Ordenação linear

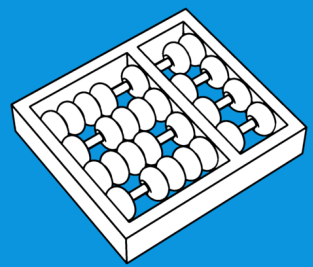
MC458 - Projeto e Análise de Algoritmos I

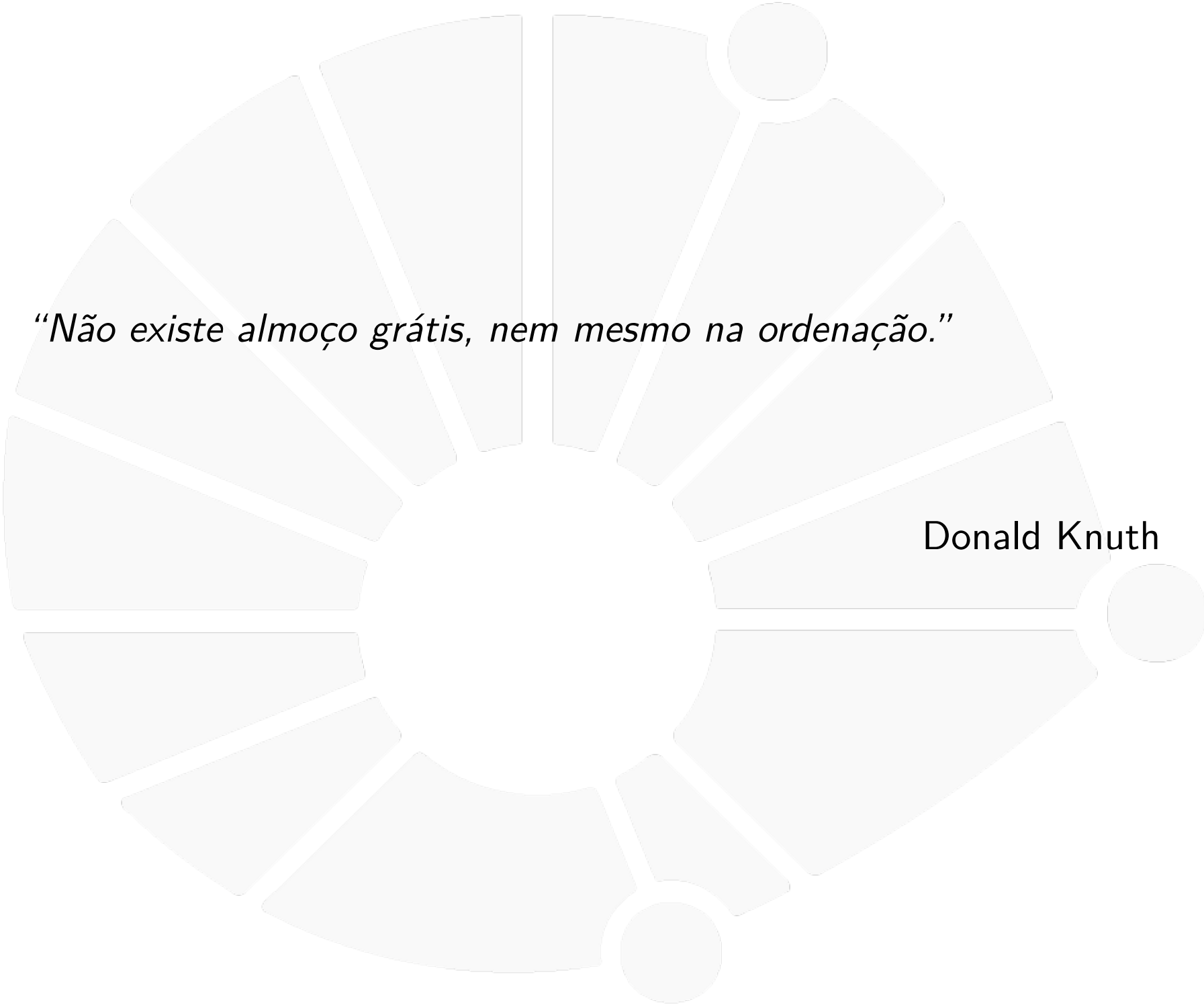
10/25

19

Santiago Valdés Ravelo
https://ic.unicamp.br/~santiago/
ravelo@unicamp.br

UNICAMP

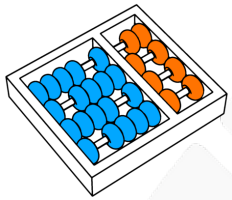




"Não existe almoço grátis, nem mesmo na ordenação."

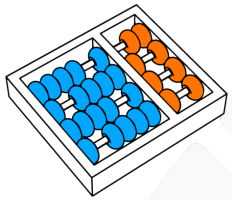
Donald Knuth

Introdução



Algoritmos lineares para ordenação

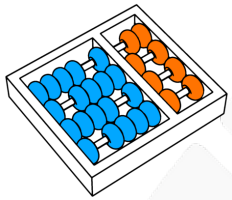
Estudaremos algoritmos de ordenação de **tempo linear**:



Algoritmos lineares para ordenação

Estudaremos algoritmos de ordenação de **tempo linear**:

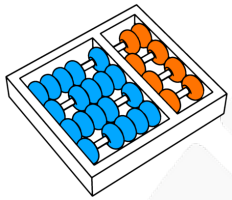
- Counting Sort:



Algoritmos lineares para ordenação

Estudaremos algoritmos de ordenação de **tempo linear**:

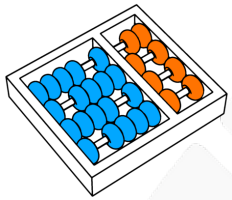
- Counting Sort:
  - Os elementos são inteiros pequenos.



Algoritmos lineares para ordenação

Estudaremos algoritmos de ordenação de **tempo linear**:

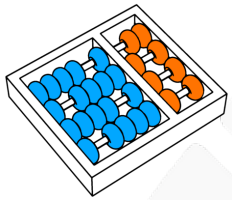
- Counting Sort:
  - Os elementos são inteiros pequenos.
  - Os valores são limitados por $O(n)$.



Algoritmos lineares para ordenação

Estudaremos algoritmos de ordenação de **tempo linear**:

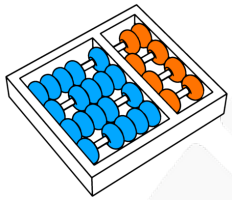
- Counting Sort:
  - Os elementos são inteiros pequenos.
  - Os valores são limitados por $O(n)$.
- Radix Sort:



Algoritmos lineares para ordenação

Estudaremos algoritmos de ordenação de **tempo linear**:

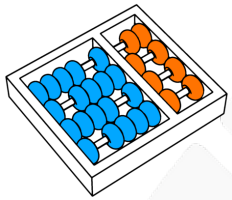
- Counting Sort:
  - Os elementos são inteiros pequenos.
  - Os valores são limitados por $O(n)$.
- Radix Sort:
  - Representação numérica com comprimento constante.



Algoritmos lineares para ordenação

Estudaremos algoritmos de ordenação de **tempo linear**:

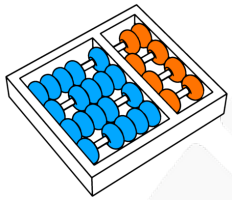
- Counting Sort:
  - Os elementos são inteiros pequenos.
  - Os valores são limitados por $O(n)$.
- Radix Sort:
  - Representação numérica com comprimento constante.
  - Os valores dos elementos independente de n.



Algoritmos lineares para ordenação

Estudaremos algoritmos de ordenação de **tempo linear**:

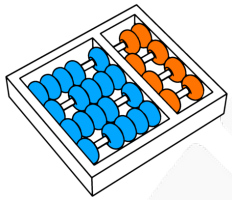
- Counting Sort:
  - Os elementos são inteiros pequenos.
  - Os valores são limitados por $O(n)$.
- Radix Sort:
  - Representação numérica com comprimento constante.
  - Os valores dos elementos independente de n.
- Bucket Sort:



Algoritmos lineares para ordenação

Estudaremos algoritmos de ordenação de **tempo linear**:

- Counting Sort:
  - Os elementos são inteiros pequenos.
  - Os valores são limitados por $O(n)$.
- Radix Sort:
  - Representação numérica com comprimento constante.
  - Os valores dos elementos independente de n.
- Bucket Sort:
  - Elementos do vetor são sorteados no intervalo [0..1).

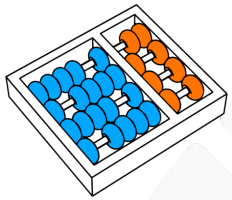


Algoritmos lineares para ordenação

Estudaremos algoritmos de ordenação de **tempo linear**:

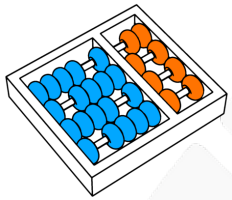
- Counting Sort:
  - Os elementos são inteiros pequenos.
  - Os valores são limitados por $O(n)$.
- Radix Sort:
  - Representação numérica com comprimento constante.
  - Os valores dos elementos independente de n.
- Bucket Sort:
  - Elementos do vetor são sorteados no intervalo $[0..1)$.
  - Os valores são distribuídos uniformemente.

Counting sort



Counting Sort

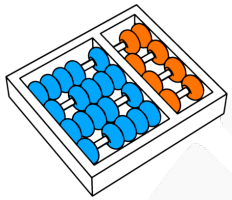
Ideia:



Counting Sort

Ideia:

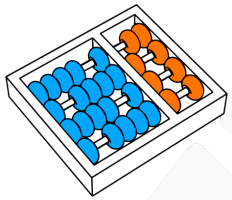
- Entrada é um vetor $A[1 \ldots n]$ de inteiros.



Counting Sort

Ideia:

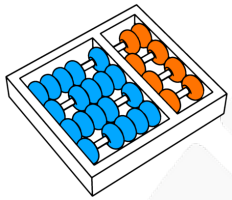
- Entrada é um vetor $A[1 \ldots n]$ de inteiros.
- Saída será outro vetor $B[1 \ldots n]$ de inteiros.



