

Algoritmos de Ordenação - Numpy

Funcionamento, exemplos e mais.



Tópicos

- O que é o método sort
- Algoritmos de ordenação dentro do `np.sort`
- Exemplo de uso prático e comparação entre algoritmos
- Considerações finais



- O método sort da NumPy é utilizado para ordenar arrays de maneira eficiente.
- Pode ordenar arrays unidimensionais e multidimensionais.
- Possui diversos algoritmos de ordenação integrados (quicksort, mergesort e heapsort)



 np.sort

```
import numpy as np
```

```
a = np.array([5, 8, 2, 30, 0, 1])
```

```
np.sort(a, axis=-1, kind='quicksort', order=None, stable=None)
```

```
array([0, 1, 2, 5, 8, 30])
```

- a: array a ser ordenado.
- axis: eixo ao longo do qual ordenar.
- kind: algoritmo de ordenação.
- order: especifica quais campos ordenar se a é um array estruturado.
- stable: Se True, o array de saída mantém a ordem relativa em valores iguais.



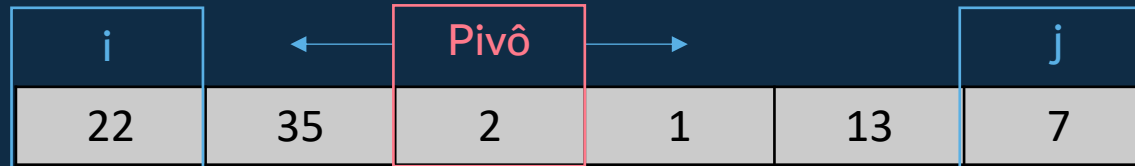
Um algoritmo de ordenação rápido que utiliza a técnica de **Dividir e Conquistar**. Usa um pivô como referência.



Quicksort - Funcionamento

Array =

22	35	2	1	13	7
----	----	---	---	----	---



SE $i \geq \text{pivô}$ E $j \leq \text{pivô}$, TROCA i com j

Senão, incremente / decremente aquele i e j que não atendeu à comparação



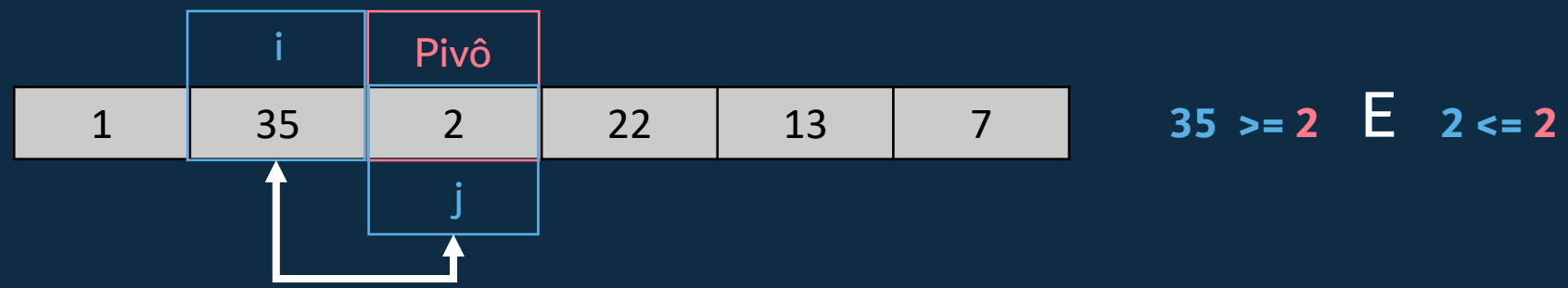
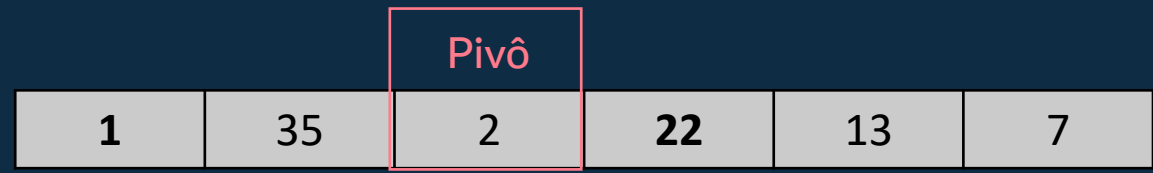
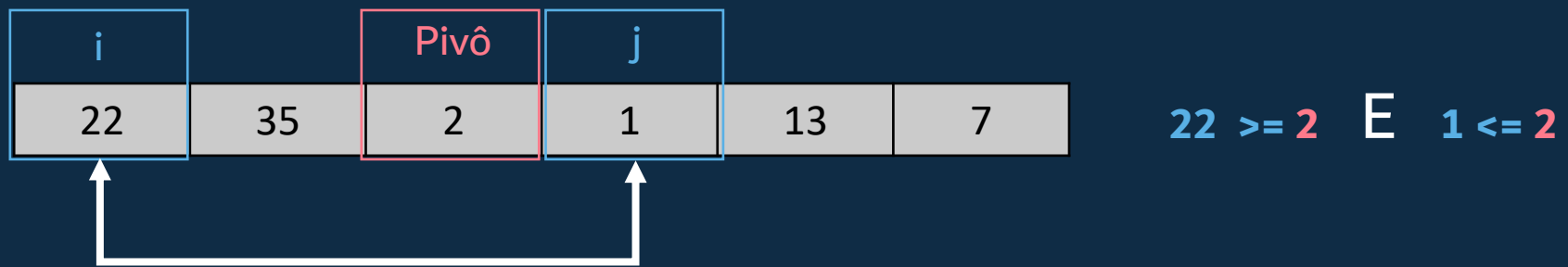
$22 \geq 2$ E ~~$7 \leq 2$~~



