

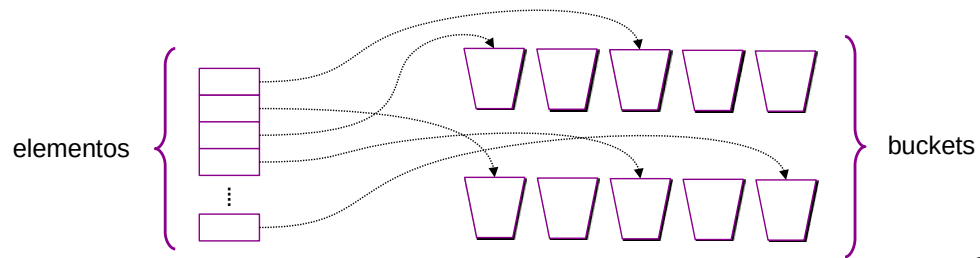
Programa

- Algoritmos de ordenação:
 - Bucket Sort
 - TimSort
 - Counting Sort
 - Radix Sort
 - Bitonic Sort

Bucket Sort

► **Bucket Sort** é uma técnica de ordenação que **divide** um conjunto de **elementos** em um número finito de **recipientes** (chamados **buckets**)

- Cada **recipiente** é então **ordenado individualmente** usando **qualquer algoritmo** de ordenação



Funcionamento do Bucket Sort

- Criar **n buckets vazios** para mapeamento de elementos em um **intervalo entre [0..1)**, que podem ser construídos com uma **lista de listas** com n posições
 - **buckets = [[] for _ in range(n)]**, onde $n = \text{len}(\text{array})$
- Para cada elemento do **array inicial** fazer o seguinte:
 - 1) Inserir o elemento $\text{array}[i]$ no bucket:
 - Forma 1: **bucket[int(n * array[i])]**, onde $n = \text{len}(\text{array})$
 - Forma 2: **bucket[int(array[i] / s)]**, onde $s = \text{max}(\text{array})/\text{len}(\text{array})$
 - 2) **Ordenar os buckets** individuais usando ordenação por inserção
 - 3) **Concatenar** (reunir) todos os **buckets ordenados**

Exemplo com Bucket Sort

- Como funciona a **ordenação por bucket**?
- Exemplo:

0.78	0.17	0.39	0.26	0.72	0.94	0.21	0.12	0.23	0.68
------	------	------	------	------	------	------	------	------	------

- **1º. passo:** criar um array de 10 slots, onde **cada slot** represente **um bucket**

null	null	null	null	null	null	null	null	null	null
0	1	2	3	4	5	6	7	8	9

Exemplo com Bucket Sort

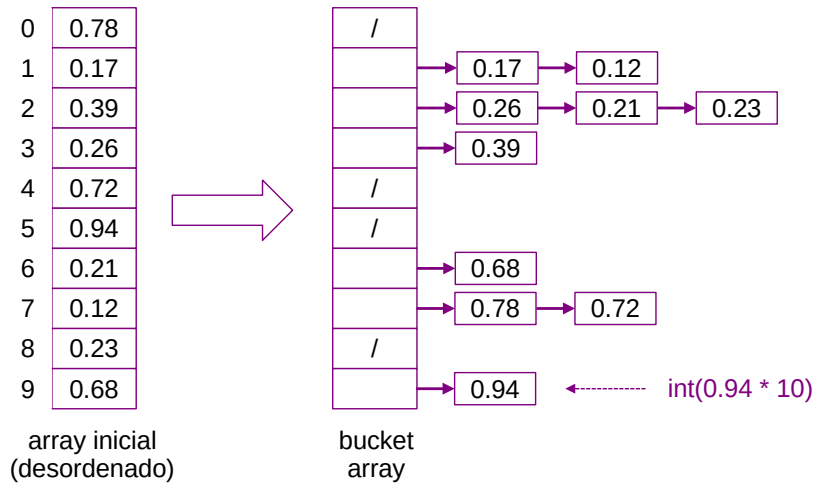
- **2º. passo:** inserir elementos nos buckets com base em seu respectivo intervalo, ou seja, **cada elemento é multiplicado pelo tamanho do array de buckets** (10 neste caso)
 - Por exemplo, $0.23 * 10$ é igual a 2.3 que, convertido para inteiro, representará o índice do intervalo **$\text{int}(2.3) = 2$**
- **Inserir** o elemento no **intervalo correspondente** ao índice calculado e repetir essas etapas para todos os elementos do array inicial (desordenado)