Counting Sort

Ideia:

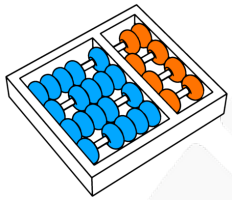
- Entrada é um vetor $A[1 \ldots n]$ de inteiros.
- Saída será outro vetor $B[1 \ldots n]$ de inteiros.
- Supomos que o valores estão no intervalo entre 0 e k.



Counting Sort

Ideia:

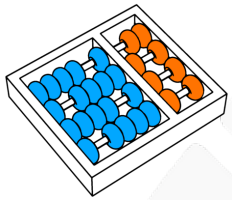
- Entrada é um vetor $A[1 \ldots n]$ de inteiros.
- Saída será outro vetor $B[1 \ldots n]$ de inteiros.
- Supomos que o valores estão no intervalo entre 0 e k.
- Para cada inteiro i no vetor, **contamos** o número $C[i]$ de elementos menores ou iguais i em A.



Counting Sort

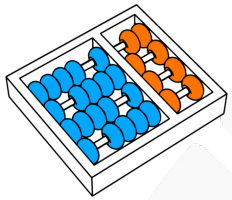
Ideia:

- Entrada é um vetor $A[1 \ldots n]$ de inteiros.
- Saída será outro vetor $B[1 \ldots n]$ de inteiros.
- Supomos que o valores estão no intervalo entre 0 e k.
- Para cada inteiro i no vetor, **contamos** o número $C[i]$ de elementos menores ou iguais i em A.
- Então, na posição $C[i]$ do vetor B, deve haver o elemento i.



