

# RECURSÃO E ORDENAÇÃO

MC102 - Algoritmos e  
Programação de  
Computadores

Santiago Valdés Ravelo  
[https://ic.unicamp.br/~santiago/  
ravelo@unicamp.br](https://ic.unicamp.br/~santiago/ravelo@unicamp.br)

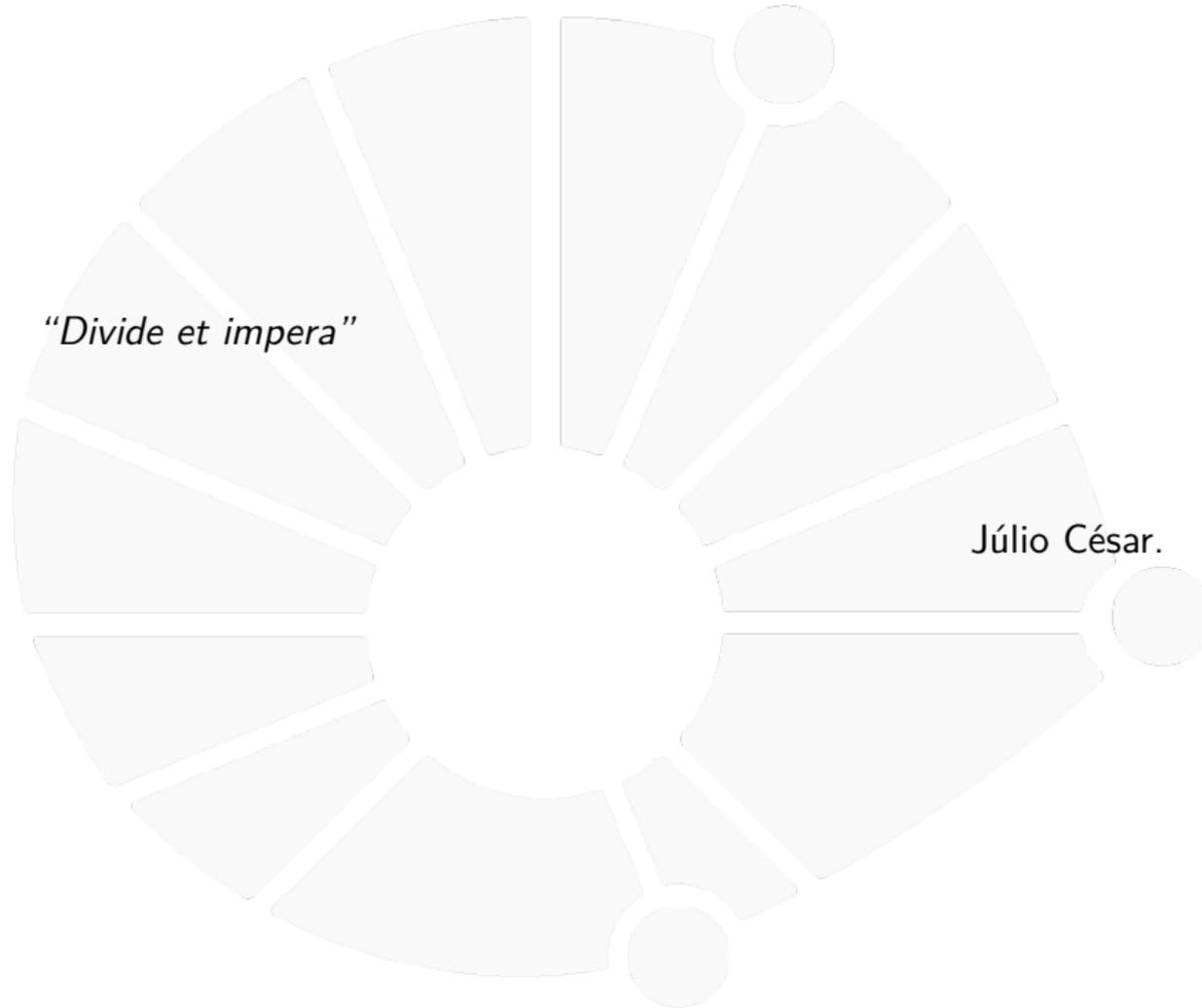
06/25

24



UNICAMP



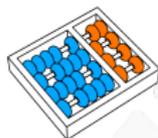
A circular diagram consisting of 16 light gray segments arranged in a ring around a central white circle. The segments are separated by white gaps. Three small light gray circles are positioned at the top, right, and bottom of the ring. The text "Divide et impera" is written in the top-left segment, and "Júlio César." is written in the right segment.

*"Divide et impera"*

Júlio César.

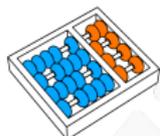


# DÚVIDAS DA AULA ANTERIOR



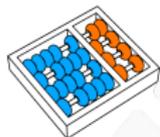
## Dúvidas selecionadas

- ▶ Tem como fazer duas funções recursivamente chamando uma à outra?



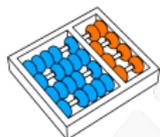
## Dúvidas selecionadas

- ▶ Tem como fazer duas funções recursivamente chamando uma à outra?
- ▶ Não entendi pq o problema  $3n+1$  é considerado um mau caso de recursão. Mesmo que a conjectura de collatz ainda não tenha sido demonstrada, existe algo de errado em uma função recursiva para números pequenos?



## Dúvidas selecionadas

- ▶ Tem como fazer duas funções recursivamente chamando uma à outra?
- ▶ Não entendi pq o problema  $3n+1$  é considerado um mau caso de recursão. Mesmo que a conjectura de collatz ainda não tenha sido demonstrada, existe algo de errado em uma função recursiva para números pequenos?
- ▶ Qual é o limite, no python, de chamadas recursivas que eu posso fazer?



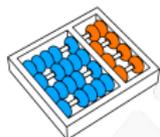
## Dúvidas selecionadas

- ▶ Tem como fazer duas funções recursivamente chamando uma à outra?
- ▶ Não entendi pq o problema  $3n+1$  é considerado um mau caso de recursão. Mesmo que a conjectura de collatz ainda não tenha sido demonstrada, existe algo de errado em uma função recursiva para números pequenos?
- ▶ Qual é o limite, no python, de chamadas recursivas que eu posso fazer?
- ▶ O exemplo da recursão para o fibonacci é mais eficiente que uma forma convencional para calcular?



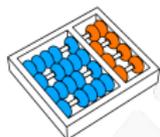
## Dúvidas selecionadas

- ▶ Tem como fazer duas funções recursivamente chamando uma à outra?
- ▶ Não entendi pq o problema  $3n+1$  é considerado um mau caso de recursão. Mesmo que a conjectura de collatz ainda não tenha sido demonstrada, existe algo de errado em uma função recursiva para números pequenos?
- ▶ Qual é o limite, no python, de chamadas recursivas que eu posso fazer?
- ▶ O exemplo da recursão para o fibonacci é mais eficiente que uma forma convencional para calcular?
- ▶ É possível adotar dois casos bases em uma recursão?



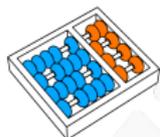
## Dúvidas selecionadas

- ▶ Tem como fazer duas funções recursivamente chamando uma à outra?
- ▶ Não entendi pq o problema  $3n+1$  é considerado um mau caso de recursão. Mesmo que a conjectura de collatz ainda não tenha sido demonstrada, existe algo de errado em uma função recursiva para números pequenos?
- ▶ Qual é o limite, no python, de chamadas recursivas que eu posso fazer?
- ▶ O exemplo da recursão para o fibonacci é mais eficiente que uma forma convencional para calcular?
- ▶ É possível adotar dois casos bases em uma recursão?
- ▶ Quais situações as funções recursivas são mais eficientes que apenas usar um for na função?



## Dúvidas selecionadas

- ▶ Tem como fazer duas funções recursivamente chamando uma à outra?
- ▶ Não entendi pq o problema  $3n+1$  é considerado um mau caso de recursão. Mesmo que a conjectura de collatz ainda não tenha sido demonstrada, existe algo de errado em uma função recursiva para números pequenos?
- ▶ Qual é o limite, no python, de chamadas recursivas que eu posso fazer?
- ▶ O exemplo da recursão para o fibonacci é mais eficiente que uma forma convencional para calcular?
- ▶ É possível adotar dois casos bases em uma recursão?
- ▶ Quais situações as funções recursivas são mais eficientes que apenas usar um for na função?
- ▶ Não entendi a diferença de eficiência entre os dois exemplos de recursão com lista.



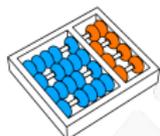
## Dúvidas selecionadas

- ▶ Tem como fazer duas funções recursivamente chamando uma à outra?
- ▶ Não entendi pq o problema  $3n+1$  é considerado um mau caso de recursão. Mesmo que a conjectura de collatz ainda não tenha sido demonstrada, existe algo de errado em uma função recursiva para números pequenos?
- ▶ Qual é o limite, no python, de chamadas recursivas que eu posso fazer?
- ▶ O exemplo da recursão para o fibonacci é mais eficiente que uma forma convencional para calcular?
- ▶ É possível adotar dois casos bases em uma recursão?
- ▶ Quais situações as funções recursivas são mais eficientes que apenas usar um for na função?
- ▶ Não entendi a diferença de eficiência entre os dois exemplos de recursão com lista.
- ▶ Quando uma função F retorna outra função H, essa função F é removida da pilha?



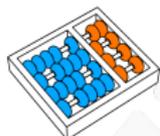
## Dúvidas selecionadas

- ▶ Tem como fazer duas funções recursivamente chamando uma à outra?
- ▶ Não entendi pq o problema  $3n+1$  é considerado um mau caso de recursão. Mesmo que a conjectura de collatz ainda não tenha sido demonstrada, existe algo de errado em uma função recursiva para números pequenos?
- ▶ Qual é o limite, no python, de chamadas recursivas que eu posso fazer?
- ▶ O exemplo da recursão para o fibonacci é mais eficiente que uma forma convencional para calcular?
- ▶ É possível adotar dois casos bases em uma recursão?
- ▶ Quais situações as funções recursivas são mais eficientes que apenas usar um for na função?
- ▶ Não entendi a diferença de eficiência entre os dois exemplos de recursão com lista.
- ▶ Quando uma função F retorna outra função H, essa função F é removida da pilha?
- ▶ Existem métodos recursivos em classes também?



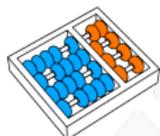
## Dúvidas selecionadas

- ▶ Tem como fazer duas funções recursivamente chamando uma à outra?
- ▶ Não entendi pq o problema  $3n+1$  é considerado um mau caso de recursão. Mesmo que a conjectura de collatz ainda não tenha sido demonstrada, existe algo de errado em uma função recursiva para números pequenos?
- ▶ Qual é o limite, no python, de chamadas recursivas que eu posso fazer?
- ▶ O exemplo da recursão para o fibonacci é mais eficiente que uma forma convencional para calcular?
- ▶ É possível adotar dois casos bases em uma recursão?
- ▶ Quais situações as funções recursivas são mais eficientes que apenas usar um for na função?
- ▶ Não entendi a diferença de eficiência entre os dois exemplos de recursão com lista.
- ▶ Quando uma função F retorna outra função H, essa função F é removida da pilha?
- ▶ Existem métodos recursivos em classes também?
- ▶ Como escolher o caso base ideal em exercícios mais complexos ao escrever uma função recursiva?