null	null	0.23	null	null	null	null	null	null	null
0	1	2	3	4	5	6	7	8	9

6

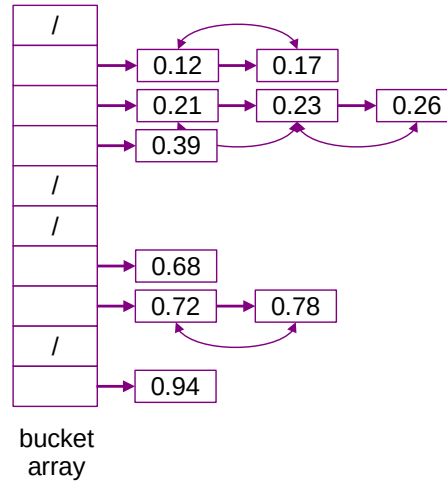
Aqui usamos a “Forma 1” de inserir os elementos nos respectivos buckets, ou seja, `bucket[int(n * array[i])]`

Exemplo com Bucket Sort



Exemplo com Bucket Sort

- **3º. passo:** ordenar os elementos em cada bucket usando **qualquer algoritmo de ordenação** estável (geralmente, o insertion sort)

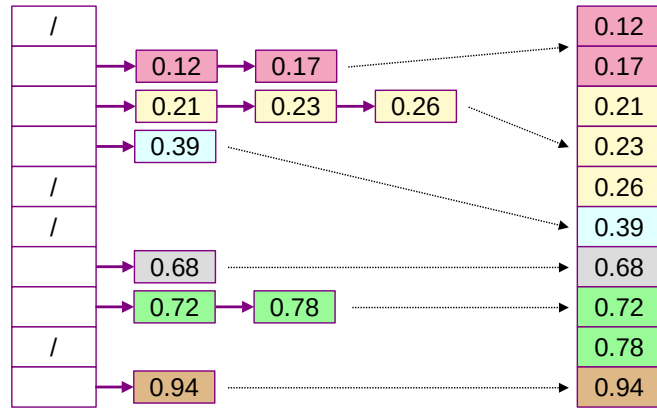


Exemplo com Bucket Sort

- **4º. passo:** reunir os elementos de cada bucket no array original
- Para reunir os elementos de cada bucket é necessário:
 - **Iterar** em cada **bucket** em **ordem** (do 1º. p/ o último)
 - Inserir cada **elemento individual** do bucket no **array original**
 - Depois que um **elemento** é **copiado**, ele é **removido do bucket**
 - **Repetir** este processo para **todos os buckets** até que todos os **elementos** tenham sido **reunidos** no **array original**

Exemplo com Bucket Sort

- 4º. passo:




Exemplo com Bucket Sort


- Por fim, o array original foi modificado e agora contém os elementos ordenados

array original

0.78	0.17	0.39	0.26	0.72	0.94	0.21	0.12	0.23	0.68
------	------	------	------	------	------	------	------	------	------




Bucket Sort



array modificado

0.12	0.17	0.21	0.23	0.26	0.39	0.68	0.72	0.78	0.94
------	------	------	------	------	------	------	------	------	------

TimSort

 **TimSort** é um algoritmo de ordenação criado por **Tim Peters em 2002**. É considerado um "**algoritmo híbrido**", derivado da ordenação por intercalação/mesclagem (**merge sort**) e da ordenação por inserção (**insertion sort**)

- Foi projetado para funcionar bem em muitos tipos de dados do mundo real e é o algoritmo de ordenação padrão do Python, usado pelas funções **sorted()** e **list.sort()**

Funcionamento do TimSort

- A ideia principal por trás do TimSort é **explorar a ordem natural** existente nos **dados**, para **minimizar** o número de **comparações e trocas**
- O algoritmo faz isso **dividindo o array inicial** em pequenas **subsequências** chamadas “execuções” (**runs**), que já estão naturalmente ordenadas, e depois mesclando essas execuções usando um **algoritmo de ordenação por mesclagem** modificado

Exemplo com o TimSort

- Como funciona a **ordenação** com o **TimSort**?
- Exemplo:

4	2	8	6	1	5	9	3	7
---	---	---	---	---	---	---	---	---

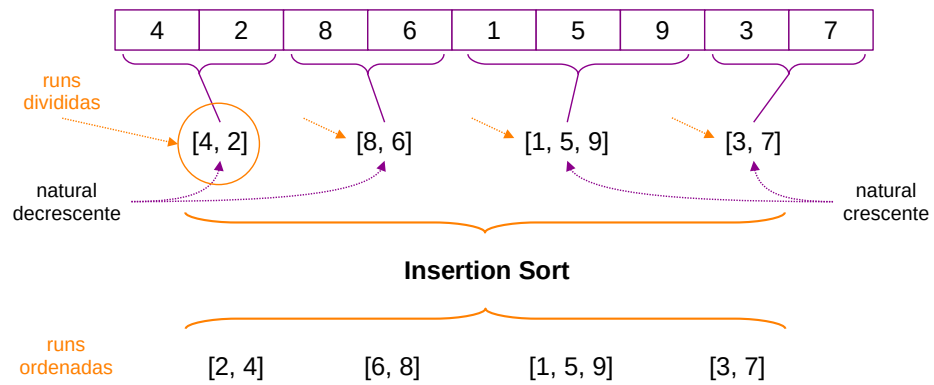
- **1º. passo:** Definir o tamanho da execução
 - **Tamanho mínimo** de cada subsequência (**run**): 32
(ignoraremos esta etapa porque nosso array é pequeno)

Funcionamento do TimSort

- **2º. passo:** dividir o array em execuções (runs)
- Para fazer isso é necessário:
 - Separar o array em **execuções naturalmente ordenadas**, seja em ordem **crescente** ou em ordem **decrescente**
 - **Ordenar as execuções** (subsequências) na **ordem desejada**. Nesta etapa pode ser utilizado o **insertion sort**

Funcionamento do TimSort

- 2º. passo:

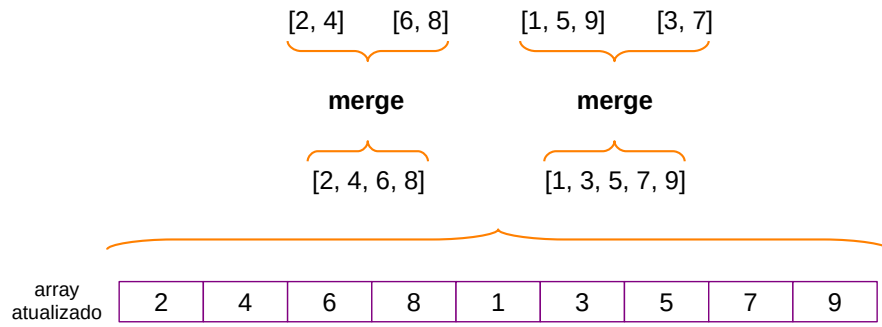


Funcionamento do TimSort

- **3º. passo:** mesclar as execuções
- Para fazer isso é necessário:
 - **Mesclar as execuções** ordenadas usando o **algoritmo** de classificação por **mesclagem** modificado do Timsort
 - E **atualizar o array** original

Funcionamento do TimSort

- **3º. passo:**



Funcionamento do TimSort

- **4º. passo:** ajustar o tamanho da execução
 - Uma característica do TimSort é que **após cada operação de mesclagem**, o **tamanho da execução** deve ser **dobrado** até que exceda o comprimento do array
 - O **tamanho da execução dobra**: 32, 64, 128 (ignoraremos esta etapa porque nosso array é pequeno)
- **5º passo:** continuar mesclando
 - **Repetir** o processo de **mesclagem** até que todo o **array esteja ordenado**

array final ordenado	1	2	3	4	5	6	7	8	9
-------------------------	---	---	---	---	---	---	---	---	---

Counting Sort

- ▶ **Counting Sort**, ou ordenação por contagem, é um algoritmo que **utiliza chaves** para definir as **posições dos elementos** no **array de saída** (ordenado)
- A ideia é **ordenar números inteiros** cujos **índices** dos **arrays** também são **inteiros** e podem ser **mapeados** para uma **posição** (slot) de **mesmo valor** em um **array auxiliar** ($\text{array}[i] = i$)

Funcionamento do Counting Sort

- Utiliza **três arrays**, A, B e C, onde:
 - A = **array de entrada** (desordenado)
 - B = **array de saída** (ordenado)
 - C = **array de contagem** (frequência)
- Também utiliza um **chave k**, onde:
 - k é o valor do **maior elemento** do array de entrada

Exemplo com o Counting Sort

- Como funciona a **ordenação** com o **Counting Sort**?
- Exemplo:

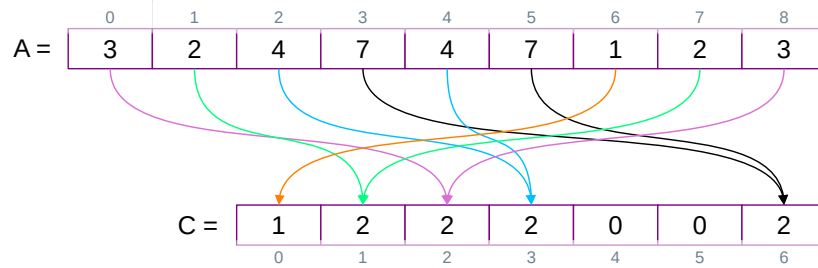
3	2	4	7	4	7	1	2	3
0	1	2	3	4	5	6	7	8

- **1º. passo:** obter o valor de k , que no exemplo é o maior valor do array de entrada ($k = 7$)
- **2º. passo:** criar um array de contagem, C , que possui o número de slots igual a k

null	null	null	null	null	null	null
0	1	2	3	4	5	6

Exemplo com o Counting Sort

- **3º. passo:** registrar em C a frequência dos elementos de A



- Os valores 3, 2, 4 e 7 aparecem duas vezes em A
- O valor 1 aparece uma vez
- Os valores 5 e 6 não aparecem (= 0)

Exemplo com o Counting Sort

- **4º. passo:** calcular o valor acumulativo de elementos em C

C =

	+						
	↖						
1	2	2	2	0	0	2	
0	1	2	3	4	5	6	

- Calcular $C[i] = C[i] + C[i-1]$, para 1 de 1 até 6

C =

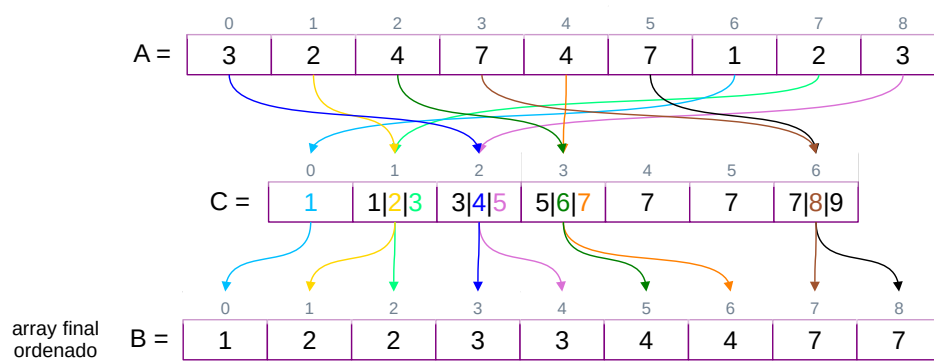
	2+1	2+3	2+5	0+7	0+7	2+7	
1	3	5	7	7	7	9	
0	1	2	3	4	5	6	

Exemplo com o Counting Sort


- **5º. passo:** registrar os elementos de A em B utilizando os valores de C
- Para isso é necessário:
 - **Contando do último** para o primeiro **elemento do array A**, verificar o **elemento em C** cujo slot é indicado por $A[i]$, tal que $C[A[i]]$
 - O **elemento $C[A[i]]$** deverá indicar o **slot de B** para **registrar** o elemento de $A[i]$
 - Por fim, **decrementar em 1** o **elemento de $C[A[i]]$** para que um novo valor do array A não ocupe um mesmo slot de B

Exemplo com o Counting Sort

- 5º. passo:



Radix Sort

 **Radix Sort** é um algoritmo de ordenação linear que **classifica elementos** processando-os **dígito por dígito**. Em vez de comparar os elementos diretamente, ele **distribui** os elementos em slots com base no **valor de cada dígito**

- Ao classificar repetidamente os elementos por seus **dígitos significativos**, do menos significativo ao mais significativo (ou vice-versa), o Radix Sort obtém a **ordenação final dos elementos**

Funcionamento do Radix Sort

- A ideia principal no **Radix Sort** é explorar o **conceito** de **valor posicional**. Ele pressupõe que a **ordenação** dos números, **dígito por dígito**, resultará eventualmente em uma lista totalmente ordenada
- A **ordenação** com **Radix** pode ser realizada usando **diferentes variantes**, como:
 - Ordenação Radix de Dígitos Menos Significativos (LSD) ou
 - Ordenação Radix de Dígitos Mais Significativos (MSD)