Counting Sort - Exemplo

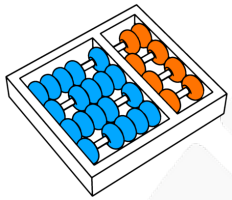
A	3	6	4	1	3	4	1	4
	1	2	3	4	5	6	7	8



Counting Sort - Exemplo

A	3	6	4	1	3	4	1	4
	1	2	3	4	5	6	7	8

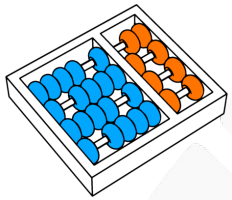
C	0	0	0	0	0	0
	1	2	3	4	5	6



Counting Sort - Exemplo

A	3	6	4	1	3	4	1	4
	1	2	3	4	5	6	7	8

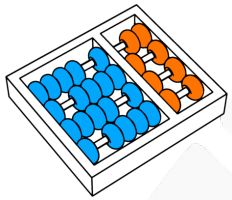
C	2	0	2	3	0	1
	1	2	3	4	5	6



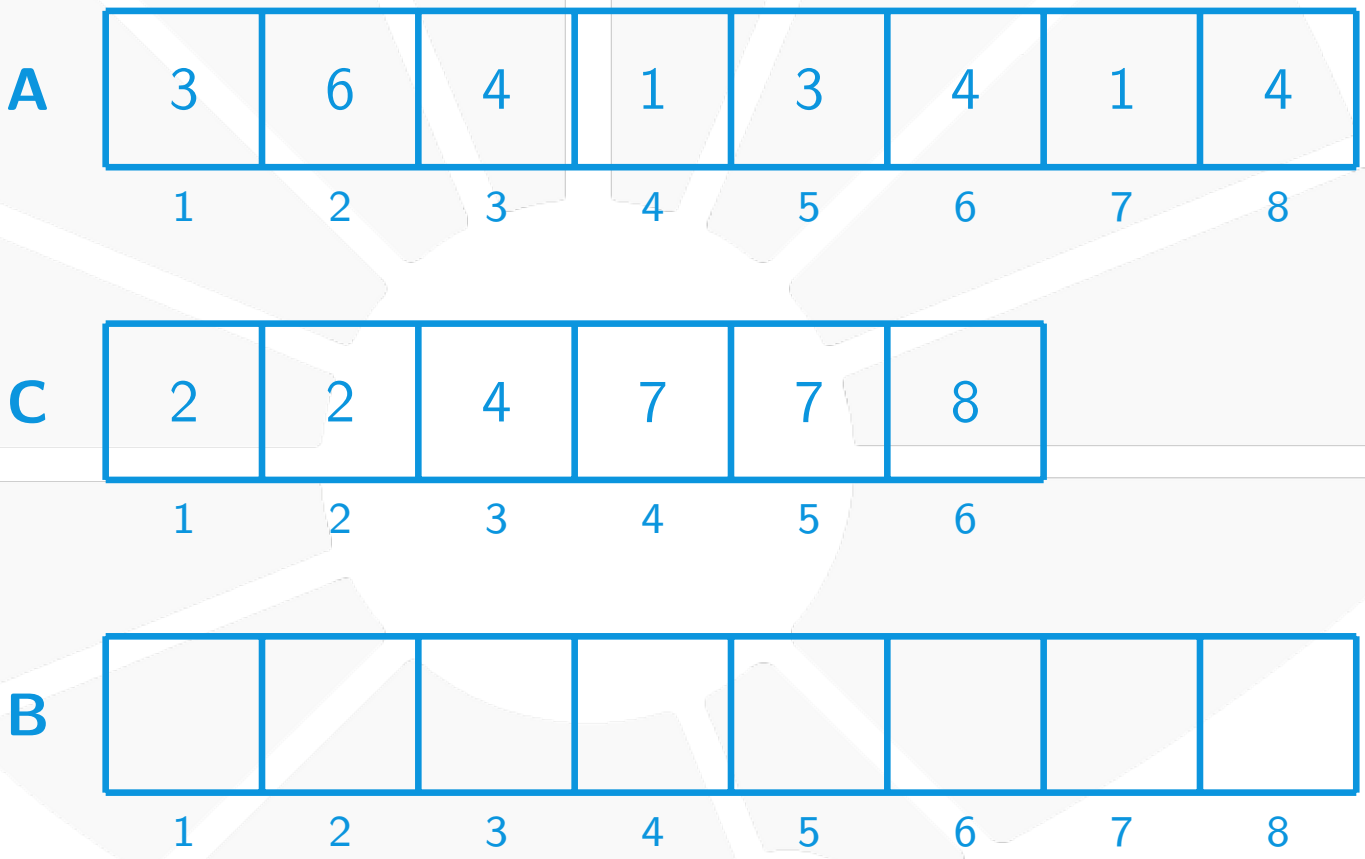
Counting Sort - Exemplo

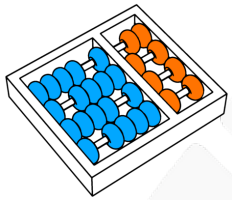
A	3	6	4	1	3	4	1	4
	1	2	3	4	5	6	7	8

C	2	2	4	7	7	8
	1	2	3	4	5	6

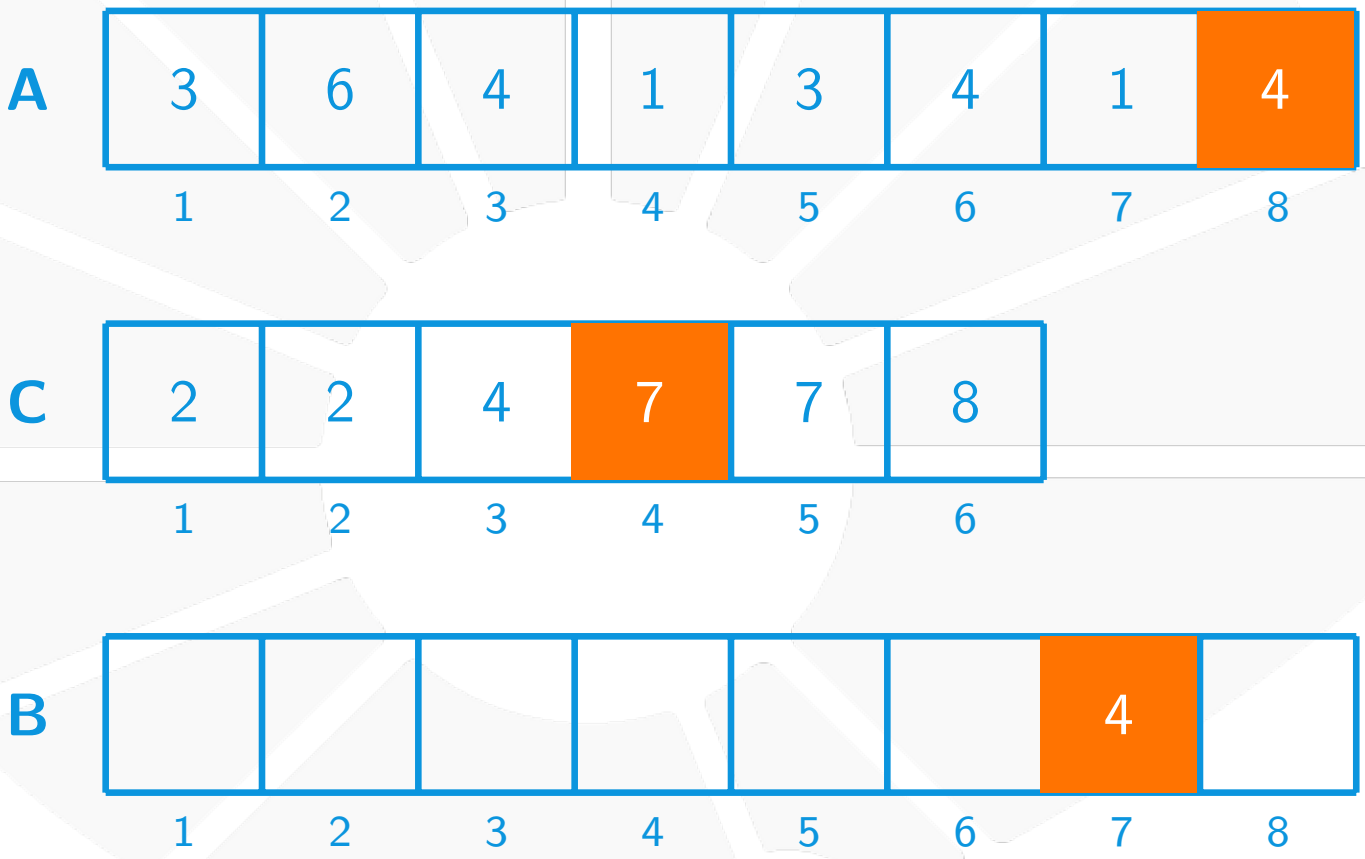


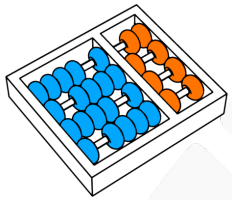
Counting Sort - Exemplo



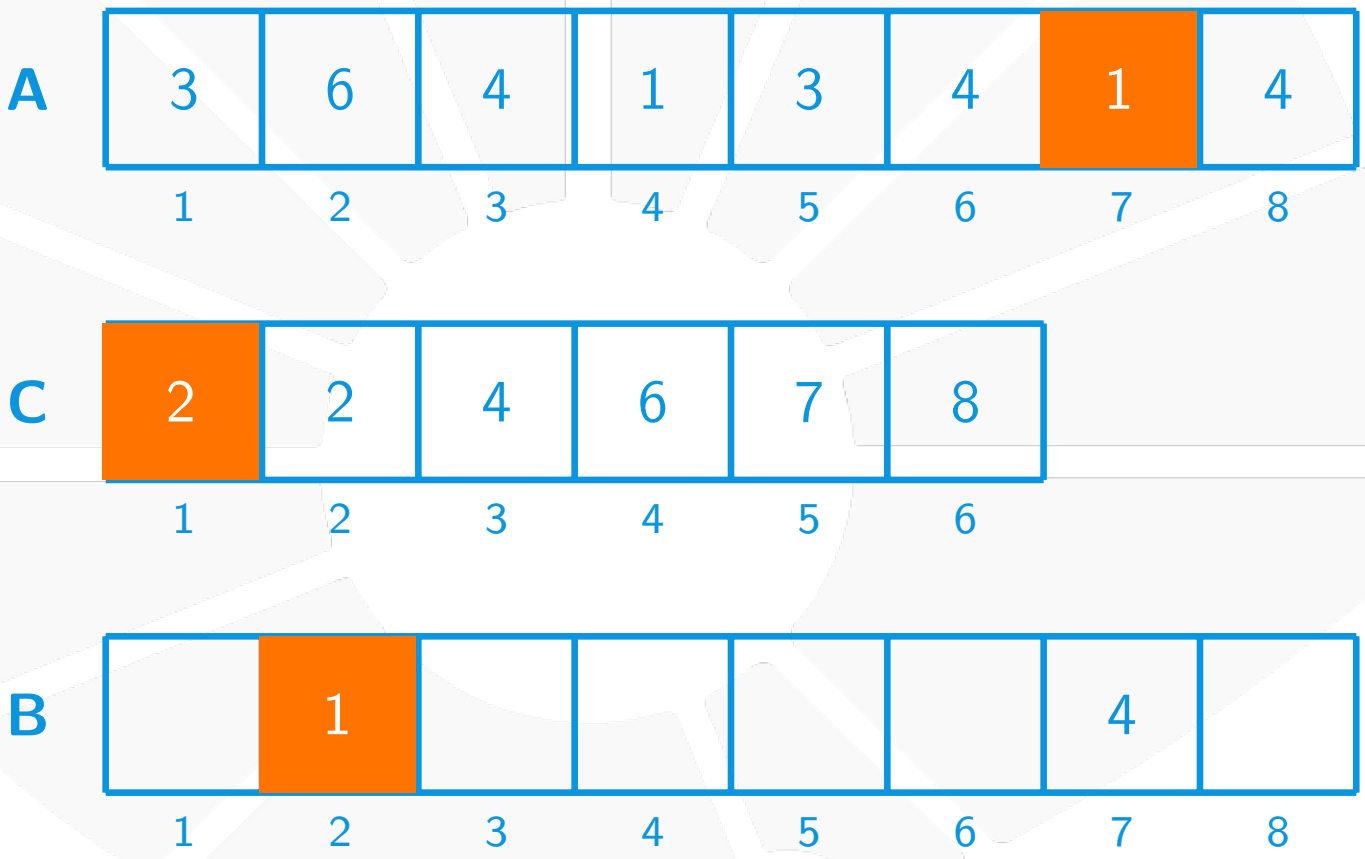


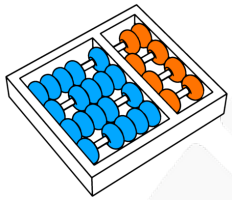
Counting Sort - Exemplo



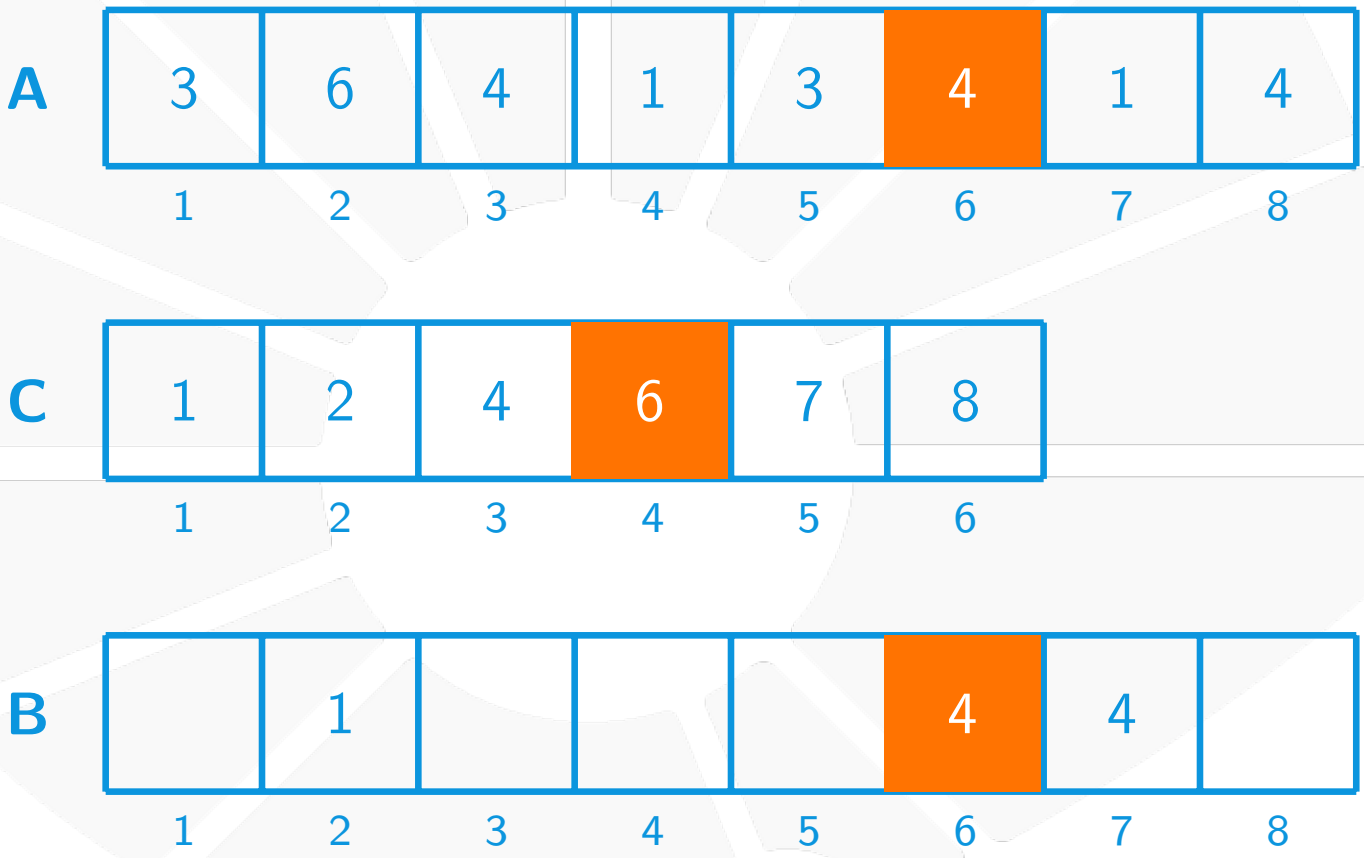


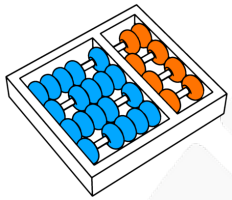
Counting Sort - Exemplo





Counting Sort - Exemplo



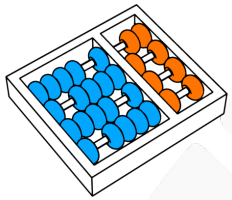


Counting Sort - Exemplo

A	3	6	4	1	3	4	1	4
	1	2	3	4	5	6	7	8

C	0	2	2	4	7	7
	1	2	3	4	5	6

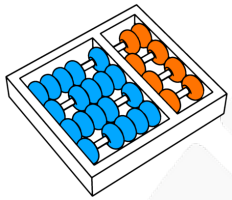
B	1	1	3	3	4	4	4	6
	1	2	3	4	5	6	7	8



Counting Sort - Algoritmo

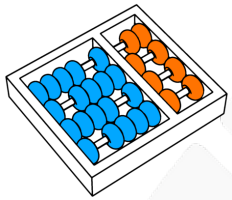
Algoritmo: COUNTING-SORT(A, B, n, k)

```
1 para i ← 0 até k
2     C[i] ← 0
3 para j ← 1 até n
4     C[A[j]] ← C[A[j]] + 1
5 para i ← 1 até k
6     C[i] ← C[i] + C[i − 1]
7 para j ← n até 1
8     B[C[A[j]]] ← A[j]
9     C[A[j]] ← C[A[j]] − 1
```



Counting Sort - Complexidade

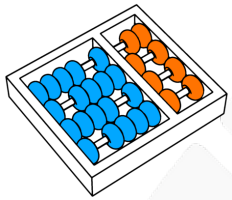
Qual a complexidade de COUNTING-SORT?



Counting Sort - Complexidade

Qual a complexidade de COUNTING-SORT?

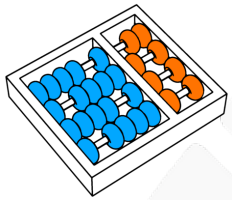
- COUNTING-SORT realiza $O(n+k)$ instruções elementares.



Counting Sort - Complexidade

Qual a complexidade de COUNTING-SORT?