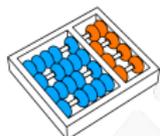
## Dúvidas selecionadas

- ▶ Tem como fazer duas funções recursivamente chamando uma à outra?
- ▶ Não entendi pq o problema  $3n+1$  é considerado um mau caso de recursão. Mesmo que a conjectura de collatz ainda não tenha sido demonstrada, existe algo de errado em uma função recursiva para números pequenos?
- ▶ Qual é o limite, no python, de chamadas recursivas que eu posso fazer?
- ▶ O exemplo da recursão para o fibonacci é mais eficiente que uma forma convencional para calcular?
- ▶ É possível adotar dois casos bases em uma recursão?
- ▶ Quais situações as funções recursivas são mais eficientes que apenas usar um for na função?
- ▶ Não entendi a diferença de eficiência entre os dois exemplos de recursão com lista.
- ▶ Quando uma função F retorna outra função H, essa função F é removida da pilha?
- ▶ Existem métodos recursivos em classes também?
- ▶ Como escolher o caso base ideal em exercícios mais complexos ao escrever uma função recursiva?
- ▶ Não entendi direito o algoritmo do Euclides.



## Dúvidas selecionadas

- ▶ Tem como fazer duas funções recursivamente chamando uma à outra?
- ▶ Não entendi pq o problema  $3n+1$  é considerado um mau caso de recursão. Mesmo que a conjectura de collatz ainda não tenha sido demonstrada, existe algo de errado em uma função recursiva para números pequenos?
- ▶ Qual é o limite, no python, de chamadas recursivas que eu posso fazer?
- ▶ O exemplo da recursão para o fibonacci é mais eficiente que uma forma convencional para calcular?
- ▶ É possível adotar dois casos bases em uma recursão?
- ▶ Quais situações as funções recursivas são mais eficientes que apenas usar um for na função?
- ▶ Não entendi a diferença de eficiência entre os dois exemplos de recursão com lista.
- ▶ Quando uma função F retorna outra função H, essa função F é removida da pilha?
- ▶ Existem métodos recursivos em classes também?
- ▶ Como escolher o caso base ideal em exercícios mais complexos ao escrever uma função recursiva?
- ▶ Não entendi direito o algoritmo do Euclides.
- ▶ Para quais tipos de input eh necessário utilizar a recursão?



### Dúvidas selecionadas

- ▶ Tem como fazer duas funções recursivamente chamando uma à outra?
- ▶ Não entendi pq o problema  $3n+1$  é considerado um mau caso de recursão. Mesmo que a conjectura de collatz ainda não tenha sido demonstrada, existe algo de errado em uma função recursiva para números pequenos?
- ▶ Qual é o limite, no python, de chamadas recursivas que eu posso fazer?
- ▶ O exemplo da recursão para o fibonacci é mais eficiente que uma forma convencional para calcular?
- ▶ É possível adotar dois casos bases em uma recursão?
- ▶ Quais situações as funções recursivas são mais eficientes que apenas usar um for na função?
- ▶ Não entendi a diferença de eficiência entre os dois exemplos de recursão com lista.
- ▶ Quando uma função F retorna outra função H, essa função F é removida da pilha?
- ▶ Existem métodos recursivos em classes também?
- ▶ Como escolher o caso base ideal em exercícios mais complexos ao escrever uma função recursiva?
- ▶ Não entendi direito o algoritmo do Euclides.
- ▶ Para quais tipos de input eh necessário utilizar a recursão?
- ▶ 

```
def duvida():  
    return "essa dúvida na verdade são duas dúvidas:\n'' + duvida() + "'' + "\n'' + duvida()  
print(duvida())
```

A large, light gray circular graphic composed of several wedge-shaped segments separated by white lines. Three small gray circles are positioned at the outer edge of the circle: one at the top, one at the bottom, and one on the right side. A solid blue horizontal bar is centered across the middle of the graphic.

LEMBRANDO



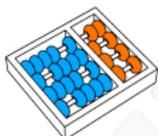
## ORDENAÇÃO

**Ordenação:** Dada uma lista  $l$  de  $n$  elementos, rearranjar os elementos de  $l$  de forma que  $l[1] \leq l[2] \leq \dots \leq l[n]$ .

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



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



## ORDENAÇÃO

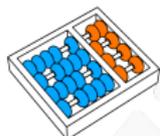
**Ordenação:** Dada uma lista  $l$  de  $n$  elementos, rearranjar os elementos de  $l$  de forma que  $l[1] \leq l[2] \leq \dots \leq l[n]$ .

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



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

Vimos três algoritmos:



## ORDENAÇÃO

**Ordenação:** Dada uma lista  $l$  de  $n$  elementos, rearranjar os elementos de  $l$  de forma que  $l[1] \leq l[2] \leq \dots \leq l[n]$ .

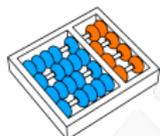
3	7	1	6	5	2	4	0	8	9
---	---	---	---	---	---	---	---	---	---



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

Vimos três algoritmos:

- ▶ **SelectionSort:** Seleciona o  $i$ -ésimo menor elemento e coloca na posição  $i$ .



## ORDENAÇÃO

**Ordenação:** Dada uma lista  $l$  de  $n$  elementos, rearranjar os elementos de  $l$  de forma que  $l[1] \leq l[2] \leq \dots \leq l[n]$ .

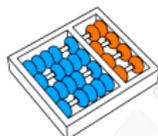
3	7	1	6	5	2	4	0	8	9
---	---	---	---	---	---	---	---	---	---



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

Vimos três algoritmos:

- ▶ **SelectionSort:** Seleciona o  $i$ -ésimo menor elemento e coloca na posição  $i$ .
- ▶ **BubbleSort:** Faz várias passagens do final para o começo trocando pares invertidos.



## ORDENAÇÃO

**Ordenação:** Dada uma lista  $l$  de  $n$  elementos, rearranjar os elementos de  $l$  de forma que  $l[1] \leq l[2] \leq \dots \leq l[n]$ .

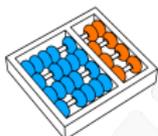
3	7	1	6	5	2	4	0	8	9
---	---	---	---	---	---	---	---	---	---



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

Vimos três algoritmos:

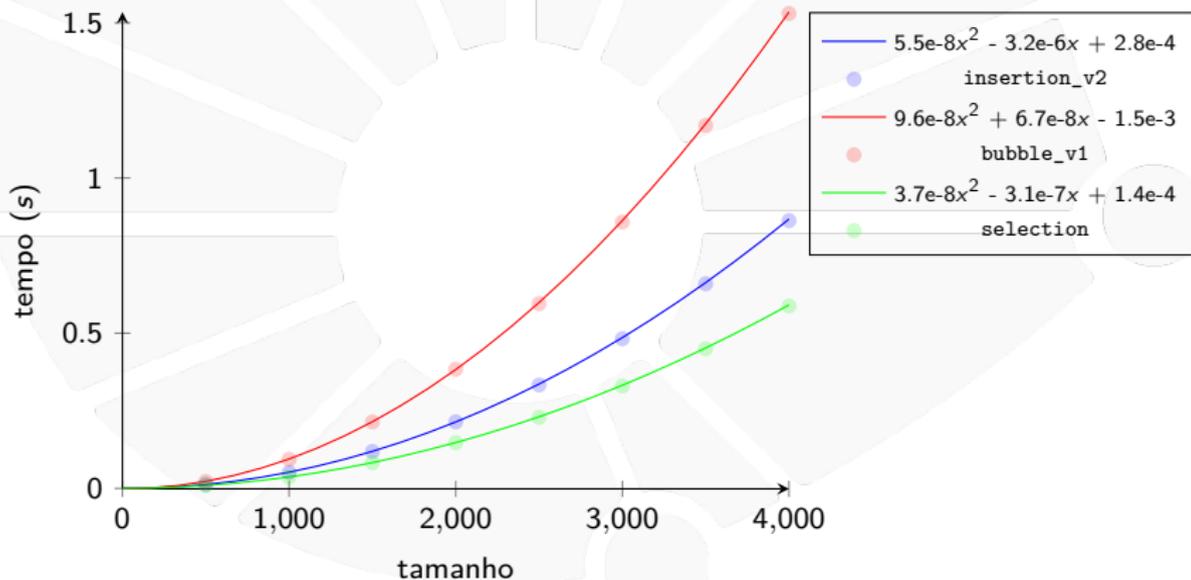
- ▶ **SelectionSort:** Seleciona o  $i$ -ésimo menor elemento e coloca na posição  $i$ .
- ▶ **BubbleSort:** Faz varias passagens do final para o começo trocando pares invertidos.
- ▶ **InsertionSort:** Insere o  $i$ -ésimo elemento na posição correta.



## Experimento

Tempo cresce **quadraticamente** com o tamanho da lista:

- ▶ Listas de tamanho 100, 200, ..., 4000, com elementos aleatórios entre 0 e 1.
- ▶ Tiramos a média do tempo de 10 execuções.



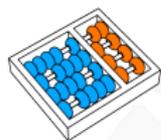


# ORDENAÇÃO RECURSIVA



## Outros algoritmos

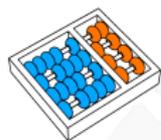
Na aula de hoje veremos:



## Outros algoritmos

Na aula de hoje veremos:

- ▶ Dois outros algoritmos de ordenação.



## Outros algoritmos

Na aula de hoje veremos:

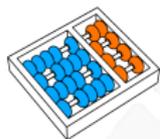
- ▶ Dois outros algoritmos de ordenação.
- ▶ Baseados em recursão.



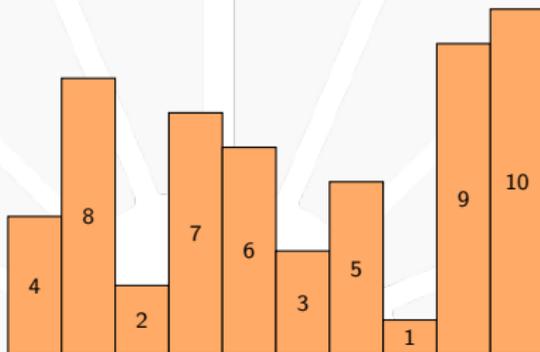
## Outros algoritmos

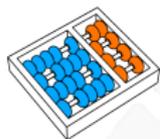
Na aula de hoje veremos:

- ▶ Dois outros algoritmos de ordenação.
- ▶ Baseados em recursão.
- ▶ Mais rápidos que os outros três.