$22 \geq 2$ E ~~$13 \leq 2$~~

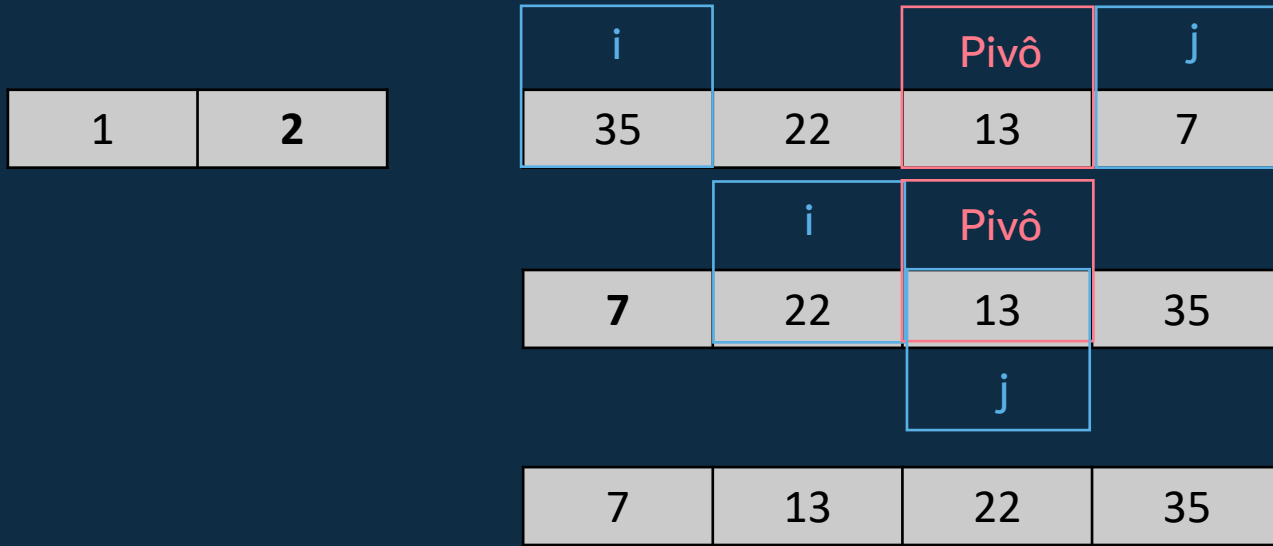


Quicksort - Funcionamento





Quicksort - Funcionamento



$35 \geq 13$ E $7 \leq 13$



Array Ordenado

1	2	7	13	22	35
---	---	---	----	----	----



Quicksort

Um algoritmo de ordenação rápido que utiliza a técnica de **Dividir e Conquistar**. Usa um pivô como referência.

Vantagens: Rápido na prática para a maioria dos casos; boa eficiência média.

Desvantagens: Pode ter desempenho ruim no pior caso; uso recursivo profundo.

Indicação: Ótimo para arrays grandes onde a média de tempo de execução é mais importante que o pior caso.

Complexidade: $O(n \log n)$ melhor caso

Estável: não

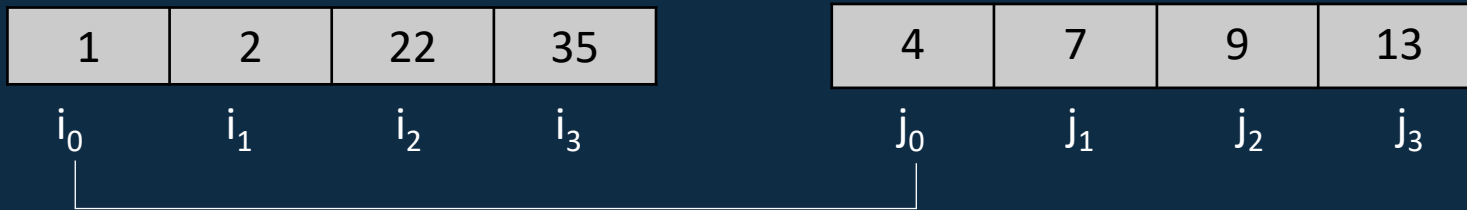


Também utiliza a técnica de **Dividir e Conquistar**, porém não possui um pivô de referência. Em vez disso, divide o array em sub-arrays e depois mescla-os de volta de maneira ordenada.