- COUNTING-SORT realiza $O(n+k)$ instruções elementares.
- Quando $k \in O(n)$, ele tem complexidade $O(n)$.

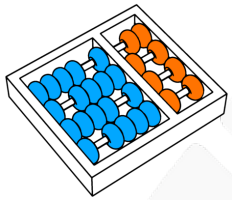


Counting Sort - Complexidade

Qual a complexidade de COUNTING-SORT?

- COUNTING-SORT realiza $O(n+k)$ instruções elementares.
- Quando $k \in O(n)$, ele tem complexidade $O(n)$.

E a cota inferior de $\Omega(n \log n)$ para ordenação?



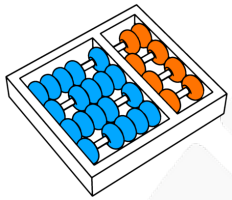
Counting Sort - Complexidade

Qual a complexidade de COUNTING-SORT?

- COUNTING-SORT realiza $O(n+k)$ instruções elementares.
- Quando $k \in O(n)$, ele tem complexidade $O(n)$.

E a cota inferior de $\Omega(n \log n)$ para ordenação?

- **NÃO** há comparações entre elementos de A.



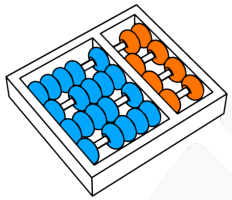
Counting Sort - Complexidade

Qual a complexidade de COUNTING-SORT?

- COUNTING-SORT realiza $O(n+k)$ instruções elementares.
- Quando $k \in O(n)$, ele tem complexidade $O(n)$.

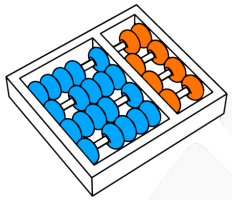
E a cota inferior de $\Omega(n \log n)$ para ordenação?

- **NÃO** há comparações entre elementos de A.
- A cota só vale para algoritmos baseados em comparação.



Algoritmos in-place e estáveis

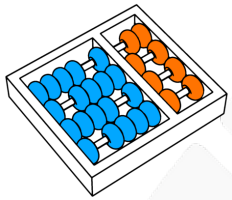
Algoritmos de ordenação in-place:



Algoritmos in-place e estáveis

Algoritmos de ordenação in-place:

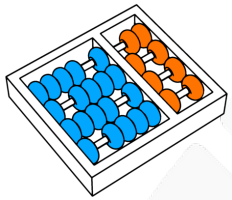
- Um algoritmo é **in-place** se a quantidade de memória adicional requerida independe de n.



Algoritmos in-place e estáveis

Algoritmos de ordenação in-place:

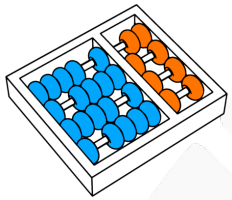
- Um algoritmo é **in-place** se a quantidade de memória adicional requerida independe de n.
- São in-place Insertion-Sort, Quick-Sort, Heap-Sort.



Algoritmos in-place e estáveis

Algoritmos de ordenação in-place:

- Um algoritmo é **in-place** se a quantidade de memória adicional requerida independe de n.
- São in-place Insertion-Sort, Quick-Sort, Heap-Sort.
- Não são in-place Merge-Sort, Counting-Sort.

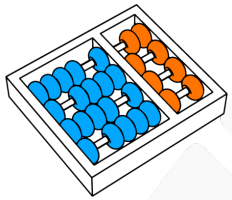


Algoritmos in-place e estáveis

Algoritmos de ordenação in-place:

- Um algoritmo é **in-place** se a quantidade de memória adicional requerida independe de n.
- São in-place Insertion-Sort, Quick-Sort, Heap-Sort.
- Não são in-place Merge-Sort, Counting-Sort.

Algoritmos de ordenação estáveis:



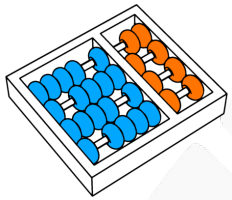
Algoritmos in-place e estáveis

Algoritmos de ordenação in-place:

- Um algoritmo é **in-place** se a quantidade de memória adicional requerida independe de n.
- São in-place Insertion-Sort, Quick-Sort, Heap-Sort.
- Não são in-place Merge-Sort, Counting-Sort.

Algoritmos de ordenação estáveis:

- Um algoritmo é **estável** se elementos com chaves iguais mantêm-se na ordem passada na entrada.



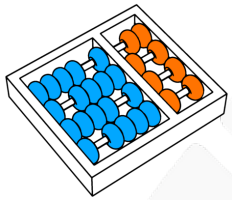
Algoritmos in-place e estáveis

Algoritmos de ordenação in-place:

- Um algoritmo é **in-place** se a quantidade de memória adicional requerida independe de n.
- São in-place Insertion-Sort, Quick-Sort, Heap-Sort.
- Não são in-place Merge-Sort, Counting-Sort.

Algoritmos de ordenação estáveis:

- Um algoritmo é **estável** se elementos com chaves iguais mantêm-se na ordem passada na entrada.
- São estáveis Insertion-Sort, Merge-Sort, Counting-Sort.



Algoritmos in-place e estáveis

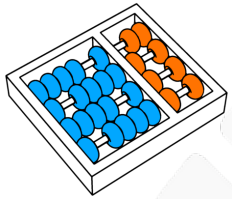
Algoritmos de ordenação in-place:

- Um algoritmo é **in-place** se a quantidade de memória adicional requerida independe de n.
- São in-place Insertion-Sort, Quick-Sort, Heap-Sort.
- Não são in-place Merge-Sort, Counting-Sort.

Algoritmos de ordenação estáveis:

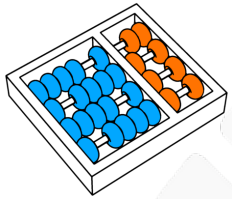
- Um algoritmo é **estável** se elementos com chaves iguais mantêm-se na ordem passada na entrada.
- São estáveis Insertion-Sort, Merge-Sort, Counting-Sort.
- O Heap-Sort e o Quick-Sort não são estáveis.

Radix sort



Radix Sort

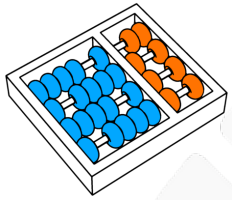
Ideia:



Radix Sort

Ideia:

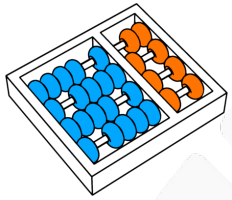
- A entrada é um vetor $A[1 \ldots n]$ de inteiros.



Radix Sort

Ideia:

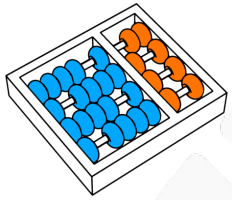
- A entrada é um vetor $A[1 \ldots n]$ de inteiros.
- Cada inteiro é representado na **base** k-ária



Radix Sort

Ideia:

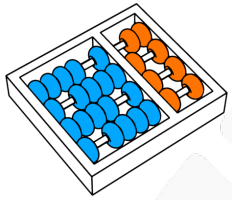
- A entrada é um vetor $A[1 \ldots n]$ de inteiros.
- Cada inteiro é representado na **base** k-ária
- Supomos que um inteiro tem d dígitos.



Radix Sort

Ideia:

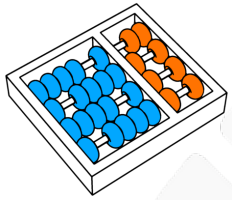
- A entrada é um vetor $A[1 \ldots n]$ de inteiros.
- Cada inteiro é representado na **base** k-ária
- Supomos que um inteiro tem d dígitos.
- Por exemplo, CEPs são inteiros de 8 dígitos na base 10.



Radix Sort

Ideia:

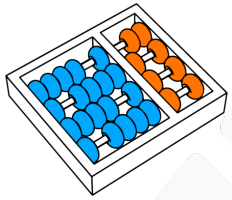
- A entrada é um vetor $A[1 \ldots n]$ de inteiros.
- Cada inteiro é representado na **base** k-ária
- Supomos que um inteiro tem d dígitos.
- Por exemplo, CEPs são inteiros de 8 dígitos na base 10.
- Ordenaremos **um dígito por vez** com um algoritmo estável.



Radix Sort

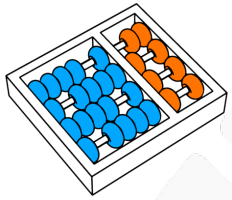
Ideia:

- A entrada é um vetor $A[1 \ldots n]$ de inteiros.
- Cada inteiro é representado na **base** k-ária
- Supomos que um inteiro tem d dígitos.
- Por exemplo, CEPs são inteiros de 8 dígitos na base 10.
- Ordenaremos **um dígito por vez** com um algoritmo estável.
- Ordenamos primeiro os dígitos menos significativos.



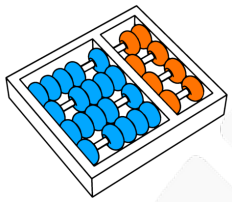
Radix Sort - Exemplo

329
457
657
839
436
720
355



Radix Sort - Exemplo

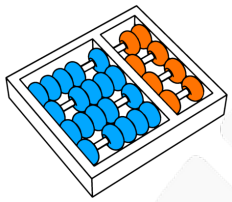
329		720		
457		355		
657		436		
839		457		
436	$\rightarrow$	657	$\rightarrow$	
720		329		
355		839		
		$\uparrow$		



Radix Sort - Exemplo

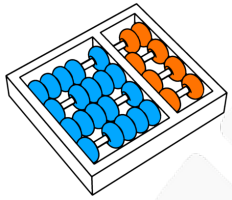
329	720	720	
457	355	329	
657	436	436	
839	457	839	
436	657	355	
720	329	457	
355	839	657	
	↑	↑	

$\rightarrow$ $\rightarrow$ $\rightarrow$



Radix Sort - Exemplo

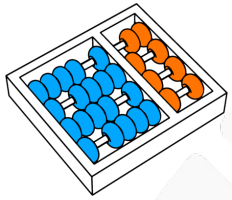
329		720		720		329
457		355		329		355
657		436		436		436
839		457		839		457
436	$\rightarrow$	657	$\rightarrow$	355	$\rightarrow$	657
720		329		457		720
355		839		657		839
		$\uparrow$		$\uparrow$		$\uparrow$



Radix Sort - Algoritmo

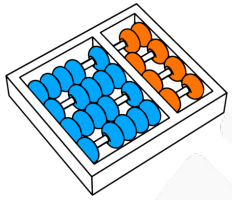
Algoritmo: RADIX-SORT(A, n, d)

```
1 para i ← 1 até d
2     Ordene A[1 . . . n] pelo i-ésimo dígito usando um método
        estável
```



Radix Sort - Correção

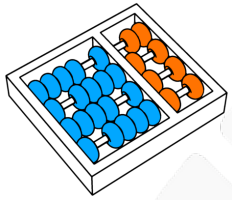
Demonstramos a correção do algoritmo por indução:



Radix Sort - Correção

Demonstramos a correção do algoritmo por indução:

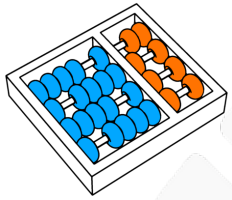
- Suponha que o vetor de números esteja ordenado em relação aos $i - 1$ dígitos menos significativos.