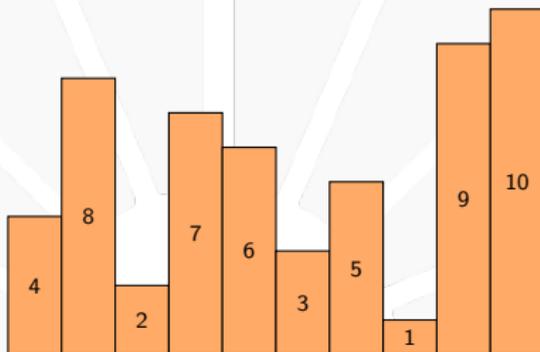


## Estratégia: Recursão:

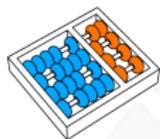




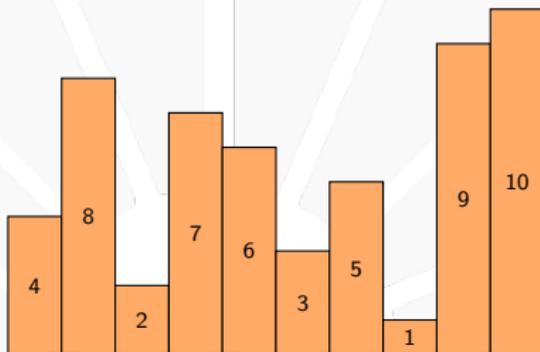
## Estratégia: Recursão:



Como ordenar a primeira metade da lista?

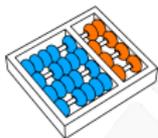


## Estratégia: Recursão:

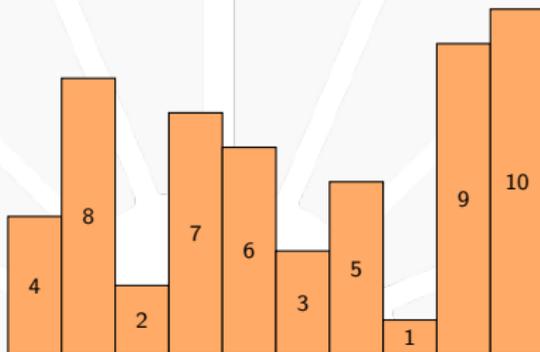


Como ordenar a primeira metade da lista?

- ▶ Usamos uma função **ordenar(l, e, d)**:

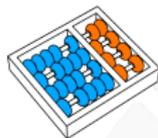


## Estratégia: Recursão:

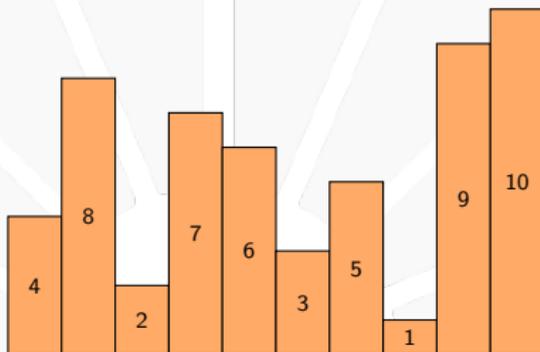


Como ordenar a primeira metade da lista?

- ▶ Usamos uma função **ordenar(l, e, d)**:
  - ▶ Ordena a lista **l** das posições **e** a **d** (inclusive).

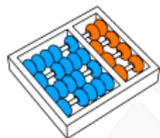


## Estratégia: Recursão:

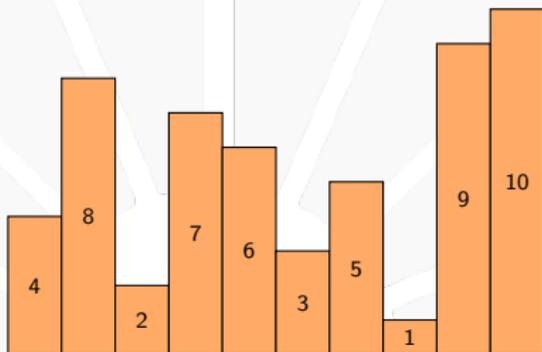


Como ordenar a primeira metade da lista?

- ▶ Usamos uma função **ordenar(l, e, d)**:
  - ▶ Ordena a lista **l** das posições **e** a **d** (inclusive).
  - ▶ Poderia ser um dos algoritmos vistos anteriormente.

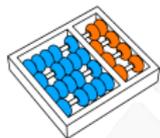


## Estratégia: Recursão:

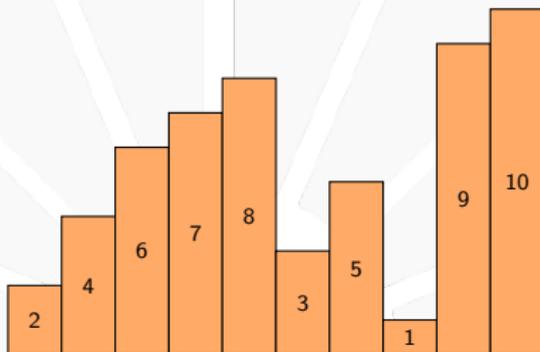


Como ordenar a primeira metade da lista?

- ▶ Usamos uma função **ordenar(l, e, d)**:
  - ▶ Ordena a lista **l** das posições **e** a **d** (inclusive).
  - ▶ Poderia ser um dos algoritmos vistos anteriormente.
  - ▶ Mas usaremos recursão aqui!

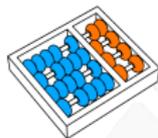


## Estratégia: Recursão:

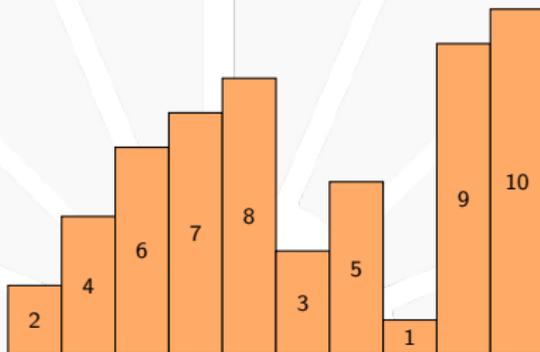


Como ordenar a primeira metade da lista?

- ▶ Usamos uma função **ordenar(l, e, d)**:
  - ▶ Ordena a lista **l** das posições **e** a **d** (inclusive).
  - ▶ Poderia ser um dos algoritmos vistos anteriormente.
  - ▶ Mas usaremos recursão aqui!
- ▶ Executamos **ordenar(l, 0, 4)**.



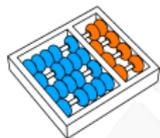
## Estratégia: Recursão:



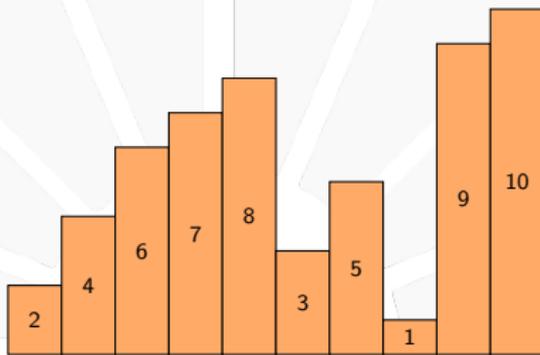
Como ordenar a primeira metade da lista?

- ▶ Usamos uma função **ordenar(l, e, d)**:
  - ▶ Ordena a lista **l** das posições **e** a **d** (inclusive).
  - ▶ Poderia ser um dos algoritmos vistos anteriormente.
  - ▶ Mas usaremos recursão aqui!
- ▶ Executamos **ordenar(l, 0, 4)**.

E se quiséssemos ordenar a segunda parte?



## Ordenando a segunda parte

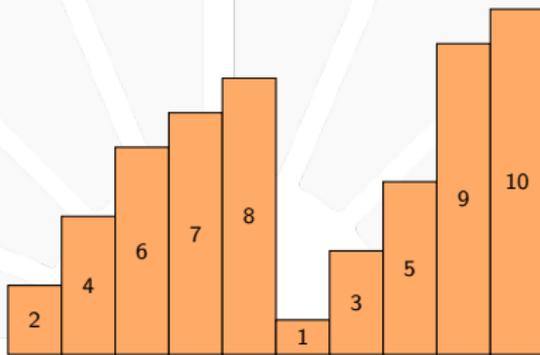


Para ordenar a segunda metade:

- ▶ Executamos `ordenar(1, 5, 9)`.

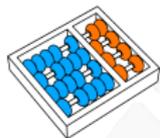


## Ordenando a segunda parte



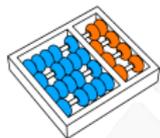
Para ordenar a segunda metade:

- ▶ Executamos `ordenar(1, 5, 9)`.



## Ordenando toda a lista

Se temos um lista com as suas duas metades já ordenadas:



## Ordenando toda a lista

Se temos um lista com as suas duas metades já ordenadas:

- ▶ Como ordenar toda a lista?



## Ordenando toda a lista

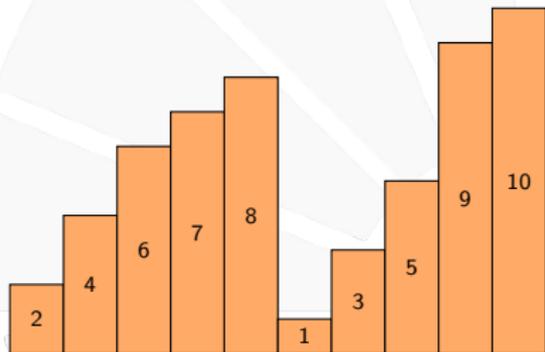
Se temos um lista com as suas duas metades já ordenadas:

- ▶ Como ordenar toda a lista?





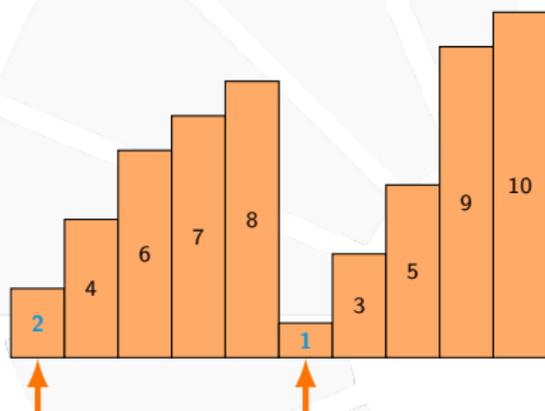
## Intercalando



- ▶ Percorreremos as duas sub-listas.



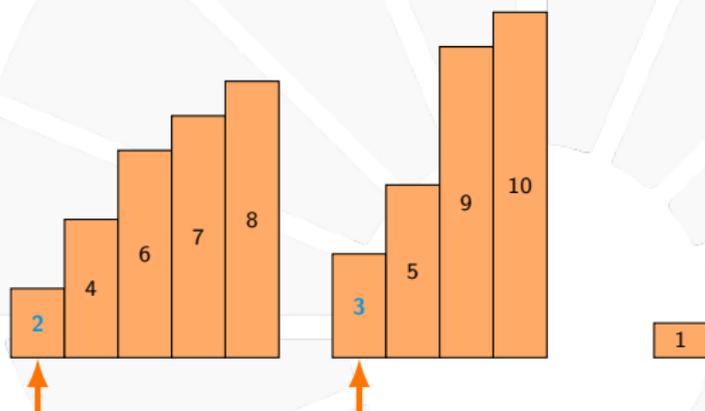
## Intercalando



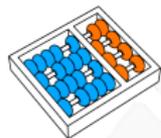
- ▶ Percorreremos as duas sub-listas.
- ▶ Pegamos o **mínimo** e inserimos em uma lista auxiliar.