Mergesort - Funcionamento

Parte da ideia de fazer “merge” de duas listas já ordenadas.





Mergesort - Funcionamento

Parte da ideia de fazer “merge” de duas listas já ordenadas





Mergesort - Funcionamento

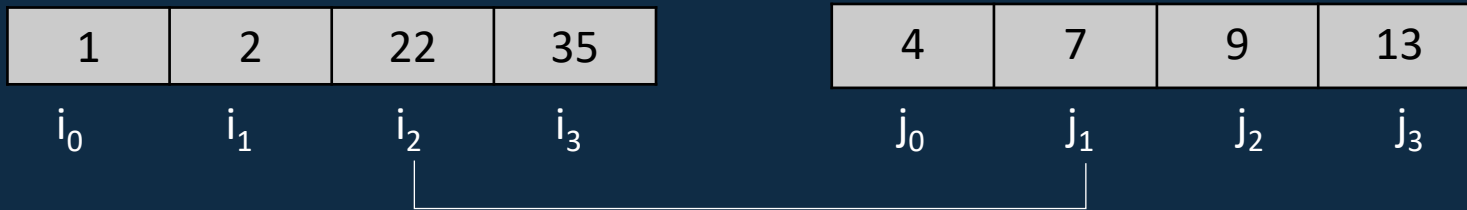
Parte da ideia de fazer “merge” de duas listas já ordenadas





Mergesort - Funcionamento

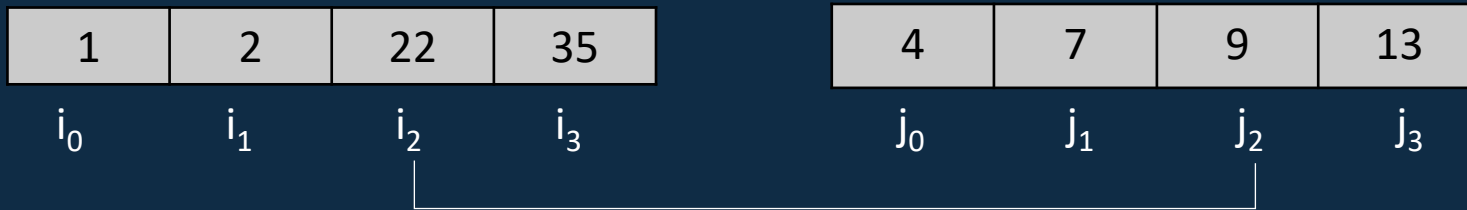
Parte da ideia de fazer “merge” de duas listas já ordenadas