Radix Sort - Correção

Demonstramos a correção do algoritmo por indução:

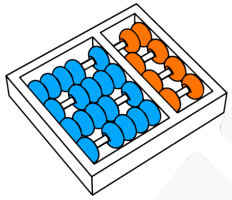
- Suponha que o vetor de números esteja ordenado em relação aos $i - 1$ dígitos menos significativos.
- Ordene o vetor pelo i-ésimo dígito com um método estável.



Radix Sort - Correção

Demonstramos a correção do algoritmo por indução:

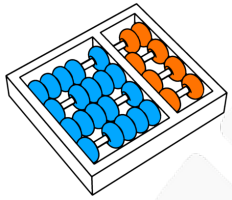
- Suponha que o vetor de números esteja ordenado em relação aos $i - 1$ dígitos menos significativos.
- Ordene o vetor pelo i-ésimo dígito com um método estável.
- Considere números a e b no vetor resultante com $a < b$:



Radix Sort - Correção

Demonstramos a correção do algoritmo por indução:

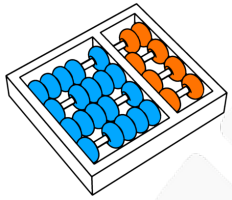
- Suponha que o vetor de números esteja ordenado em relação aos $i - 1$ dígitos menos significativos.
- Ordene o vetor pelo i-ésimo dígito com um método estável.
- Considere números a e b no vetor resultante com $a < b$:
  1. Se os i-ésimos dígitos forem distintos, então eles estão em ordem em relação a este dígito, independentemente dos $i - 1$ primeiros dígitos. Portanto, a aparece antes de b.



Radix Sort - Correção

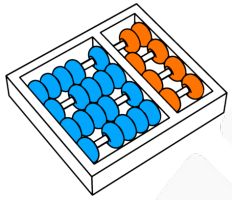
Demonstramos a correção do algoritmo por indução:

- Suponha que o vetor de números esteja ordenado em relação aos $i-1$ dígitos menos significativos.
- Ordene o vetor pelo i-ésimo dígito com um método estável.
- Considere números a e b no vetor resultante com $a < b$:
  1. Se os i-ésimos dígitos forem distintos, então eles estão em ordem em relação a este dígito, independentemente dos $i-1$ primeiros dígitos. Portanto, a aparece antes de b.
  2. Se os i-ésimos dígitos forem iguais, então por hipótese de indução, eles estavam em ordem pelos $i-1$ primeiros dígitos antes da i-ésima iteração. Como usamos um algoritmo estável nesta iteração, a ordem é mantida de forma que a aparecia antes de b e continua assim.



Radix Sort - Complexidade

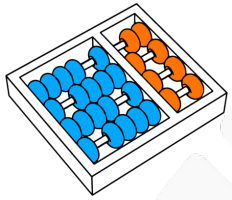
Qual é a complexidade do RADIX-SORT?



Radix Sort - Complexidade

Qual é a complexidade do RADIX-SORT?

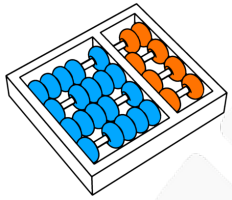
- O algoritmo de ordenação usado tem tempo $\Theta(f(n))$.



Radix Sort - Complexidade

Qual é a complexidade do RADIX-SORT?

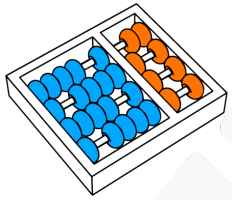
- O algoritmo de ordenação usado tem tempo $\Theta(f(n))$.
- Então RADIX-SORT tem tempo total $\Theta(d\,f(n))$.



Radix Sort - Complexidade

Qual é a complexidade do RADIX-SORT?

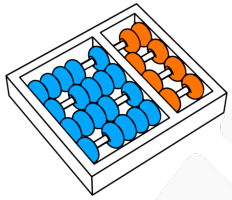
- O algoritmo de ordenação usado tem tempo $\Theta(f(n))$.
- Então RADIX-SORT tem tempo total $\Theta(d\,f(n))$.
- Quando d é constante, a complexidade é $\Theta(f(n))$.



Radix Sort - Complexidade

Qual é a complexidade do RADIX-SORT?

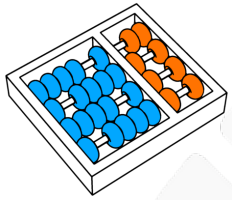
- O algoritmo de ordenação usado tem tempo $\Theta(f(n))$.
- Então RADIX-SORT tem tempo total $\Theta(d\, f(n))$.
- Quando d é constante, a complexidade é $\Theta(f(n))$.
- Se usarmos COUNTING-SORT, a complexidade é $\Theta(n + k)$.



Radix Sort - Complexidade

Qual é a complexidade do RADIX-SORT?

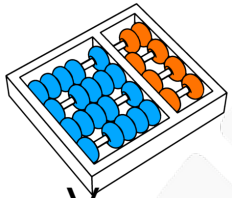
- O algoritmo de ordenação usado tem tempo $\Theta(f(n))$.
- Então RADIX-SORT tem tempo total $\Theta(d\, f(n))$.
- Quando d é constante, a complexidade é $\Theta(f(n))$.
- Se usarmos COUNTING-SORT, a complexidade é $\Theta(n + k)$.
  - Lembre-se de que $k - 1$ é o maior valor do número na entrada do COUNTING-SORT (e.g., $k = 9$ na base decimal).



Radix Sort - Complexidade

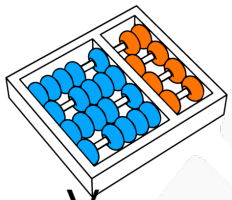
Qual é a complexidade do RADIX-SORT?

- O algoritmo de ordenação usado tem tempo $\Theta(f(n))$.
- Então RADIX-SORT tem tempo total $\Theta(d\, f(n))$.
- Quando d é constante, a complexidade é $\Theta(f(n))$.
- Se usarmos COUNTING-SORT, a complexidade é $\Theta(n + k)$.
  - Lembre-se de que $k - 1$ é o maior valor do número na entrada do COUNTING-SORT (e.g., $k = 9$ na base decimal).
  - Se k é constante, então o tempo resultante é $\Theta(n)$.



Radix Sort - Comparação

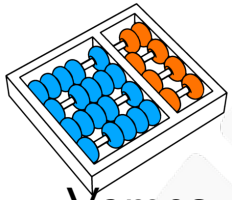
Vamos comparar RADIX-SORT e MERGE-SORT:



Radix Sort - Comparação

Vamos comparar RADIX-SORT e MERGE-SORT:

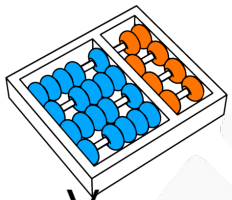
- Temos $n = 2^{20}$ números de 64 bits.



Radix Sort - Comparação

Vamos comparar RADIX-SORT e MERGE-SORT:

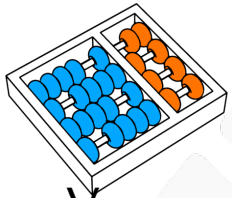
- Temos $n = 2^{20}$ números de 64 bits.
- Interpretamos os números na base $k = 2^{16}$ (cada 16 bits corresponde a um dígito).



Radix Sort - Comparação

Vamos comparar RADIX-SORT e MERGE-SORT:

- Temos $n = 2^{20}$ números de 64 bits.
- Interpretamos os números na base $k = 2^{16}$ (cada 16 bits corresponde a um dígito).
- Cada número tem $d = 4$ dígitos.

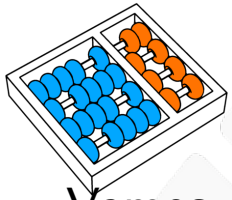


Radix Sort - Comparação

Vamos comparar RADIX-SORT e MERGE-SORT:

- Temos $n = 2^{20}$ números de 64 bits.
- Interpretamos os números na base $k = 2^{16}$ (cada 16 bits corresponde a um dígito).
- Cada número tem $d = 4$ dígitos.

1. Ordenando com MERGE-SORT:

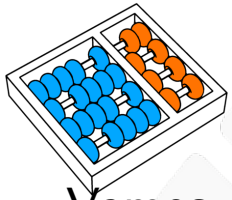


Radix Sort - Comparação

Vamos comparar RADIX-SORT e MERGE-SORT:

- Temos $n = 2^{20}$ números de 64 bits.
- Interpretamos os números na base $k = 2^{16}$ (cada 16 bits corresponde a um dígito).
- Cada número tem $d = 4$ dígitos.

1. Ordenando com MERGE-SORT:
   - Executamos cerca de $n \log_2 n = 20 \times 2^{20}$ **comparações**.

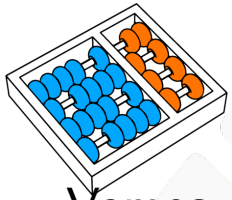


Radix Sort - Comparação

Vamos comparar RADIX-SORT e MERGE-SORT:

- Temos $n = 2^{20}$ números de 64 bits.
- Interpretamos os números na base $k = 2^{16}$ (cada 16 bits corresponde a um dígito).
- Cada número tem $d = 4$ dígitos.