## Intercalando



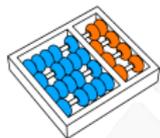
- ▶ Percorreremos as duas sub-listas.
- ▶ Pegamos o **mínimo** e inserimos em uma lista auxiliar.



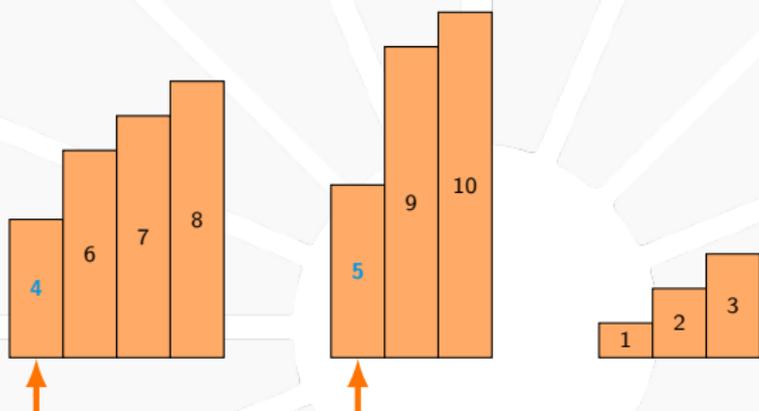
## Intercalando



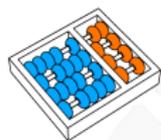
- ▶ Percorreremos as duas sub-listas.
- ▶ Pegamos o **mínimo** e inserimos em uma lista auxiliar.



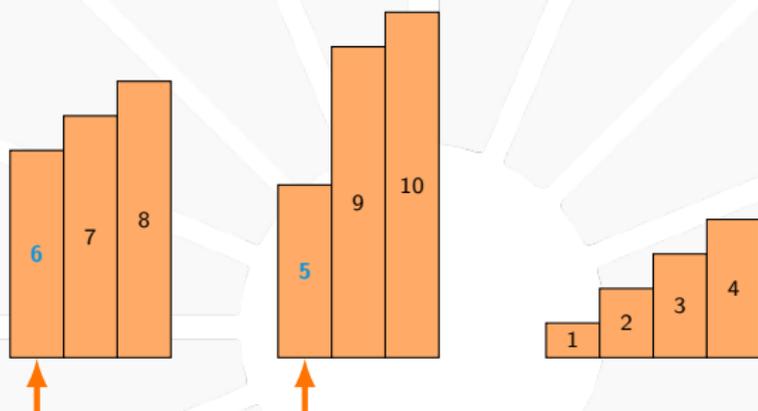
## Intercalando



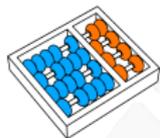
- ▶ Percorreremos as duas sub-listas.
- ▶ Pegamos o **mínimo** e inserimos em uma lista auxiliar.



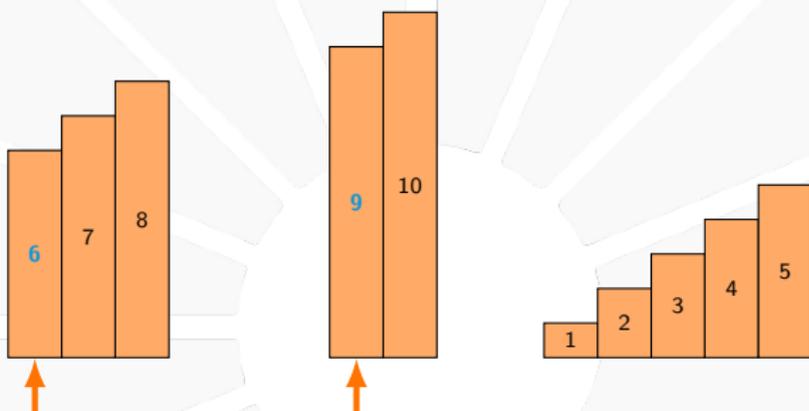
## Intercalando



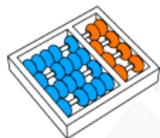
- ▶ Percorreremos as duas sub-listas.
- ▶ Pegamos o **mínimo** e inserimos em uma lista auxiliar.



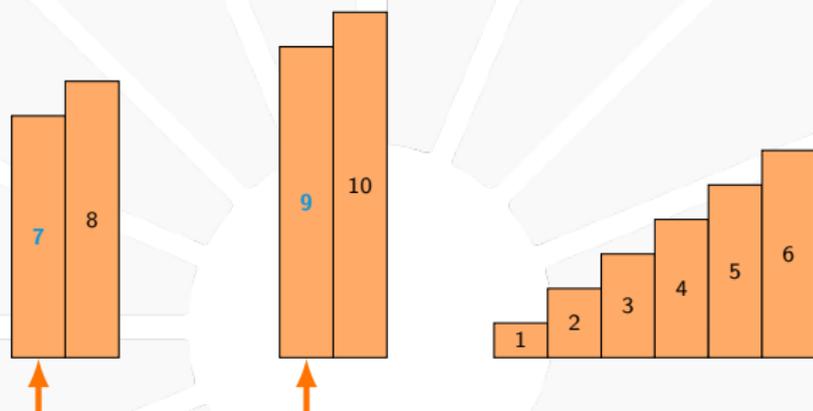
## Intercalando



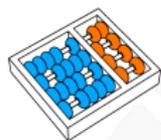
- ▶ Percorreremos as duas sub-listas.
- ▶ Pegamos o **mínimo** e inserimos em uma lista auxiliar.



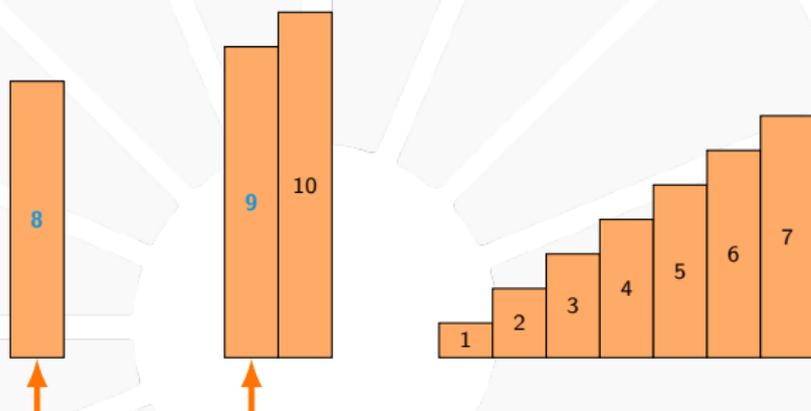
## Intercalando



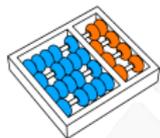
- ▶ Percorreremos as duas sub-listas.
- ▶ Pegamos o **mínimo** e inserimos em uma lista auxiliar.



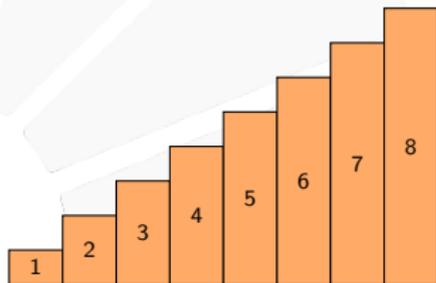
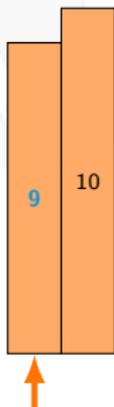
## Intercalando



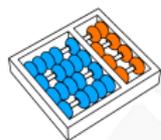
- ▶ Percorreremos as duas sub-listas.
- ▶ Pegamos o **mínimo** e inserimos em uma lista auxiliar.



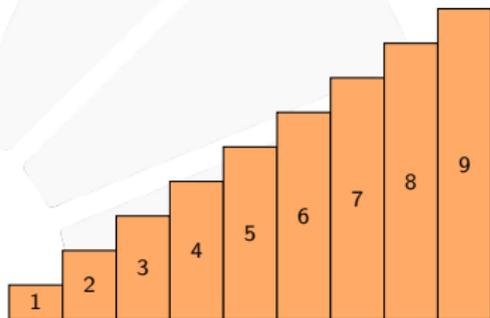
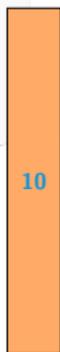
## Intercalando



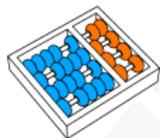
- ▶ Percorremos as duas sub-listas.
- ▶ Pegamos o **mínimo** e inserimos em uma lista auxiliar.
- ▶ Depois copiamos o restante.



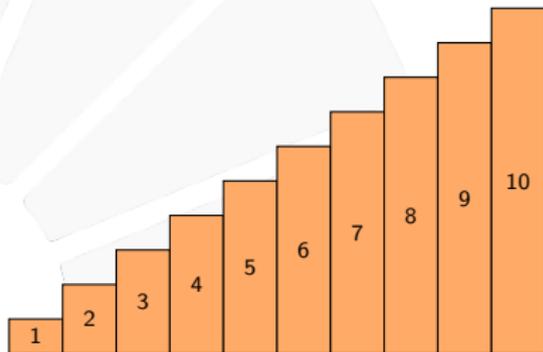
## Intercalando



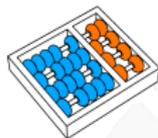
- ▶ Percorremos as duas sub-listas.
- ▶ Pegamos o **mínimo** e inserimos em uma lista auxiliar.
- ▶ Depois copiamos o restante.



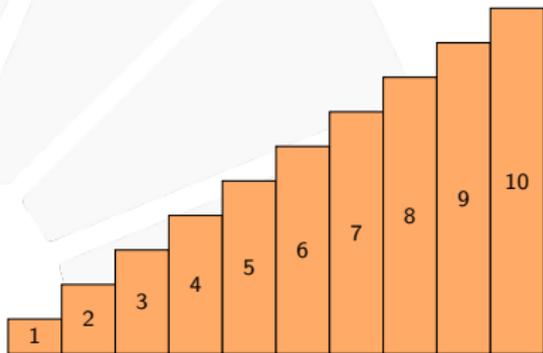
## Intercalando



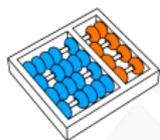
- ▶ Percorremos as duas sub-listas.
- ▶ Pegamos o **mínimo** e inserimos em uma lista auxiliar.
- ▶ Depois copiamos o restante.