Mergesort - Funcionamento

Parte da ideia de fazer “merge” de duas listas já ordenadas





Mergesort - Funcionamento

Parte da ideia de fazer “merge” de duas listas já ordenadas



1	2	4	7	9	13		
---	---	---	---	---	----	--	--



Mergesort - Funcionamento

Parte da ideia de fazer “merge” de duas listas já ordenadas

1	2	22	35
i_0	i_1	i_2	i_3

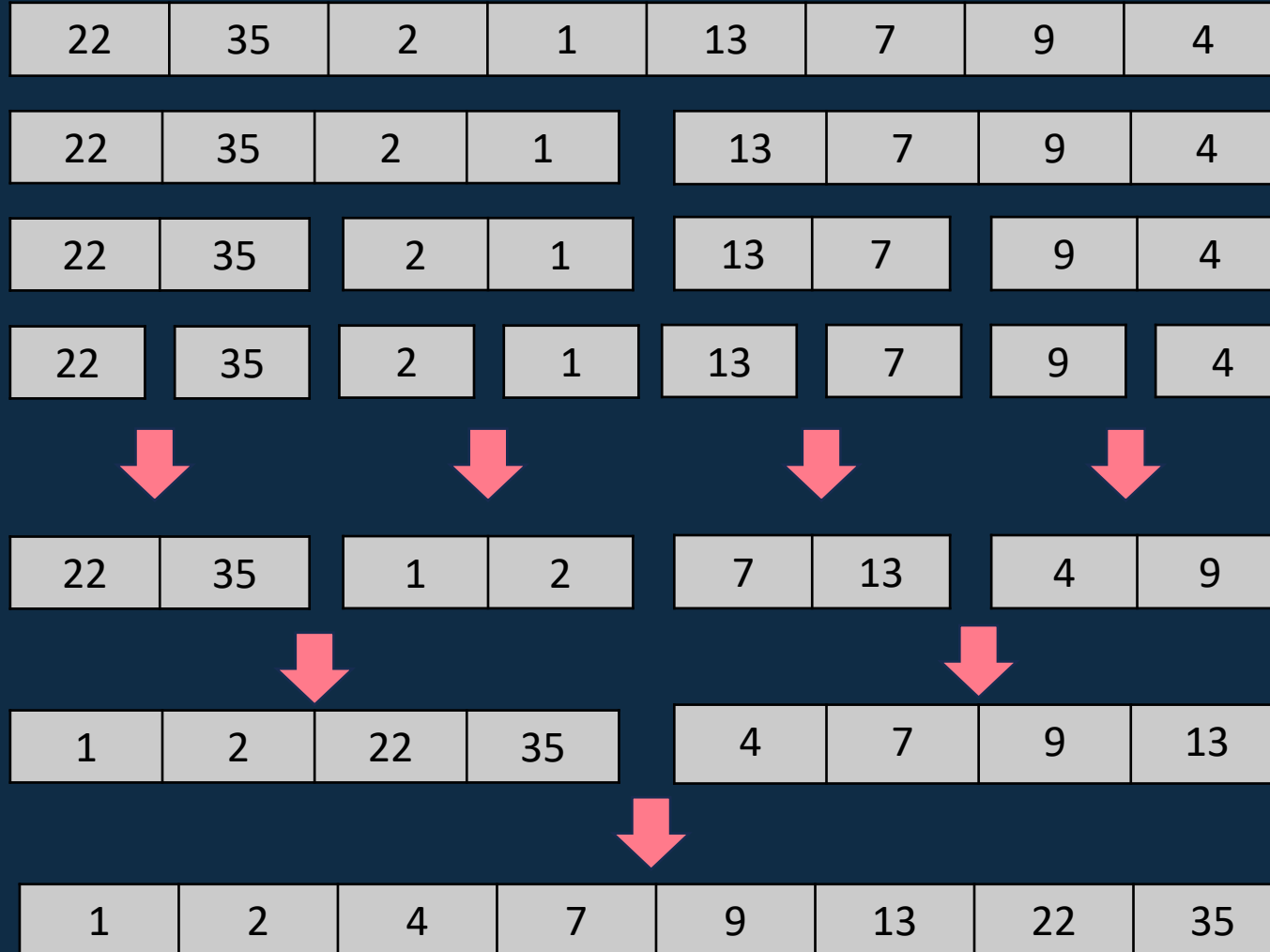
4	7	9	13
j_0	j_1	j_2	j_3

1	2	4	7	9	13	22	35
---	---	---	---	---	----	----	----



AZ Mergesort - Funcionamento

Array =





Mergesort

Também utiliza a técnica de **Dividir e Conquistar**, porém não possui um pivô de referência. Em vez disso, divide o array em sub-arrays e depois mescla-os de volta de maneira ordenada.

Vantagens: Estável e com um tempo de execução garantido $O(n \log n)$.

Desvantagens: Mais consumo de memória devido à necessidade de armazenamento temporário.

Indicação: Ideal para aplicações que precisam de estabilidade e podem lidar com o uso extra de memória.

Complexidade: $O(n \log n)$

Estável: sim

Heapsort



Ordena arrays utilizando uma estrutura de dados chamada **heap**. Intuitivamente, sua estrutura se assemelha a uma estrutura de árvore.



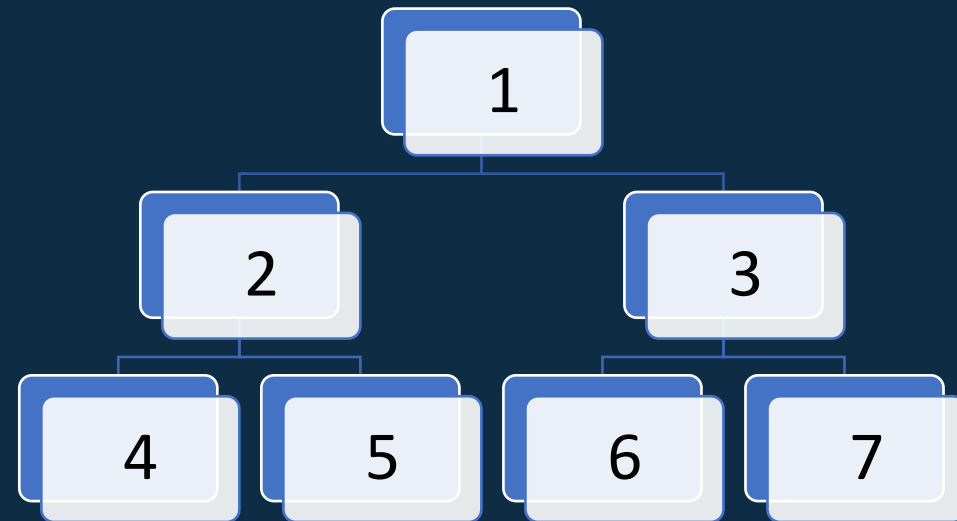
Heapsort - Funcionamento

	i_1	i_2	i_3	i_4	i_5	i_6	i_7
Array =	22	35	2	1	13	7	9

Tem como ideia que cada **i -ésimo** elemento da lista é pai dos elementos $2i$ e $2i + 1$.