1. Ordenando com MERGE-SORT:
   - Executamos cerca de $n \log_2 n = \mathbf{20 \times 2^{20}}$ **comparações**.
   - Usamos um vetor auxiliar de tamanho 2^{20}.

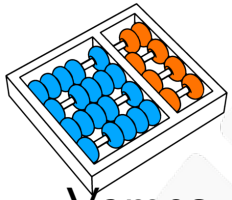


Radix Sort - Comparação

Vamos comparar RADIX-SORT e MERGE-SORT:

- Temos $n = 2^{20}$ números de 64 bits.
- Interpretamos os números na base $k = 2^{16}$ (cada 16 bits corresponde a um dígito).
- Cada número tem $d = 4$ dígitos.

1. Ordenando com MERGE-SORT:
   - Executamos cerca de $n \log_2 n = \mathbf{20 \times 2^{20}}$ **comparações**.
   - Usamos um vetor auxiliar de tamanho 2^{20}.
2. Ordenando com RADIX-SORT:

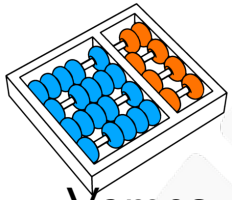


Radix Sort - Comparação

Vamos comparar RADIX-SORT e MERGE-SORT:

- Temos $n = 2^{20}$ números de 64 bits.
- Interpretamos os números na base $k = 2^{16}$ (cada 16 bits corresponde a um dígito).
- Cada número tem $d = 4$ dígitos.

1. Ordenando com MERGE-SORT:
   - Executamos cerca de $n \log_2 n = \mathbf{20 \times 2^{20}}$ **comparações**.
   - Usamos um vetor auxiliar de tamanho 2^{20}.
2. Ordenando com RADIX-SORT:
   - Utilizamos COUNTING-SORT como sub-rotina de ordenação.

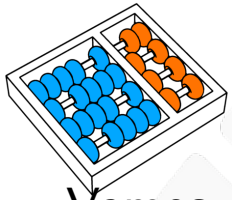


Radix Sort - Comparação

Vamos comparar RADIX-SORT e MERGE-SORT:

- Temos $n = 2^{20}$ números de 64 bits.
- Interpretamos os números na base $k = 2^{16}$ (cada 16 bits corresponde a um dígito).
- Cada número tem $d = 4$ dígitos.

1. Ordenando com MERGE-SORT:
   - Executamos cerca de $n \log_2 n = \mathbf{20 \times 2^{20}}$ **comparações**.
   - Usamos um vetor auxiliar de tamanho 2^{20}.
2. Ordenando com RADIX-SORT:
   - Utilizamos COUNTING-SORT como sub-rotina de ordenação.
   - Executamos cerca de $d(n + k) = \mathbf{4(2^{20} + 2^{16})}$ **operações**.

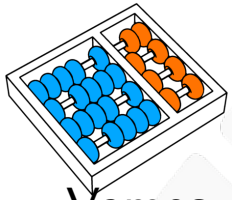


Radix Sort - Comparação

Vamos comparar RADIX-SORT e MERGE-SORT:

- Temos $n = 2^{20}$ números de 64 bits.
- Interpretamos os números na base $k = 2^{16}$ (cada 16 bits corresponde a um dígito).
- Cada número tem $d = 4$ dígitos.

1. Ordenando com MERGE-SORT:
   - Executamos cerca de $n \log_2 n = \mathbf{20 \times 2^{20}}$ **comparações**.
   - Usamos um vetor auxiliar de tamanho 2^{20}.
2. Ordenando com RADIX-SORT:
   - Utilizamos COUNTING-SORT como sub-rotina de ordenação.
   - Executamos cerca de $d(n + k) = \mathbf{4(2^{20} + 2^{16})}$ **operações**.
   - Usamos dois vetores auxiliares de tamanhos 2^{16} e 2^{20}.



Radix Sort - Comparação

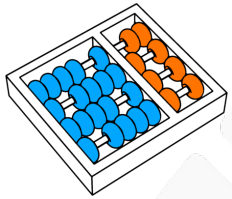
Vamos comparar Radix-Sort e Merge-Sort:

- Temos $n = 2^{20}$ números de 64 bits.
- Interpretamos os números na base $k = 2^{16}$ (cada 16 bits corresponde a um dígito).
- Cada número tem $d = 4$ dígitos.

1. Ordenando com Merge-Sort:
   - Executamos cerca de $n \log_2 n = \mathbf{20 \times 2^{20}}$ **comparações**.
   - Usamos um vetor auxiliar de tamanho 2^{20}.
2. Ordenando com Radix-Sort:
   - Utilizamos Counting-Sort como sub-rotina de ordenação.
   - Executamos cerca de $d(n + k) = \mathbf{4(2^{20} + 2^{16})}$ **operações**.
   - Usamos dois vetores auxiliares de tamanhos 2^{16} e 2^{20}.

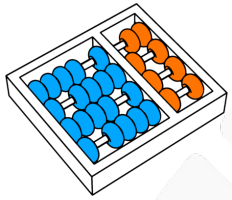
Conclusão: Radix-Sort seria mais rápido, mas usaria mais memória.

Bucket sort



Bucket Sort

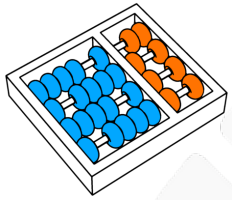
Ideia:



Bucket Sort

Ideia:

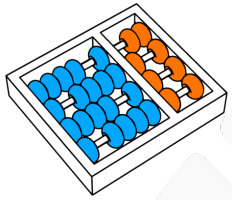
- Temos n números em $[0, 1)$ distribuídos uniformemente.



Bucket Sort

Ideia:

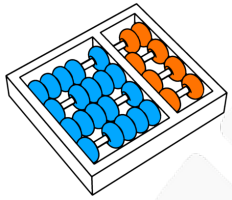
- Temos n números em $[0, 1)$ distribuídos uniformemente.
- Dividimos $[0, 1)$ em n segmentos de tamanhos iguais.



Bucket Sort

Ideia:

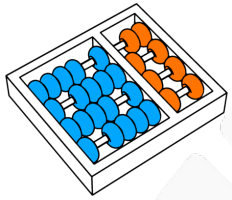
- Temos n números em $[0, 1)$ distribuídos uniformemente.
- Dividimos $[0, 1)$ em n segmentos de tamanhos iguais.
- Particionamos os elementos em listas (**buckets**) correspondentes aos segmentos.



Bucket Sort

Ideia:

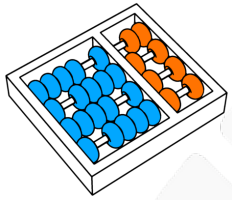
- Temos n números em $[0, 1)$ distribuídos uniformemente.
- Dividimos $[0, 1)$ em n segmentos de tamanhos iguais.
- Particionamos os elementos em listas (**buckets**) correspondentes aos segmentos.
- O número esperado de elementos por lista é constante.



Bucket Sort

Ideia:

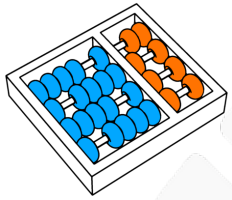
- Temos n números em $[0, 1)$ distribuídos uniformemente.
- Dividimos $[0, 1)$ em n segmentos de tamanhos iguais.
- Particionamos os elementos em listas (**buckets**) correspondentes aos segmentos.
- O número esperado de elementos por lista é constante.
- Podemos ordenar cada lista independentemente.



Bucket Sort

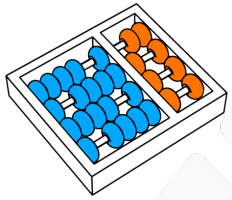
Ideia:

- Temos n números em $[0, 1)$ distribuídos uniformemente.
- Dividimos $[0, 1)$ em n segmentos de tamanhos iguais.
- Particionamos os elementos em listas (**buckets**) correspondentes aos segmentos.
- O número esperado de elementos por lista é constante.
- Podemos ordenar cada lista independentemente.
- No final, concatenamos as listas já ordenadas.



Bucket Sort - Exemplo

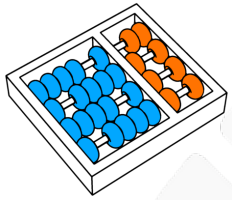
$$A = \begin{array}{c|l} 1 & .78 \\ 2 & .17 \\ 3 & .39 \\ 4 & .26 \\ 5 & .72 \\ 6 & .94 \\ 7 & .21 \\ 8 & .12 \\ 9 & .23 \\ 10 & .68 \end{array}$$



Bucket Sort - Exemplo

$$A = \begin{array}{r|l} 1 & .78 \\ 2 & .17 \\ 3 & .39 \\ 4 & .26 \\ 5 & .72 \\ 6 & .94 \\ 7 & .21 \\ 8 & .12 \\ 9 & .23 \\ 10 & .68 \end{array}$$

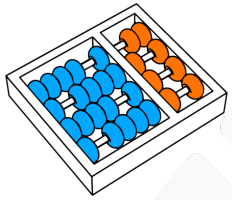
$$B = \begin{array}{r|l} 0 & \\ 1 & .12, .17 \\ 2 & .21, .23, .26 \\ 3 & .39 \\ 4 & \\ 5 & \\ 6 & .68 \\ 7 & .72, .78 \\ 8 & \\ 9 & .94 \end{array}$$



Bucket Sort - Algoritmo

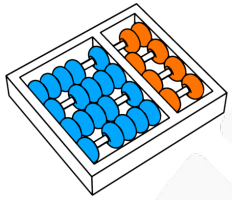
Algoritmo: BUCKET-SORT(A, n)

```
1 para i ← 0 até n − 1
2     crie uma lista ligada vazia B[i]
3 para i ← 1 até n
4     insira A[i] na lista ligada B[⌊n A[i]⌋]
5 para i ← 0 até n − 1
6     ordene a lista B[i] com INSERTION-SORT
7 concatene as listas B[0], B[1], ..., B[n − 1]
```



Bucket Sort - Correção

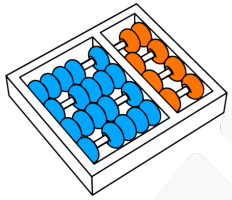
Considere dois elementos x e y da entrada com $x < y$:



Bucket Sort - Correção

Considere dois elementos x e y da entrada com $x < y$:

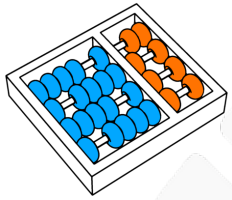
- Se ambos terminam na mesma lista:



Bucket Sort - Correção

Considere dois elementos x e y da entrada com $x < y$:

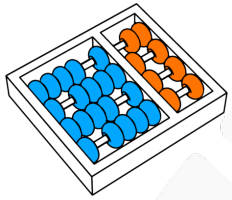
- Se ambos terminam na mesma lista:
  - x aparecerá antes de y já que a lista foi ordenada.