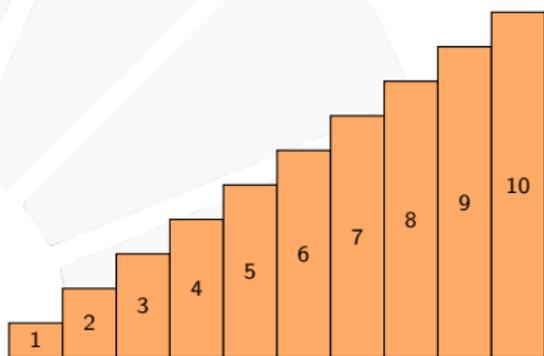
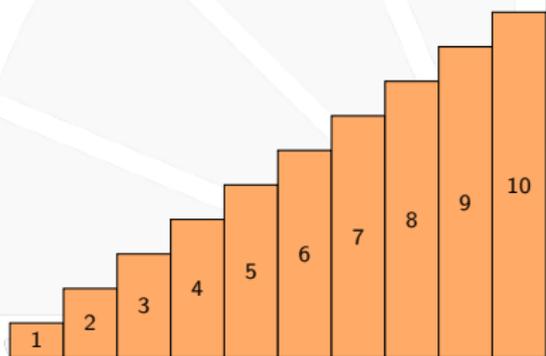
## Intercalando



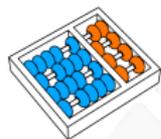
- ▶ Percorremos as duas sub-listas.
- ▶ Pegamos o **mínimo** e inserimos em uma lista auxiliar.
- ▶ Depois copiamos o restante.
- ▶ No final, copiamos da lista auxiliar para a original.



## Intercalando



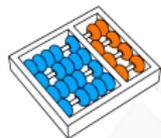
- ▶ Percorremos as duas sub-listas.
- ▶ Pegamos o **mínimo** e inserimos em uma lista auxiliar.
- ▶ Depois copiamos o restante.
- ▶ No final, copiamos da lista auxiliar para a original.



## Divisão e conquista

Ordenação recursiva

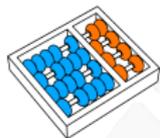
Observação:



## Divisão e conquista

Observação:

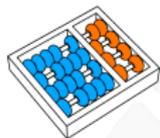
- ▶ A recursão parte do princípio que é mais fácil resolver problemas menores.



## Divisão e conquista

Observação:

- ▶ A recursão parte do princípio que é mais fácil resolver problemas menores.
- ▶ Para certos problemas, podemos dividi-lo em duas ou mais partes.

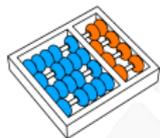


## Divisão e conquista

Observação:

- ▶ A recursão parte do princípio que é mais fácil resolver problemas menores.
- ▶ Para certos problemas, podemos dividi-lo em duas ou mais partes.

Divisão e conquista:



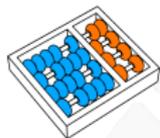
## Divisão e conquista

Observação:

- ▶ A recursão parte do princípio que é mais fácil resolver problemas menores.
- ▶ Para certos problemas, podemos dividi-lo em duas ou mais partes.

Divisão e conquista:

- ▶ **Divisão:** Quebramos o problema em vários subproblemas menores.



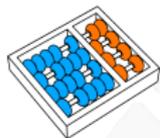
## Divisão e conquista

Observação:

- ▶ A recursão parte do princípio que é mais fácil resolver problemas menores.
- ▶ Para certos problemas, podemos dividi-lo em duas ou mais partes.

Divisão e conquista:

- ▶ **Divisão:** Quebramos o problema em vários subproblemas menores.
  - ▶ ex: quebramos uma lista a ser ordenada em duas.



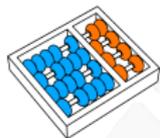
## Divisão e conquista

Observação:

- ▶ A recursão parte do princípio que é mais fácil resolver problemas menores.
- ▶ Para certos problemas, podemos dividi-lo em duas ou mais partes.

Divisão e conquista:

- ▶ **Divisão:** Quebramos o problema em vários subproblemas menores.
  - ▶ ex: quebramos uma lista a ser ordenada em duas.
- ▶ **Conquista:** Combinamos a solução dos problemas menores.



## Divisão e conquista

Observação:

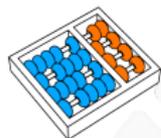
- ▶ A recursão parte do princípio que é mais fácil resolver problemas menores.
- ▶ Para certos problemas, podemos dividi-lo em duas ou mais partes.

Divisão e conquista:

- ▶ **Divisão:** Quebramos o problema em vários subproblemas menores.
  - ▶ ex: quebramos uma lista a ser ordenada em duas.
- ▶ **Conquista:** Combinamos a solução dos problemas menores.
  - ▶ ex: intercalamos as duas listas ordenadas.



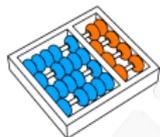
# MERGESORT



## Ordenação por intercalação (MERGESORT)

Intercalação:

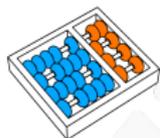




## Ordenação por intercalação (MERGESORT)

Intercalação:

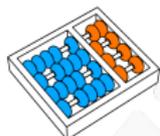
- ▶ As duas sub-listas estão armazenadas em **l**:



## Ordenação por intercalação (MERGESORT)

Intercalação:

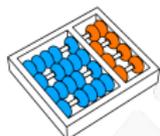
- ▶ As duas sub-listas estão armazenadas em **l**:
  - ▶ A primeira nas posições de **e** até **m**.



## Ordenação por intercalação (MERGESORT)

Intercalação:

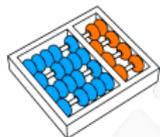
- ▶ As duas sub-listas estão armazenadas em  $l$ :
  - ▶ A primeira nas posições de  $e$  até  $m$ .
  - ▶ A segunda nas posições de  $m + 1$  até  $d$ .



## Ordenação por intercalação (MERGESORT)

Intercalação:

- ▶ As duas sub-listas estão armazenadas em  **$l$** :
  - ▶ A primeira nas posições de  **$e$**  até  **$m$** .
  - ▶ A segunda nas posições de  **$m + 1$**  até  **$d$** .
- ▶ Precisamos de uma lista auxiliar.



## Intercalação

```
1 def merge(l, e, m, d):
2     aux = []
3     i, j = e, m + 1
4     while i <= m and j <= d:
5         if l[i] <= l[j]:
6             aux.append(l[i])
7             i += 1
8         else:
9             aux.append(l[j])
10            j += 1
11     while i <= m: # Cópia o restante da primeira metade
12         aux.append(l[i])
13         i += 1
14     while j <= d: # Cópia o restante da segunda metade
15         aux.append(l[j])
16         j += 1
17     for i in range(e, d + 1): # Cópia de volta
18         l[i] = aux[i - e]
```



## Ordenação por intercalação (MERGESORT)

Ordenação:





## Ordenação por intercalação (MERGESORT)

Ordenação:

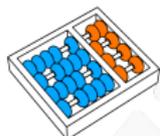
- ▶ Recebemos uma faixa da lista **l**:



## Ordenação por intercalação (MERGESORT)

Ordenação:

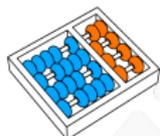
- ▶ Recebemos uma faixa da lista **l**:
- ▶ A faixa começa na posição **e**.



## Ordenação por intercalação (MERGESORT)

Ordenação:

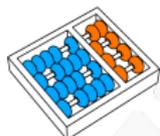
- ▶ Recebemos uma faixa da lista **l**:
  - ▶ A faixa começa na posição **e**.
  - ▶ A faixa termina na posição **d**.



## Ordenação por intercalação (MERGESORT)

Ordenação:

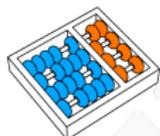
- ▶ Recebemos uma faixa da lista **l**:
  - ▶ A faixa começa na posição **e**.
  - ▶ A faixa termina na posição **d**.
- ▶ Dividimos a faixa em duas.



## Ordenação por intercalação (MERGESORT)

Ordenação:

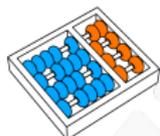
- ▶ Recebemos uma faixa da lista **l**:
  - ▶ A faixa começa na posição **e**.
  - ▶ A faixa termina na posição **d**.
- ▶ Dividimos a faixa em duas.
- ▶ O caso base é uma faixa de tamanho **0** ou **1**.



## Ordenação por intercalação (MERGESORT)

Ordenação:

- ▶ Recebemos uma faixa da lista **l**:
  - ▶ A faixa começa na posição **e**.
  - ▶ A faixa termina na posição **d**.
- ▶ Dividimos a faixa em duas.
- ▶ O caso base é uma faixa de tamanho **0** ou **1**.
  - ▶ Já está ordenada!

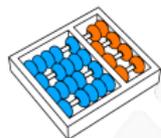


## Ordenação por intercalação (MERGESORT)

Ordenação:

- ▶ Recebemos uma faixa da lista **l**:
  - ▶ A faixa começa na posição **e**.
  - ▶ A faixa termina na posição **d**.
- ▶ Dividimos a faixa em duas.
- ▶ O caso base é uma faixa de tamanho **0** ou **1**.
  - ▶ Já está ordenada!

```
1 def mergesort(l, e, d):
2     if e < d:
3         m = (e + d) // 2
4         mergesort(l, e, m)
5         mergesort(l, m + 1, d)
6         merge(l, e, m, d)
```



## Simulação

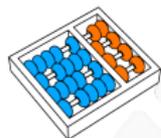
4	8	2	7	6	3	5	1	9	10
---	---	---	---	---	---	---	---	---	----



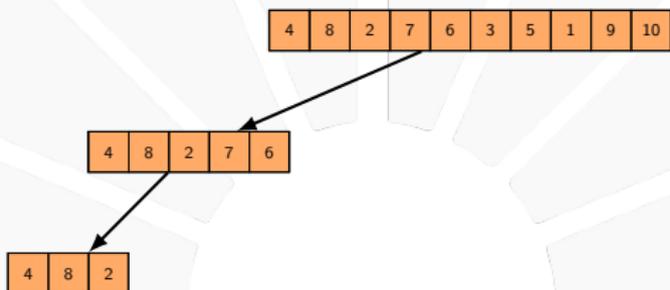
## Simulação

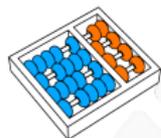
4	8	2	7	6	3	5	1	9	10
---	---	---	---	---	---	---	---	---	----

