

ORDENAÇÃO LINEAR

MC458 - Projeto e Análise de Algoritmos I

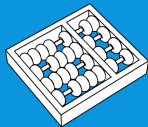
Santiago Valdés Ravelo
<https://ic.unicamp.br/~santiago/ravelo@unicamp.br>

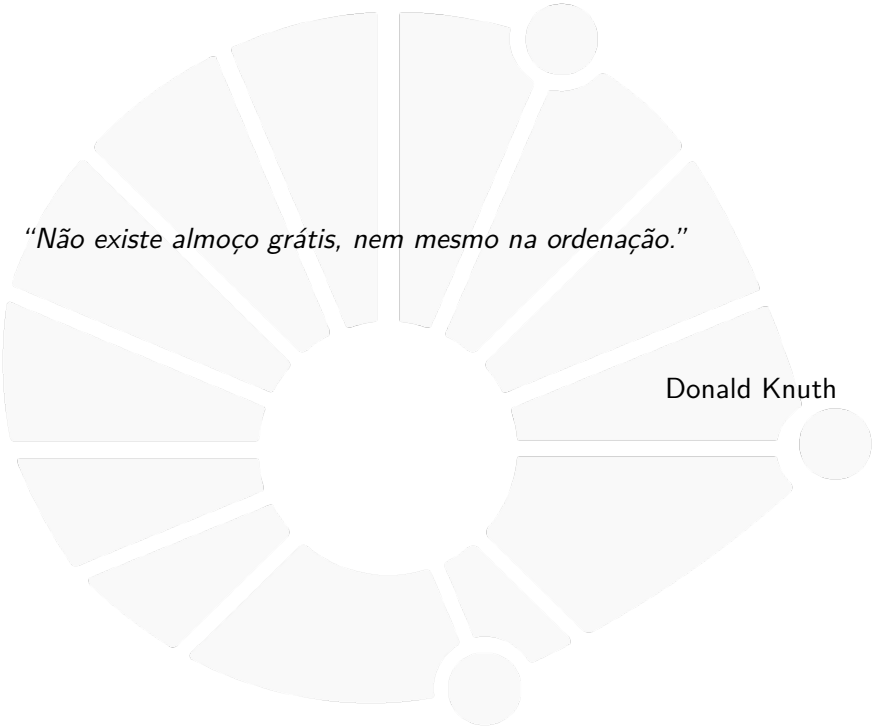
10/25

19



UNICAMP



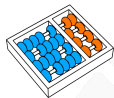


“Não existe almoço grátis, nem mesmo na ordenação.”

Donald Knuth



INTRODUÇÃO



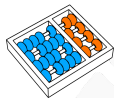
Algoritmos lineares para ordenação

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.



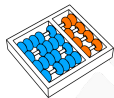
COUNTING SORT



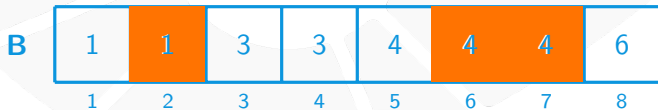
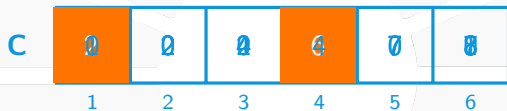
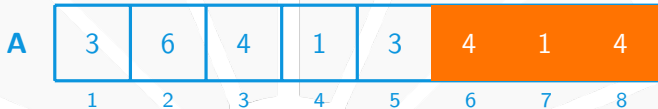
Counting Sort

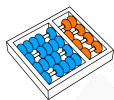
Ideia:

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

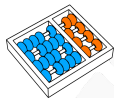




Counting Sort - Algoritmo

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

- 1 **para** $i \leftarrow 0$ até k
 - 2 $C[i] \leftarrow 0$
 - 3 **para** $j \leftarrow 1$ até n
 - 4 $C[A[j]] \leftarrow C[A[j]] + 1$
 - 5 **para** $i \leftarrow 1$ até k
 - 6 $C[i] \leftarrow C[i] + C[i - 1]$
 - 7 **para** $j \leftarrow n$ até 1
 - 8 $B[C[A[j]]] \leftarrow A[j]$
 - 9 $C[A[j]] \leftarrow C[A[j]] - 1$
-



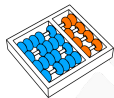
Counting Sort - Complexidade

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:

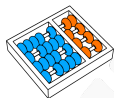
- ▶ Um algoritmo é **in-place** se a quantidade de memória adicional requerida independe de n .
- ▶ São in-place INSERTION-SORT, QUICK-SORT, HEAP-SORT.
- ▶ Não são in-place MERGE-SORT, COUNTING-SORT.