Bucket Sort - Correção

Considere dois elementos x e y da entrada com $x < y$:

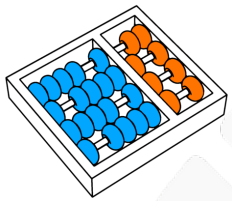
- Se ambos terminam na mesma lista:
  - x aparecerá antes de y já que a lista foi ordenada.
  - Se manterão ordenados após a concatenação.



Bucket Sort - Correção

Considere dois elementos x e y da entrada com $x < y$:

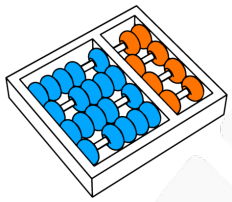
- Se ambos terminam na mesma lista:
  - x aparecerá antes de y já que a lista foi ordenada.
  - Se manterão ordenados após a concatenação.
- Se terminam em listas $B[i]$ e $B[j]$, respectivamente:



Bucket Sort - Correção

Considere dois elementos x e y da entrada com $x < y$:

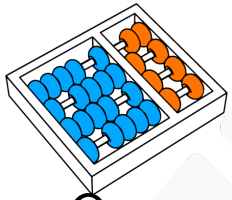
- Se ambos terminam na mesma lista:
  - x aparecerá antes de y já que a lista foi ordenada.
  - Se manterão ordenados após a concatenação.
- Se terminam em listas $B[i]$ e $B[j]$, respectivamente:
  - Como $x < y$, temos $i = \lfloor nx \rfloor \leq \lfloor ny \rfloor = j$ e então $i < j$.



Bucket Sort - Correção

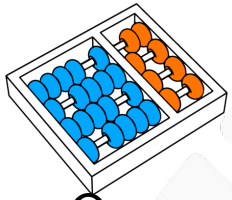
Considere dois elementos x e y da entrada com $x < y$:

- Se ambos terminam na mesma lista:
  - x aparecerá antes de y já que a lista foi ordenada.
  - Se manterão ordenados após a concatenação.
- Se terminam em listas $B[i]$ e $B[j]$, respectivamente:
  - Como $x < y$, temos $i = \lfloor nx \rfloor \leq \lfloor ny \rfloor = j$ e então $i < j$.
  - Assim, x aparecerá antes de y após a concatenação.



Bucket Sort - Complexidade

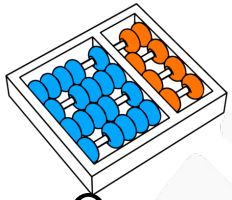
O tempo de execução é uma **váriável aleatória**:



Bucket Sort - Complexidade

O tempo de execução é uma **váriável aleatória**:

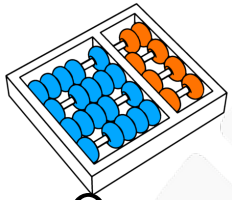
- Denote por $T(n)$ o tempo de execução do algoritmo.



Bucket Sort - Complexidade

O tempo de execução é uma **váriável aleatória**:

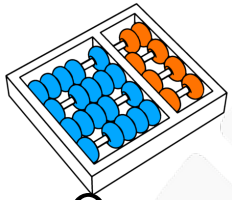
- Denote por $T(n)$ o tempo de execução do algoritmo.
- Denote por n_i o tamanho de $B[i]$.



Bucket Sort - Complexidade

O tempo de execução é uma **váriável aleatória**:

- Denote por $T(n)$ o tempo de execução do algoritmo.
- Denote por n_i o tamanho de $B[i]$.
- Observe que o número de elementos é $n = \sum_{i=1}^{n} n_i$.

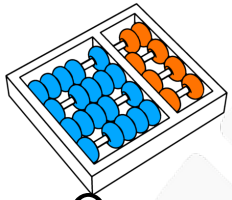


Bucket Sort - Complexidade

O tempo de execução é uma **váriável aleatória**:

- Denote por $T(n)$ o tempo de execução do algoritmo.
- Denote por n_i o tamanho de $B[i]$.
- Observe que o número de elementos é $n = \sum_{i=1}^{n} n_i$.

Somando o tempo das operações:



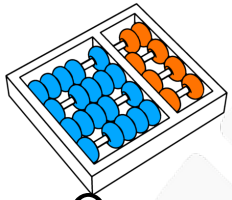
Bucket Sort - Complexidade

O tempo de execução é uma **váriável aleatória**:

- Denote por $T(n)$ o tempo de execução do algoritmo.
- Denote por n_i o tamanho de $B[i]$.
- Observe que o número de elementos é $n = \sum_{i=1}^{n} n_i$.

Somando o tempo das operações:

- Particionamos os elementos em tempo $\Theta(n)$.



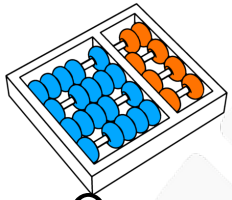
Bucket Sort - Complexidade

O tempo de execução é uma **váriável aleatória**:

- Denote por $T(n)$ o tempo de execução do algoritmo.
- Denote por n_i o tamanho de $B[i]$.
- Observe que o número de elementos é $n = \sum_{i=1}^{n} n_i$.

Somando o tempo das operações:

- Particionamos os elementos em tempo $\Theta(n)$.
- Ordenamos cada lista em tempo $\Theta(n_i^2)$.



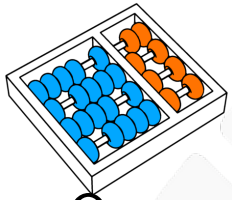
Bucket Sort - Complexidade

O tempo de execução é uma **váriável aleatória**:

- Denote por $T(n)$ o tempo de execução do algoritmo.
- Denote por n_i o tamanho de $B[i]$.
- Observe que o número de elementos é $n = \sum_{i=1}^{n} n_i$.

Somando o tempo das operações:

- Particionamos os elementos em tempo $\Theta(n)$.
- Ordenamos cada lista em tempo $\Theta(n_i^2)$.
- Concatenamos todas as listas em tempo $\Theta(n)$.



Bucket Sort - Complexidade

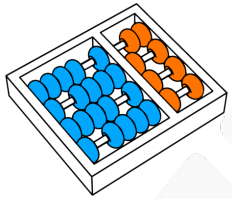
O tempo de execução é uma **váriável aleatória**:

- Denote por $T(n)$ o tempo de execução do algoritmo.
- Denote por n_i o tamanho de $B[i]$.
- Observe que o número de elementos é $n = \sum_{i=1}^{n} n_i$.

Somando o tempo das operações:

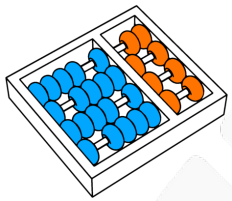
- Particionamos os elementos em tempo $\Theta(n)$.
- Ordenamos cada lista em tempo $\Theta(n_i^2)$.
- Concatenamos todas as listas em tempo $\Theta(n)$.

$$T(n) = \sum_{i=0}^{n-1} \Theta(n_i^2) + \Theta(n).$$



Bucket Sort - Pior caso

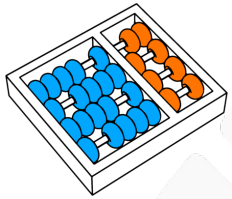
Pior caso



Bucket Sort - Pior caso

Pior caso

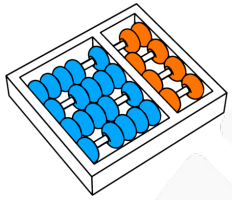
$$T(n) \leq \sum_{i=0}^{n-1} cn_i^2 + \Theta(n)$$



Bucket Sort - Pior caso

Pior caso

$$T(n) \leq \sum_{i=0}^{n-1} cn_i^2 + \Theta(n) \leq cn^2 + \Theta(n) = O(n^2).$$

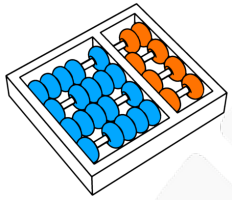


Bucket Sort - Pior caso

Pior caso

$$T(n) \leq \sum_{i=0}^{n-1} cn_i^2 + \Theta(n) \leq cn^2 + \Theta(n) = O(n^2).$$

- Um pior caso ocorre se todos números caem em uma lista.

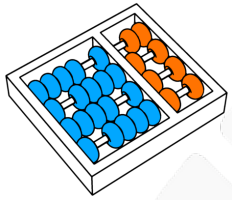


Bucket Sort - Pior caso

Pior caso

$$T(n) \leq \sum_{i=0}^{n-1} cn_i^2 + \Theta(n) \leq cn^2 + \Theta(n) = O(n^2).$$

- Um pior caso ocorre se todos números caem em uma lista.
- O tempo de execução é maior se há listas muito grandes.

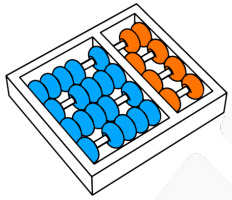


Bucket Sort - Pior caso

Pior caso

$$T(n) \leq \sum_{i=0}^{n-1} cn_i^2 + \Theta(n) \leq cn^2 + \Theta(n) = O(n^2).$$

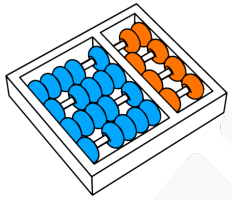
- Um pior caso ocorre se todos números caem em uma lista.
- O tempo de execução é maior se há listas muito grandes.
- Mas o **número esperado** em cada lista é pequeno.