4	8	2	7	6
---	---	---	---	---

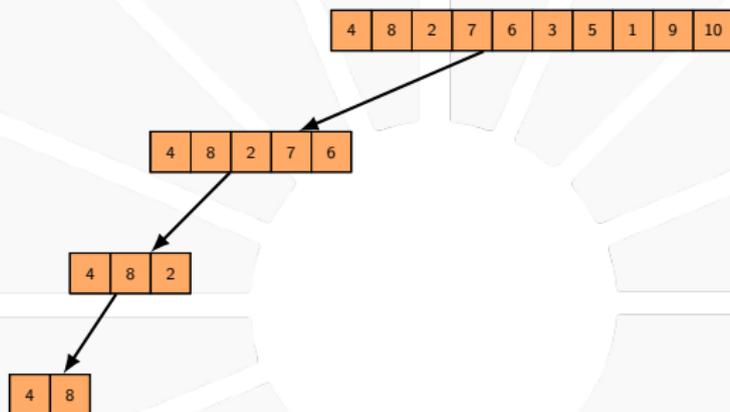


## Simulação

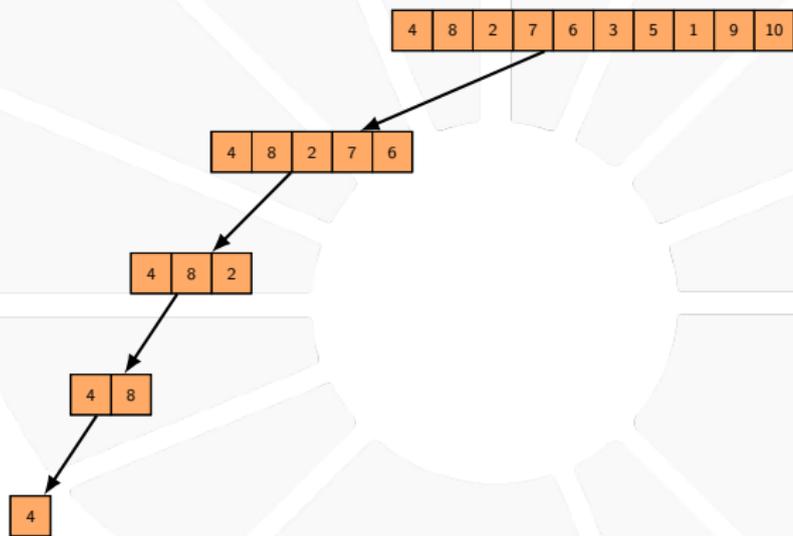
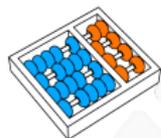




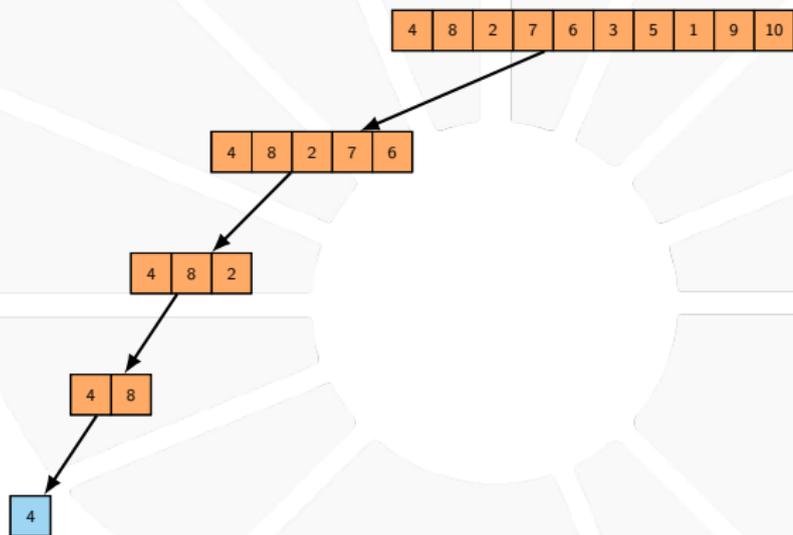
## Simulação



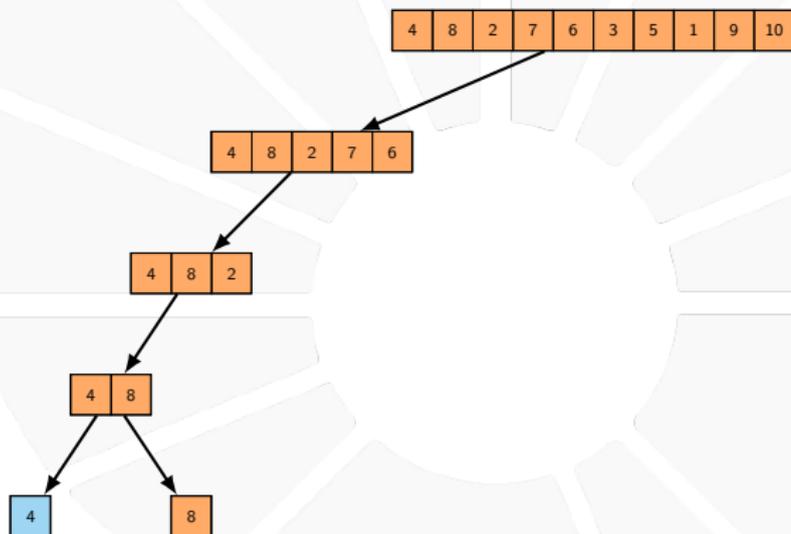
## Simulação



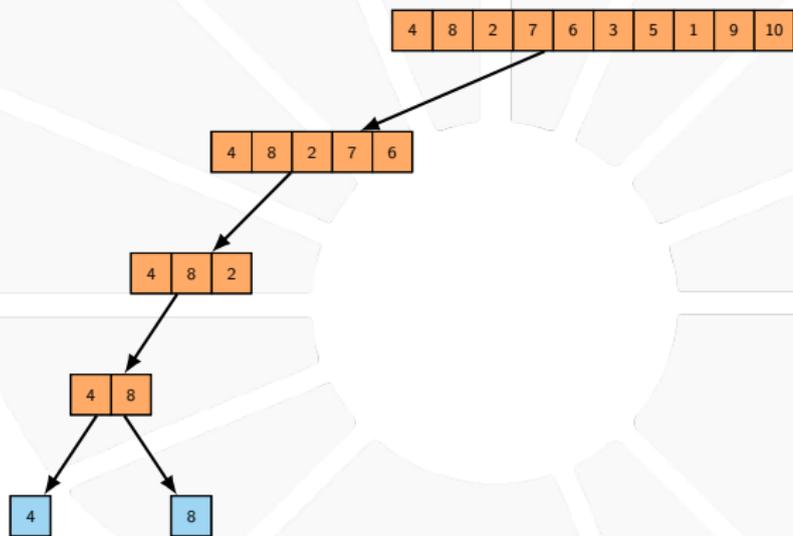
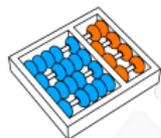
## Simulação



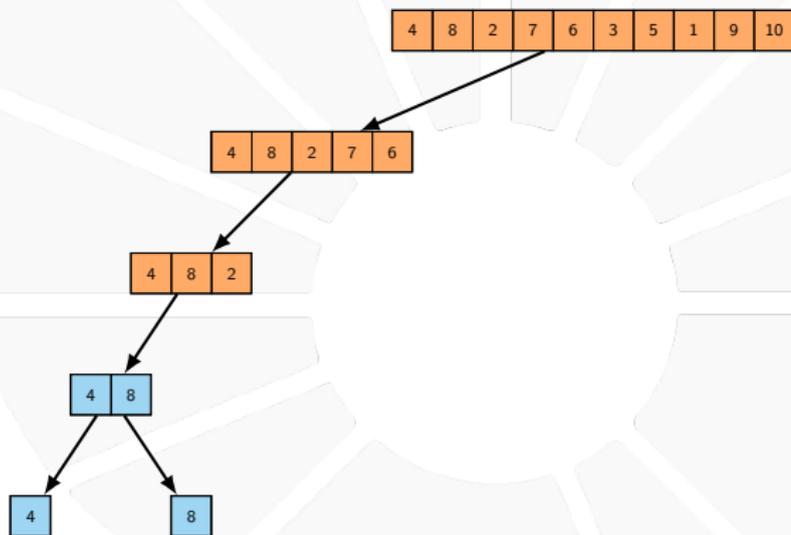
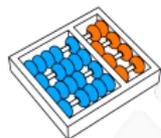
## Simulação