Por exemplo, o elemento 1 é pai dos elementos ($2*1=2$ e $2*1+1 = 3$)

Essa lógica cria uma hierarquia de árvore entre os elementos:





AZ Heapsort - Funcionamento

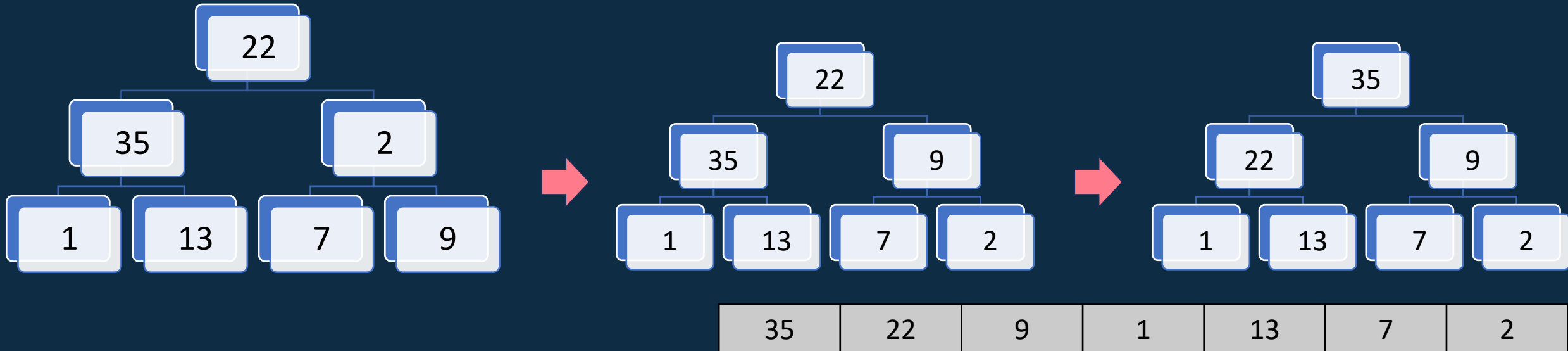
Array =

i_1	i_2	i_3	i_4	i_5	i_6	i_7
22	35	2	1	13	7	9

Inserindo os números:

Então, os passos para ordenação da lista são:

- 1 – construir a heap (hierarquia)
- 2 - atualizar heap (nó raiz > nó filhos)





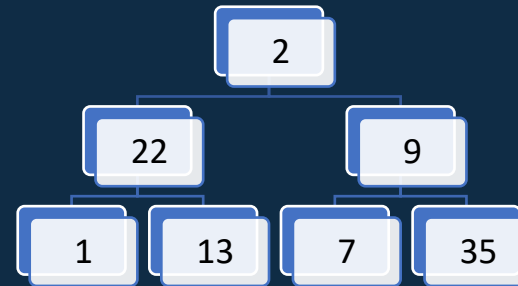
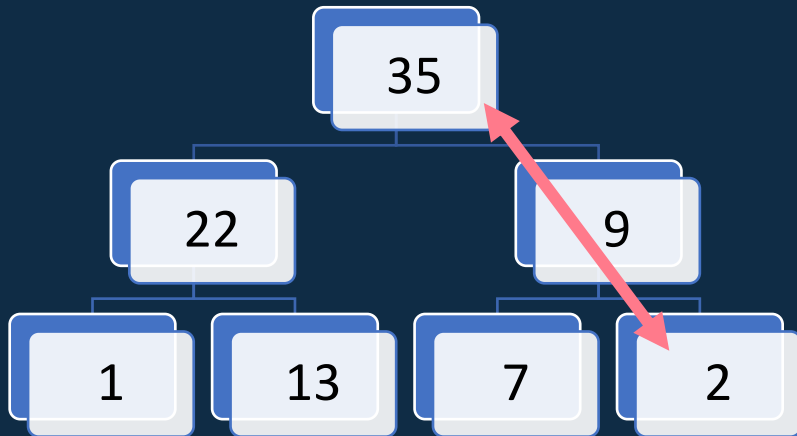
Heapsort - Funcionamento

Array =

i_1	i_2	i_3	i_4	i_5	i_6	i_7
35	22	9	1	13	7	2

Então, os passos para ordenação da lista são:

- 1 – construir a heap (hierarquia)
- 2 - atualizar heap (nó raiz > nó filhos)
- 3 – trocar o nó raiz com o último elemento do array