* LSD e MSD diferem na ordem em que as iterações do algoritmo são realizadas

Exemplo com o Radix Sort

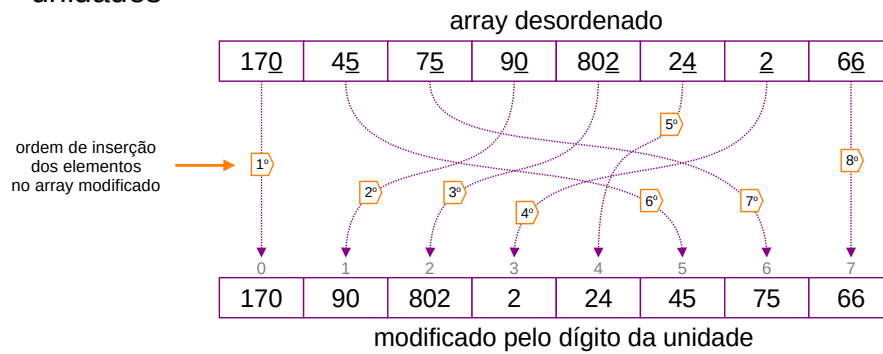
- Como funciona a **ordenação com Radix**?
- Exemplo:

170	45	75	90	802	24	2	66
-----	----	----	----	-----	----	---	----

- **1º. passo:** Encontrar o **maior elemento** do array, que é **802**. Ele tem **três dígitos**, então será necessário **iterar três vezes**, uma para cada lugar significativo

Exemplo com o Radix Sort

- **2º. passo:** classificar os elementos com base no dígito das unidades

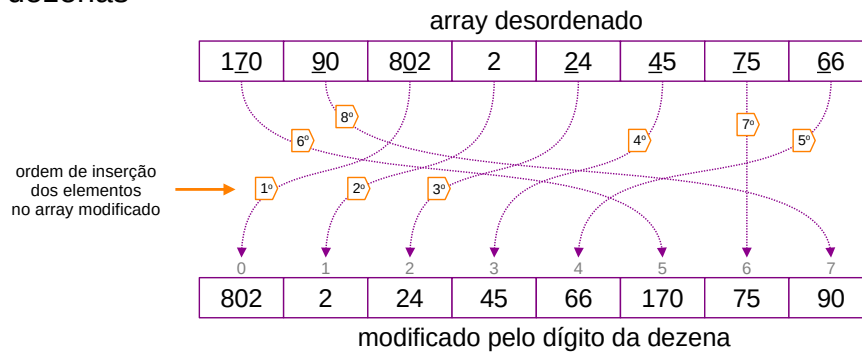


30

Neste 2º. passo a ordem de inserção dos elementos no array modificado é definida pelo dígito da unidade, e em ordem crescente 0 (170), 0 (90), 2 (802), 2 (2), 4 (24), 5 (45), 5 (75) e 6 (66)

Exemplo com o Radix Sort

- **3º. passo:** classificar os elementos com base no dígito das dezenas

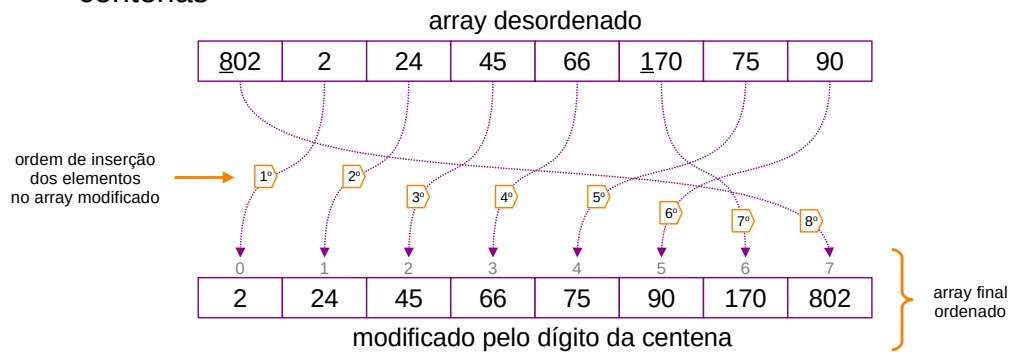


31

Neste 3º. passo a ordem de inserção dos elementos no array modificado é definida pelo dígito da dezena, e em ordem crescente 0 (802), 2 (002), 2 (24), 4 (45), 6 (66), 7 (170), 7 (75) e 9 (90)

Exemplo com o Radix Sort

- **4º. passo:** classificar os elementos com base no dígito das centenas



32

Neste 4º. passo a ordem de inserção dos elementos no array modificado é definida pelo dígito da centena, e em ordem crescente 0 (002), 0 (024), 0 (045), 0 (066), 0 (075), 0 (090), 1 (170) e 8 (802)