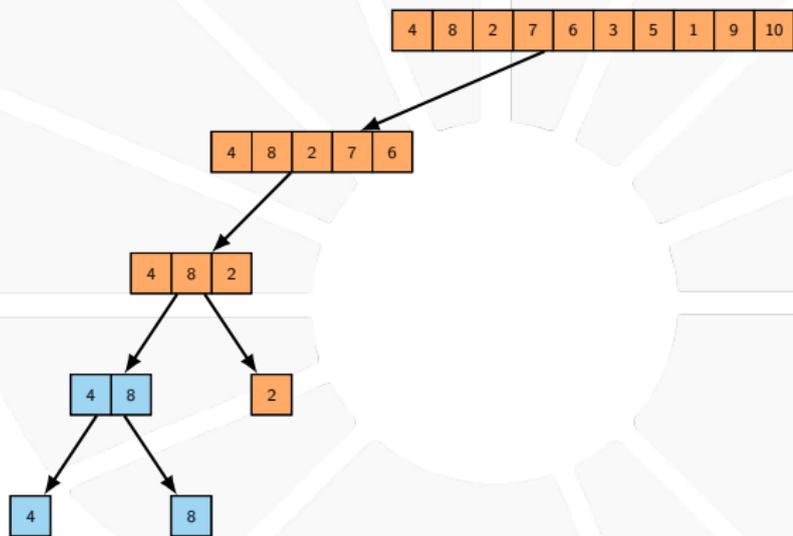
## Simulação



## Simulação

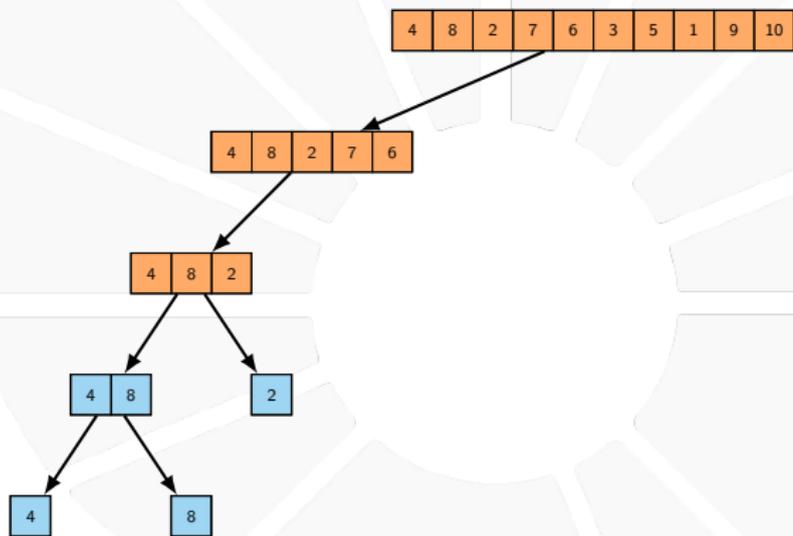


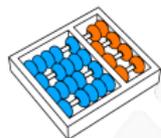
## Simulação



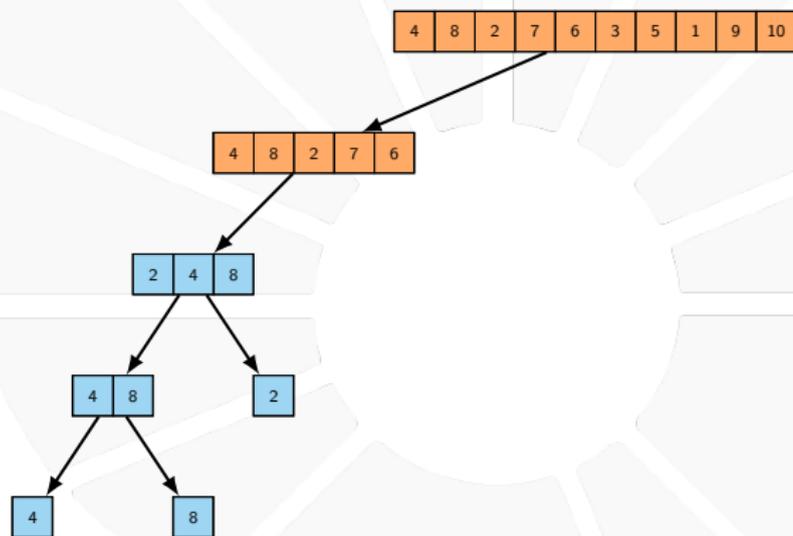


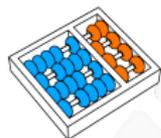
## Simulação



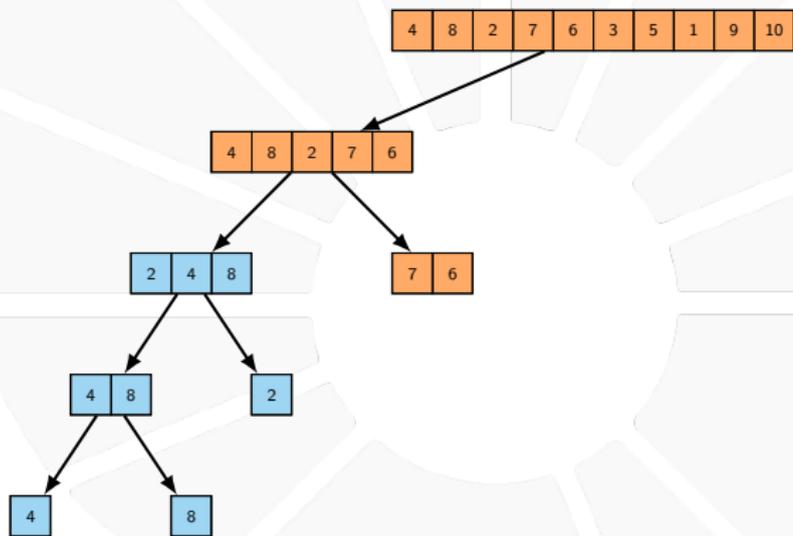


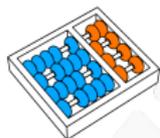
## Simulação



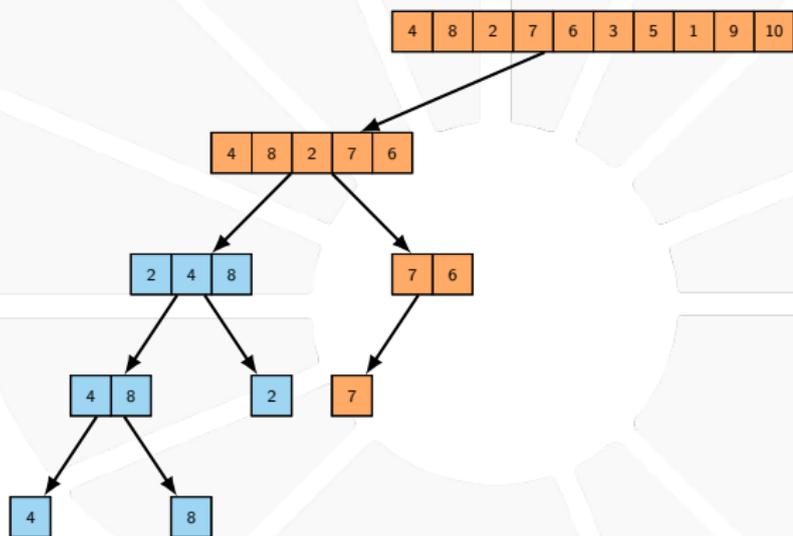


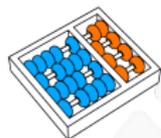
## Simulação



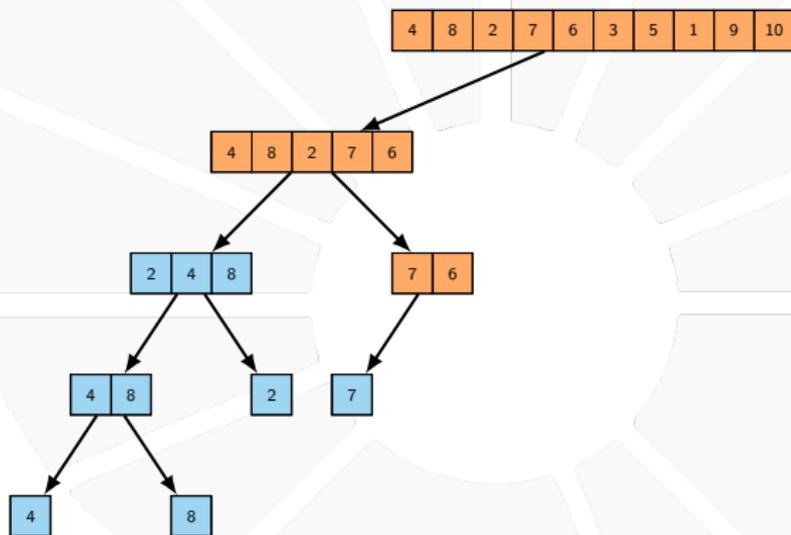


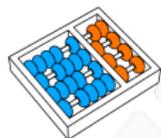
## Simulação



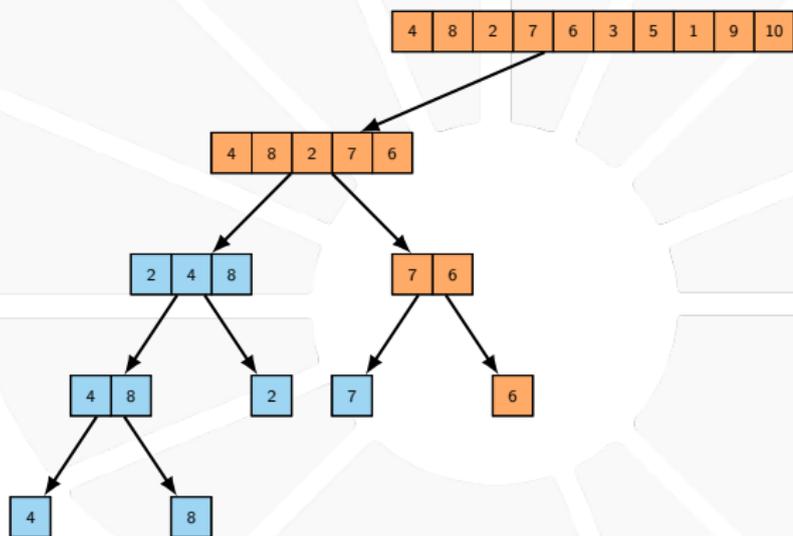


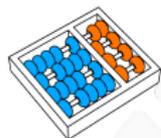
## Simulação



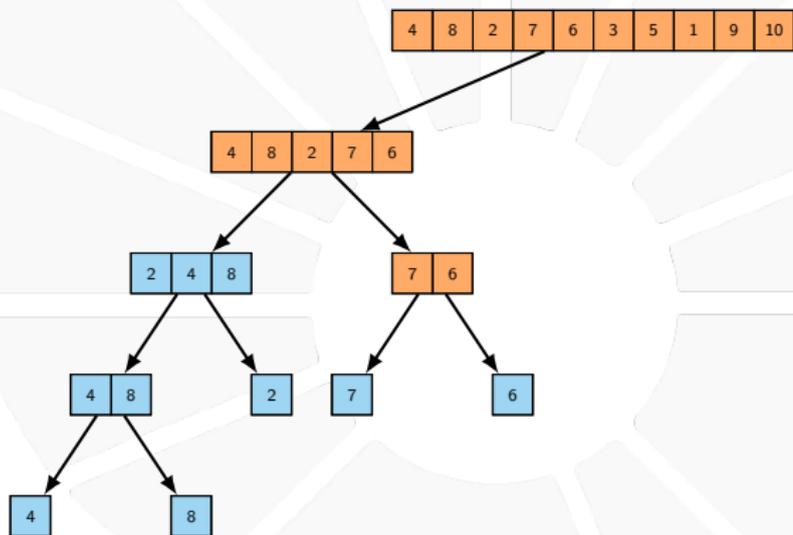


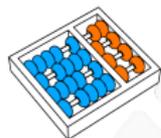
## Simulação



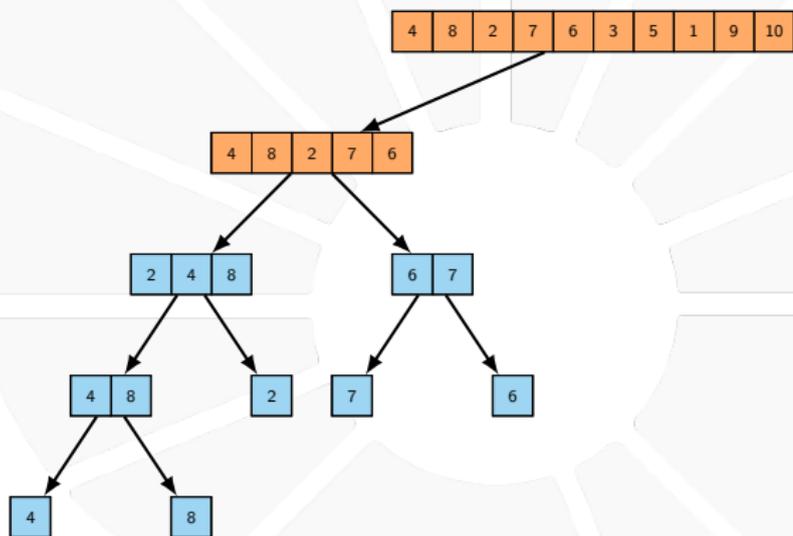


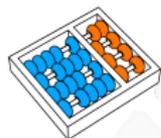
## Simulação



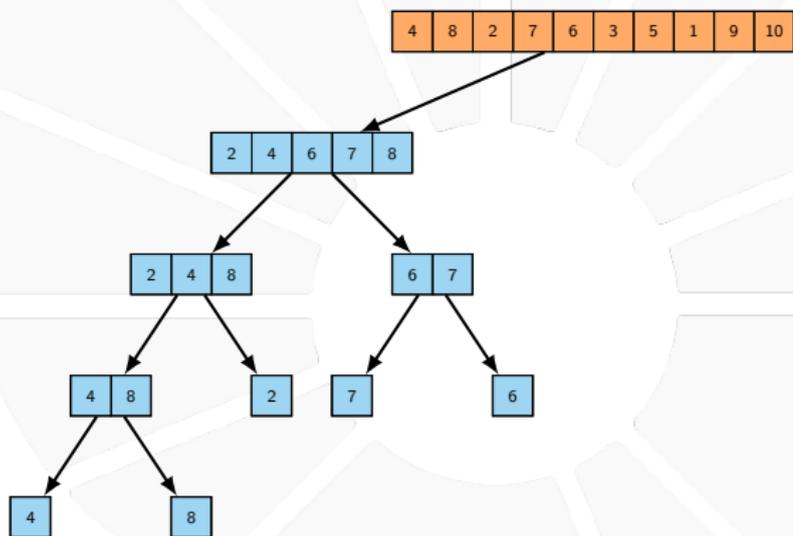


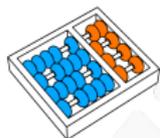
## Simulação



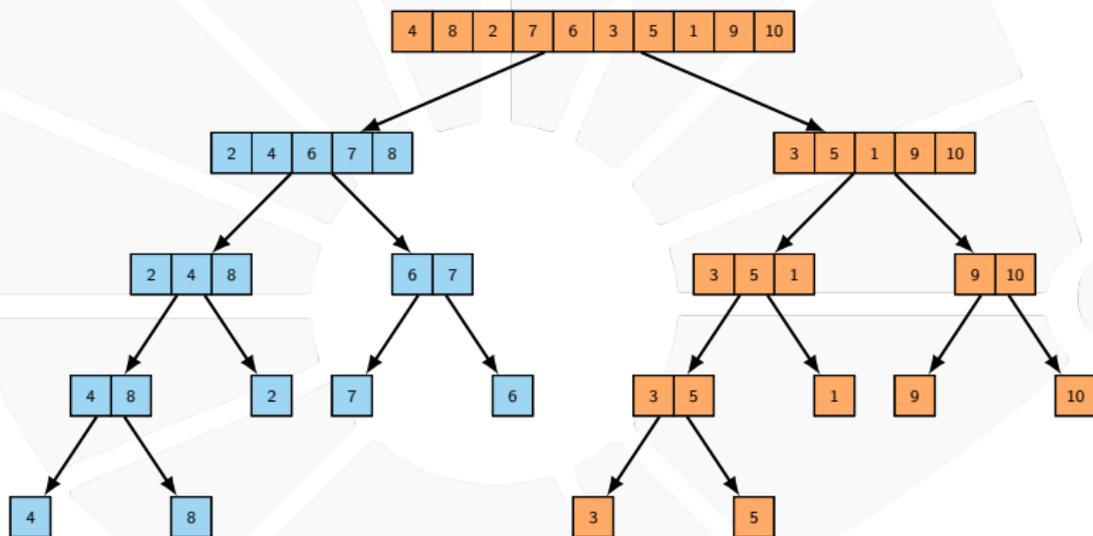