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.

A diagram illustrating the Radix Sort algorithm. It features a large, light gray circular structure composed of several radial segments, resembling a fan or a wheel. A solid blue horizontal bar is positioned across the center of the diagram. The text "RADIX SORT" is written in white, serif, all-caps font within this blue bar. The circular structure is divided into approximately 12 segments, with three small gray circles placed at the outer edge of the structure, one at the top, one on the right, and one at the bottom.

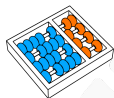
RADIX SORT



Radix Sort

Ideia:

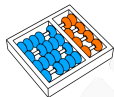
- ▶ A entrada é um vetor $A[1 \dots n]$ de inteiros.
- ▶ Cada inteiro é representado na **base k -ária**
- ▶ Supomos que um inteiro tem d dígitos.
- ▶ Por exemplo, CEPs são inteiros de 8 dígitos na base 10.
- ▶ Ordenaremos **um dígito por vez** com um algoritmo estável.
- ▶ Ordenamos primeiro os dígitos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



Radix Sort - Exemplo

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

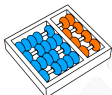
→ → →



Radix Sort - Algoritmo

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

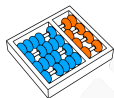
- 1 **para** $i \leftarrow 1$ **até** d
 - 2 Ordene $A[1 \dots n]$ pelo i -ésimo dígito usando um método estável
-



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 $a < b$:
 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?

- ▶ 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:

- ▶ 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**.
- ▶ Usamos um vetor auxiliar de tamanho 2^{20} .

2. Ordenando com RADIX-SORT:

- ▶ Utilizamos COUNTING-SORT como sub-rotina de ordenação.
- ▶ Executamos cerca de $d(n + k) = 4(2^{20} + 2^{16})$ **operações**.
- ▶ Usamos dois vetores auxiliares de tamanhos 2^{16} e 2^{20} .

Conclusão: RADIX-SORT seria mais rápido, mas usaria mais memória.

A diagram illustrating the Bucket Sort algorithm. It features a large circle divided into 12 equal-sized sectors, representing buckets. Three of these sectors are highlighted with a light gray fill, while the others are white. The highlighted sectors are located at the top, bottom, and right positions. A solid blue horizontal bar is centered across the middle of the circle, containing the text "BUCKET SORT" in white, serif, all-caps font.

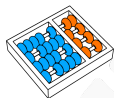
BUCKET SORT



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

$A =$

1		.78
2		.17
3		.39
4		.26
5		.72
6		.94
7		.21
8		.12
9		.23
10		.68

$B =$

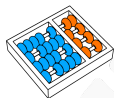
0		
1		.12, .17
2		.21, .23, .26
3		.39
4		
5		
6		.68
7		.72, .78
8		
9		.94



Bucket Sort - Algoritmo

Algoritmo: BUCKET-SORT(A, n)

- 1 **para** $i \leftarrow 0$ até $n - 1$
 - 2 └ crie uma lista ligada vazia $B[i]$
 - 3 **para** $i \leftarrow 1$ até n
 - 4 └ insira $A[i]$ na lista ligada $B[\lfloor nA[i] \rfloor]$
 - 5 **para** $i \leftarrow 0$ até $n - 1$
 - 6 └ ordene a lista $B[i]$ com INSERTION-SORT
 - 7 concatene as listas $B[0], B[1], \dots, B[n - 1]$
-



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

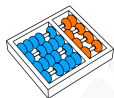
O tempo de execução é uma **váriável aleatória**:

- ▶ 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

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

- ▶ 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 $E[n_i^2]$:

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

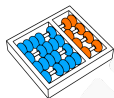
$$\begin{aligned}
 E[n_i^2] &= E \left[\left(\sum_{j=1}^n X_{ij} \right)^2 \right] \\
 &= E \left[\left(\sum_{j=1}^n \sum_{k=1}^n X_{ij} X_{ik} \right) \right] \\
 &= E \left[\sum_{j=1}^n X_{ij}^2 + \sum_{j=1}^n \sum_{k=1, k \neq j}^n X_{ij} X_{ik} \right] \\
 &= \sum_{j=1}^n E[X_{ij}^2] + \sum_{j=1}^n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



Bucket Sort - Tempo esperado (cont)

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

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



Bucket Sort - Tempo esperado (cont)

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

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

ORDENAÇÃO LINEAR

MC458 - Projeto e Análise de Algoritmos I

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

10/25

19



UNICAMP