Bucket Sort - Tempo esperado

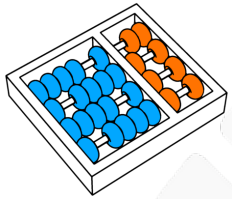
Tempo esperado:

$$
\begin{aligned}
E[T(n)] &= E\left[\sum_{i=0}^{n-1} cn_i^2\right] + \Theta(n) \\
&= \sum_{i=0}^{n-1} E[cn_i^2] + \Theta(n) \\
&= \sum_{i=0}^{n-1} cE[n_i^2] + \Theta(n).
\end{aligned}
$$



Bucket Sort - Tempo esperado (cont)

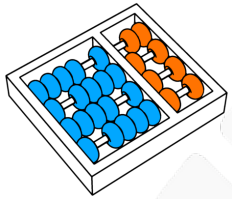
Queremos calcular $E[n_i^2]$:



Bucket Sort - Tempo esperado (cont)

Queremos calcular $E[n_i^2]$:

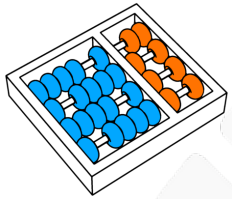
- Seja X_{ij} a variável que indica se $A[j]$ está em $B[i]$.



Bucket Sort - Tempo esperado (cont)

Queremos calcular $E[n_i^2]$:

- Seja X_{ij} a variável que indica se $A[j]$ está em $B[i]$.
- Assim, temos $n_i = \sum_{j=1}^{n} X_{ij}$.

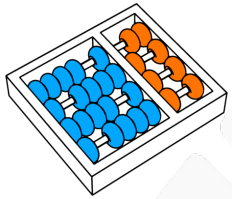


Bucket Sort - Tempo esperado (cont)

Queremos calcular $E[n_i^2]$:

- Seja X_{ij} a variável que indica se $A[j]$ está em $B[i]$.
- Assim, temos $n_i = \sum_{j=1}^{n} X_{ij}$.

$$E[n_i^2] = E\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right]$$



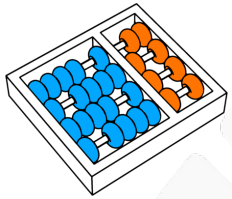
Bucket Sort - Tempo esperado (cont)

Queremos calcular $E[n_i^2]$:

- Seja X_{ij} a variável que indica se $A[j]$ está em $B[i]$.
- Assim, temos $n_i = \sum_{j=1}^{n} X_{ij}$.

$$E[n_i^2] = E\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right]$$

$$= E\left[\left(\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right)\right]$$

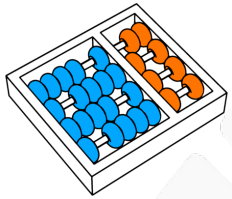


Bucket Sort - Tempo esperado (cont)

Queremos calcular $E[n_i^2]$:

- Seja X_{ij} a variável que indica se $A[j]$ está em $B[i]$.
- Assim, temos $n_i = \sum_{j=1}^{n} X_{ij}$.

$$
\begin{aligned}
E[n_i^2] &= E\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right] \\
&= E\left[\left(\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right)\right] \\
&= E\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{j=1}^{n}\sum_{k=1,k\neq j}^{n} X_{ij}X_{ik}\right]
\end{aligned}
$$

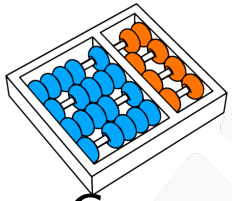


Bucket Sort - Tempo esperado (cont)

Queremos calcular $E[n_i^2]$:

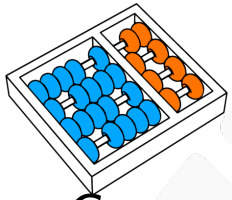
- Seja X_{ij} a variável que indica se $A[j]$ está em $B[i]$.
- Assim, temos $n_i = \sum_{j=1}^{n} X_{ij}$.

$$
\begin{aligned}
E[n_i^2] &= E\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right] \\
&= E\left[\left(\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right)\right] \\
&= E\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{j=1}^{n}\sum_{k=1,k\neq j}^{n} X_{ij}X_{ik}\right] \\
&= \sum_{j=1}^{n} E[X_{ij}^2] + \sum_{j=1}^{n}\sum_{k=1,k\neq j}^{n} E[X_{ij}X_{ik}]
\end{aligned}
$$



Bucket Sort - Tempo esperado (cont)

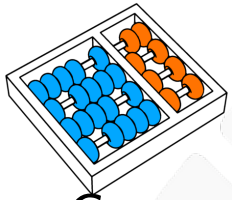
Como X_{ij} e X_{ik} são independentes:



Bucket Sort - Tempo esperado (cont)

Como X_{ij} e X_{ik} são independentes:

$$E[n_i^2] = \sum_{j=1}^{n} E[X_{ij}^2] + \sum_{j=1}^{n} \sum_{k=1, k \neq j}^{n} \mathbf{E[X_{ij} X_{ik}]}$$

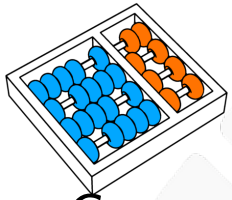


Bucket Sort - Tempo esperado (cont)

Como X_{ij} e X_{ik} são independentes:

$$E[n_i^2] = \sum_{j=1}^{n} E[X_{ij}^2] + \sum_{j=1}^{n} \sum_{k=1,k\neq j}^{n} \mathbf{E[X_{ij}X_{ik}]}$$

$$= \sum_{j=1}^{n} E[X_{ij}] + \sum_{j=1}^{n} \sum_{k=1,k\neq j}^{n} \mathbf{E[X_{ij}]E[X_{ik}]}$$



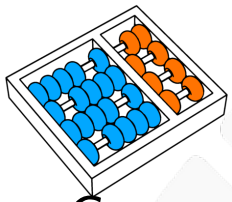
Bucket Sort - Tempo esperado (cont)

Como X_{ij} e X_{ik} são independentes:

$$E[n_i^2] = \sum_{j=1}^{n} E[X_{ij}^2] + \sum_{j=1}^{n} \sum_{k=1,k\neq j}^{n} \mathbf{E[X_{ij}X_{ik}]}$$

$$= \sum_{j=1}^{n} E[X_{ij}] + \sum_{j=1}^{n} \sum_{k=1,k\neq j}^{n} \mathbf{E[X_{ij}]E[X_{ik}]}$$

$$= \sum_{j=1}^{n} \frac{1}{n} + \sum_{j=1}^{n} \sum_{k=1,k\neq j}^{n} \frac{1}{n}\frac{1}{n}$$



Bucket Sort - Tempo esperado (cont)

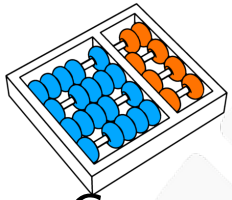
Como X_{ij} e X_{ik} são independentes:

$$E[n_i^2] = \sum_{j=1}^{n} E[X_{ij}^2] + \sum_{j=1}^{n} \sum_{k=1,k\neq j}^{n} \mathbf{E[X_{ij}X_{ik}]}$$

$$= \sum_{j=1}^{n} E[X_{ij}] + \sum_{j=1}^{n} \sum_{k=1,k\neq j}^{n} \mathbf{E[X_{ij}]E[X_{ik}]}$$

$$= \sum_{j=1}^{n} \frac{1}{n} + \sum_{j=1}^{n} \sum_{k=1,k\neq j}^{n} \frac{1}{n}\frac{1}{n}$$

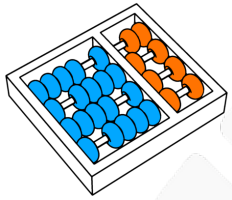
$$= 1 + n(n-1)\frac{1}{n^2}$$



Bucket Sort - Tempo esperado (cont)

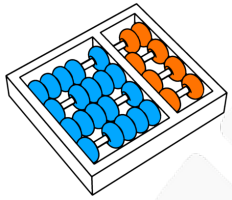
Como X_{ij} e X_{ik} são independentes:

$$
\begin{aligned}
E[n_i^2] &= \sum_{j=1}^{n} E[X_{ij}^2] + \sum_{j=1}^{n} \sum_{k=1,k\neq j}^{n} \mathbf{E[X_{ij}X_{ik}]} \\
&= \sum_{j=1}^{n} E[X_{ij}] + \sum_{j=1}^{n} \sum_{k=1,k\neq j}^{n} \mathbf{E[X_{ij}]E[X_{ik}]} \\
&= \sum_{j=1}^{n} \frac{1}{n} + \sum_{j=1}^{n} \sum_{k=1,k\neq j}^{n} \frac{1}{n}\frac{1}{n} \\
&= 1 + n(n-1)\frac{1}{n^2} \\
&= 2 - \frac{1}{n} = O(1).
\end{aligned}
$$



Bucket Sort - Tempo esperado (cont)

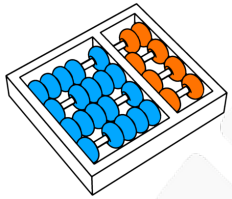
Voltando à estimativa de $E[T(n)]$, temos:



Bucket Sort - Tempo esperado (cont)

Voltando à estimativa de $E[T(n)]$, temos:

$$E[T(n)] = \sum_{i=0}^{n-1} cE[n_i^2] + \Theta(n)$$

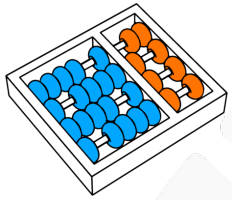


Bucket Sort - Tempo esperado (cont)

Voltando à estimativa de $E[T(n)]$, temos:

$$E[T(n)] = \sum_{i=0}^{n-1} cE[n_i^2] + \Theta(n)$$

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



Bucket Sort - Tempo esperado (cont)

Voltando à estimativa de $E[T(n)]$, temos:

$$\begin{aligned} E[T(n)] &= \sum_{i=0}^{n-1} cE[n_i^2] + \Theta(n) \\ &= \sum_{i=0}^{n-1} cO(1) + \Theta(n) \\ &= \Theta(n). \end{aligned}$$

Ordenação linear

MC458 - Projeto e Análise de Algoritmos I

10/25

19

Santiago Valdés Ravelo
https://ic.unicamp.br/~santiago/
ravelo@unicamp.br

UNICAMP