 33, 298 e 7
  - Utilize o critério decrescente de ordenação
  - Execute passo a passo cada etapa dos algoritmos