Neste momento, a última posição do array está em sua posição correta

2	22	9	1	13	7	35
---	----	---	---	----	---	----



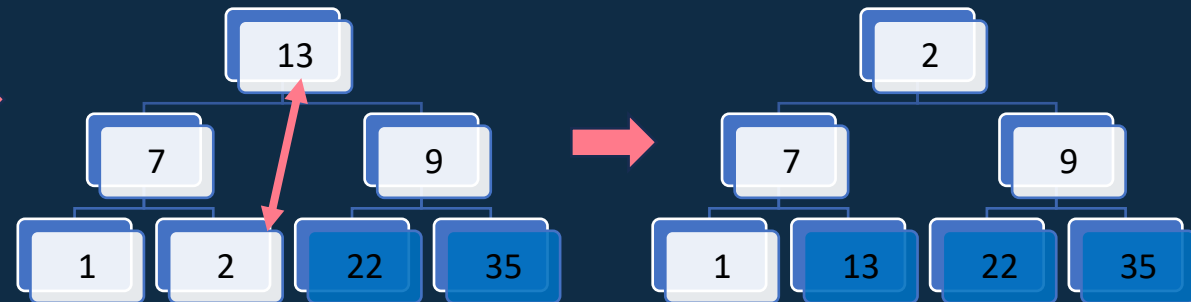
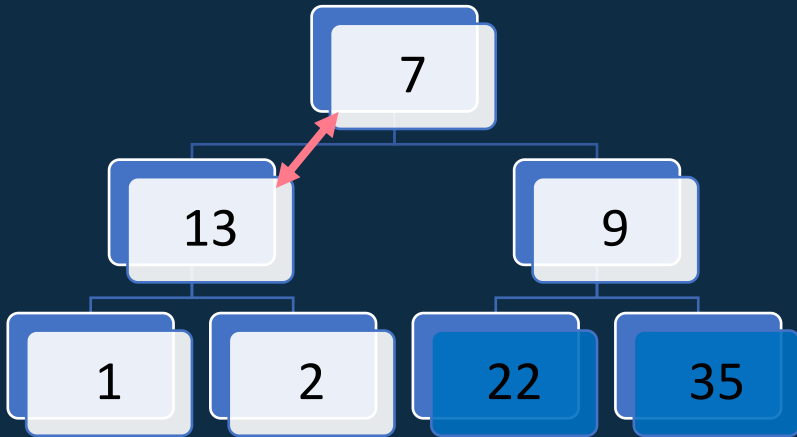
AZ Heapsort - Funcionamento

Array =

i_1	i_2	i_3	i_4	i_5	i_6	i_7
7	13	9	1	2	22	35

Então, os passos para ordenação da lista são:

- 1 – construir a heap (hierarquia)
- 2 - atualizar heap (nó raiz > nó filhos)
- 3 – trocar o nó raiz com o último elemento do array
- 4 – Repetir 2 e 3 até ordenar toda a lista



2	7	9	1	13	22	35
---	---	---	---	----	----	----

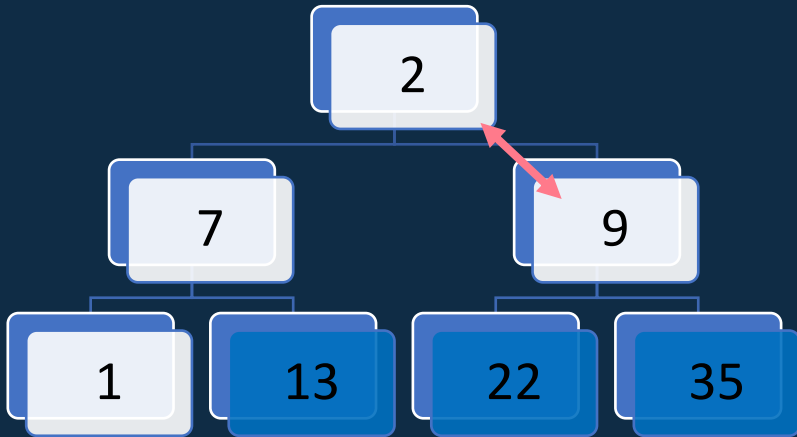


Heapsort - Funcionamento

	i_1	i_2	i_3	i_4	i_5	i_6	i_7
Array =	2	7	9	1	13	22	35

Então, os passos para ordenação da lista são:

- 1 – construir a heap (hierarquia)
- 2 - atualizar heap (nó raiz > nó filhos)
- 3 – trocar o nó raiz com o último elemento do array
- 4 – Repetir 2 e 3 até ordenar toda a lista



1	7	2	9	13	7	35
---	---	---	---	----	---	----

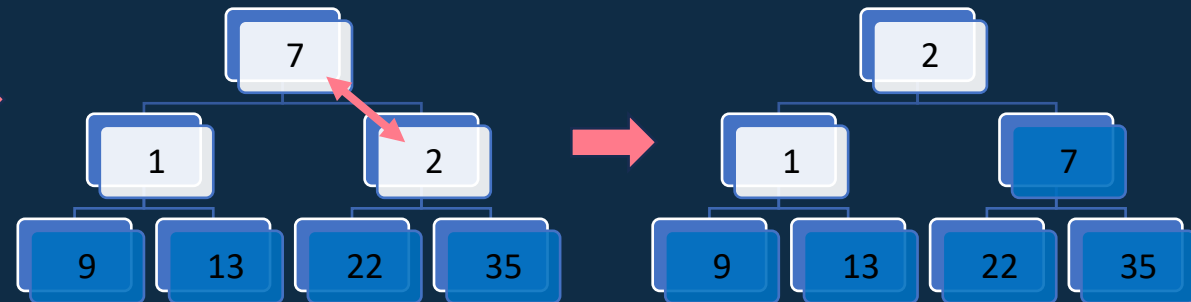
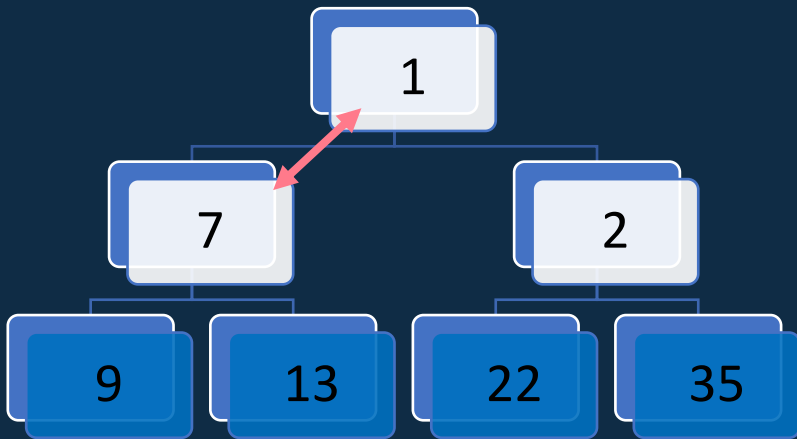


AZ Heapsort - Funcionamento

	i_1	i_2	i_3	i_4	i_5	i_6	i_7
Array =	1	7	2	9	13	22	35

Então, os passos para ordenação da lista são:

- 1 – construir a heap (hierarquia)
- 2 - atualizar heap (nó raiz > nó filhos)
- 3 – trocar o nó raiz com o último elemento do array
- 4 – Repetir 2 e 3 até ordenar toda a lista



2	1	7	9	13	7	35
---	---	---	---	----	---	----



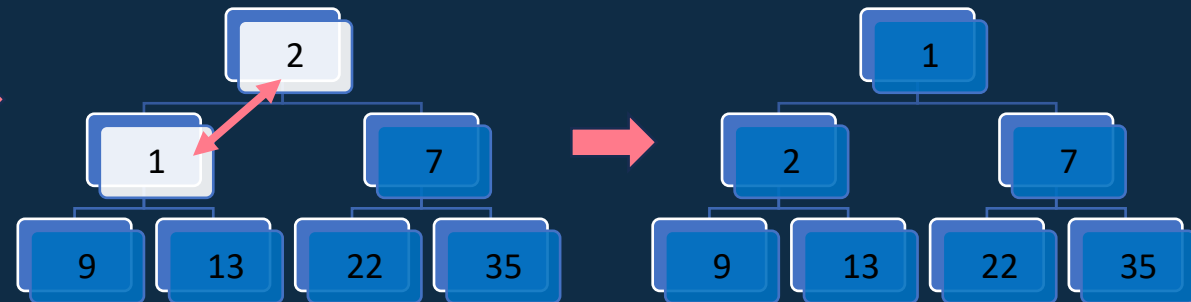
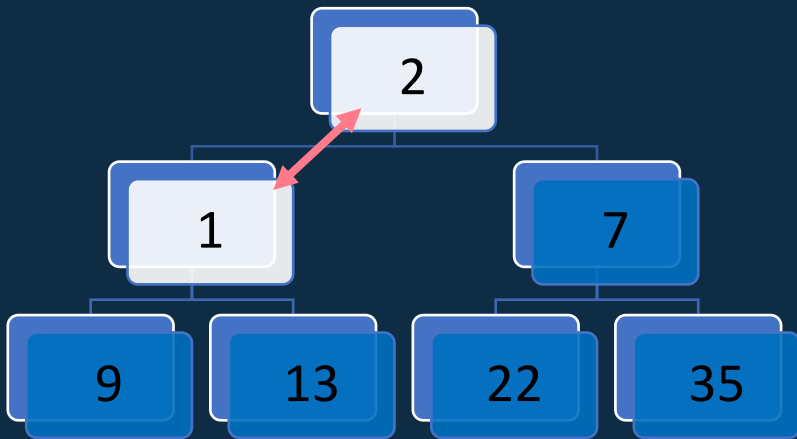
AZ Heapsort - Funcionamento

Array =

i_1	i_2	i_3	i_4	i_5	i_6	i_7
2	1	7	9	13	22	35

Então, os passos para ordenação da lista são:

- 1 – construir a heap (hierarquia)
- 2 - atualizar heap (nó raiz > nó filhos)
- 3 – trocar o nó raiz com o último elemento do array
- 4 – Repetir 2 e 3 até ordenar toda a lista



1	2	7	9	13	7	35
---	---	---	---	----	---	----



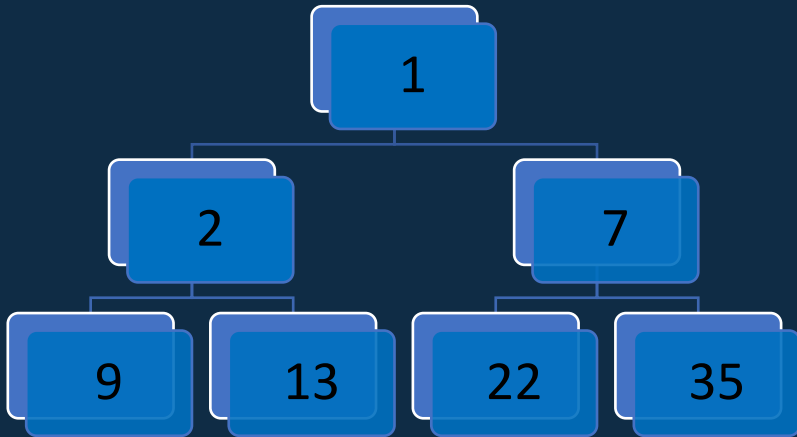
Heapsort - Funcionamento

Array =

i_1	i_2	i_3	i_4	i_5	i_6	i_7
1	2	7	9	13	22	35

Então, os passos para ordenação da lista são:

- 1 – construir a heap (hierarquia)
- 2 - atualizar heap (nó raiz > nó filhos)
- 3 – trocar o nó raiz com o último elemento do array
- 4 – Repetir 2 e 3 até ordenar toda a lista





Heapsort

Ordena arrays utilizando uma estrutura de dados chamada **heap**. Intuitivamente, sua estrutura se assemelha a uma estrutura de árvore.

Vantagens: Tempo de execução garantido $O(n \log n)$ e não requer memória extra significativa.

Desvantagens: Não é estável e geralmente mais lento em prática do que QuickSort.

Indicação: Útil quando a estabilidade não é uma preocupação e quando a memória é limitada.

Complexidade: $O(n \log n)$

Estável: não



Considerações sobre implementação do numpy




`kind{'quicksort', 'mergesort', 'heapsort', 'stable'}, optional`

Sorting algorithm. The default is 'quicksort'. Note that both 'stable' and 'mergesort' use timsort or radix sort under the covers and, in general, the actual implementation will vary with data type. The 'mergesort' option is retained for backwards compatibility.

Teste em vetor aleatório de 10.000 elementos

```
np.random.seed(0)  
vetor = np.random.randint(1,10000, 10000)
```

	Quicksort	Mergesort	Heapsort
Implementação básica	52.1 ms	71.7 ms	132 ms
 NumPy	99.4 μ s	599 μ s	898 μ s



Otimização do numpy



`kind{'quicksort', 'mergesort', 'heapsort', 'stable'}, optional`

Sorting algorithm. The default is 'quicksort'. Note that both 'stable' and 'mergesort' use timsort or radix sort under the covers and, in general, the actual implementation will vary with data type. The 'mergesort' option is retained for backwards compatibility.

Teste em vetor aleatório de 10.000 elementos

```
np.random.seed(0)  
vetor = np.random.randint(1,10000, 10000)
```

Quicksort

Mergesort

Heapsort

Implementação
básica

52.1 ms

71.7 ms

132 ms



99.4 μ s

599 μ s

898 μ s



Considerações finais



- O método sort da biblioteca numpy possui diversas configurações e algoritmos em sua implementação;
- A escolha do algoritmo depende de vários fatores, como estabilidade, estado inicial da lista etc.;
- Bibliotecas prontas possuem otimização que melhoram muito a performance do algoritmo.



Referências

- Documentação NumPy – sort:
(<https://numpy.org/doc/stable/reference/generated/numpy.sort.html>)
- "Introduction to Algorithms" por Cormen, Leiserson, Rivest e Stein.
- "Python for Data Analysis" por Wes McKinney.



Obrigado!

youtube.com/@Tech_dados

linkedin.com/in/itallo-dias/