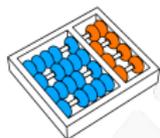
## Simulação



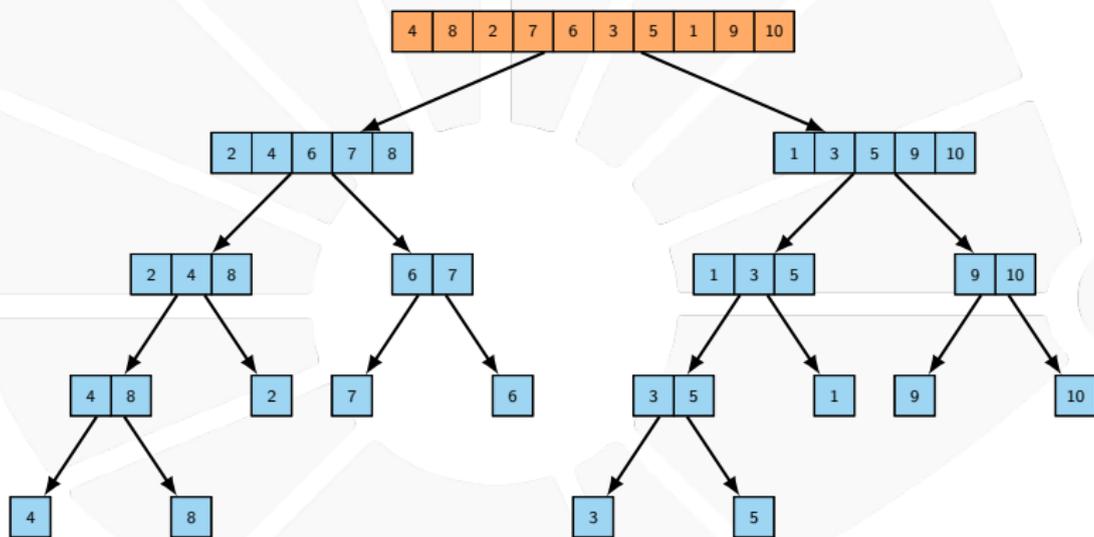


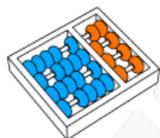
## Simulação



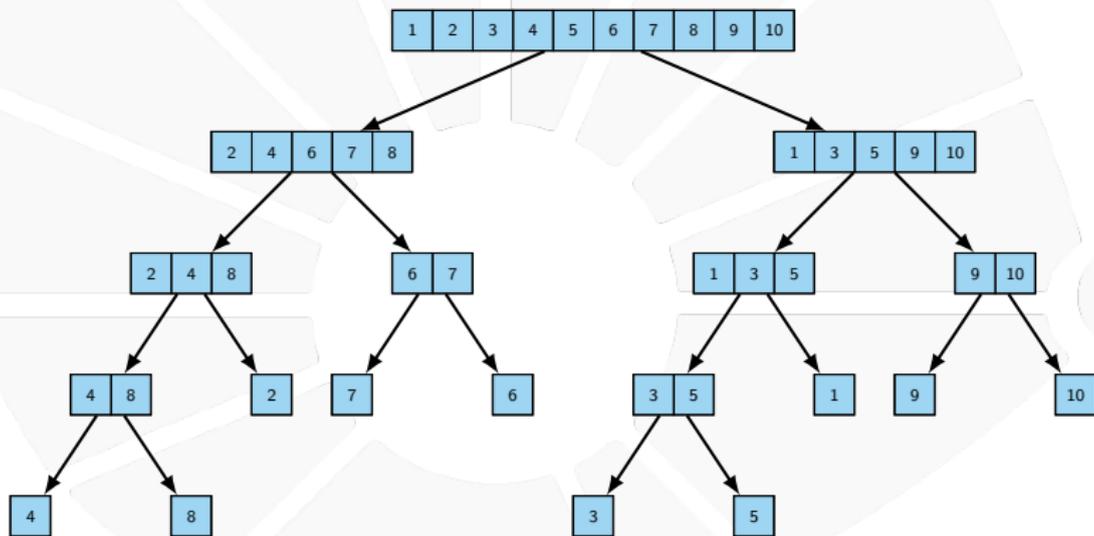


## Simulação



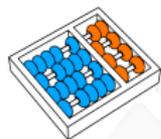


## Simulação

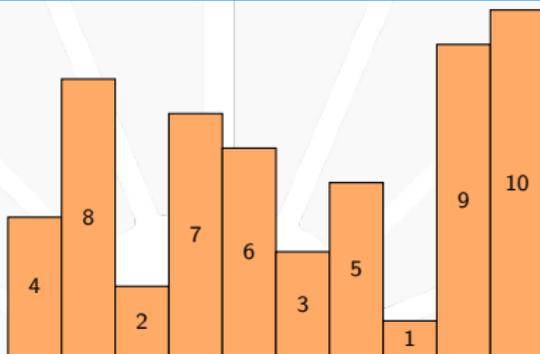


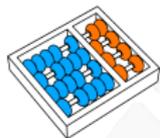


# QUICKSORT

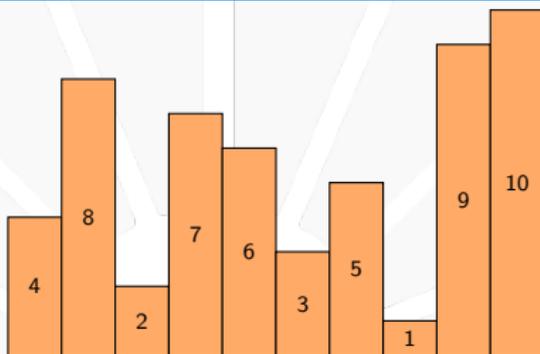


## Quicksort — Ideia

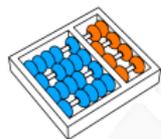




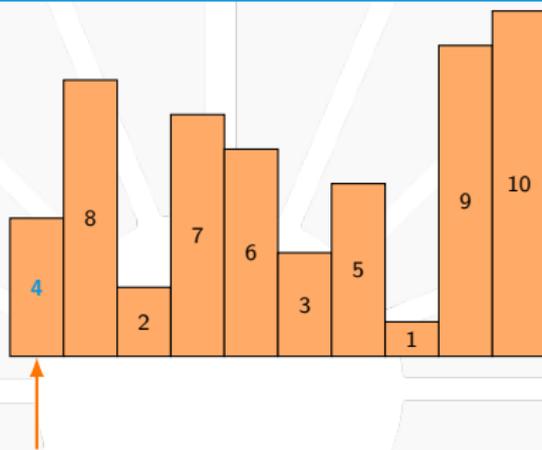
## Quicksort — Ideia



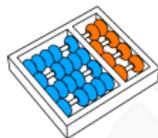
- ▶ Escolhemos um **pivô** (ex: 4).



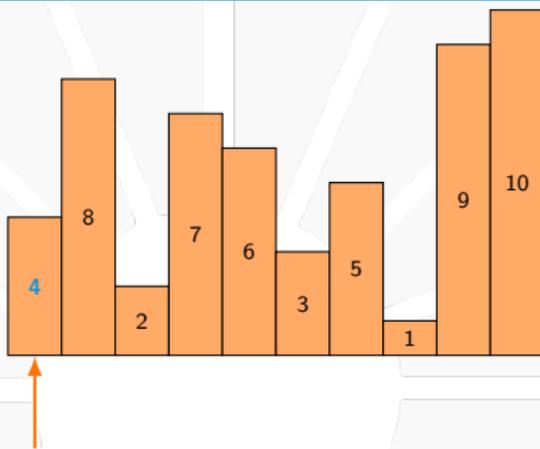
## Quicksort — Ideia



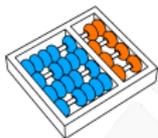
- ▶ Escolhemos um **pivô** (ex: 4).



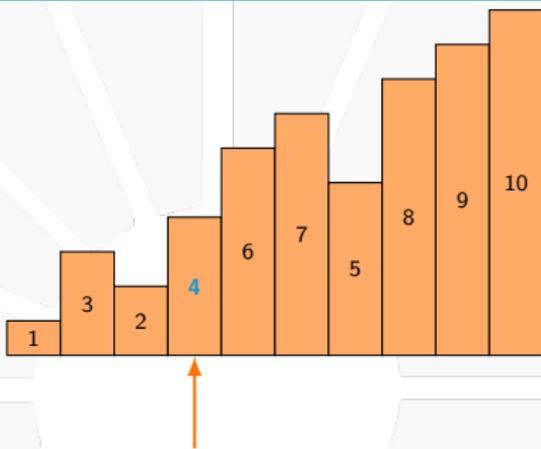
## Quicksort — Ideia



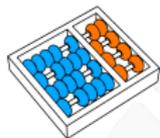
- ▶ Escolhemos um **pivô** (ex: 4).
- ▶ Colocamos:
  - ▶ Os elementos **menores** que o pivô **na esquerda**.
  - ▶ Os elementos **maiores** que o pivô **na direita**.



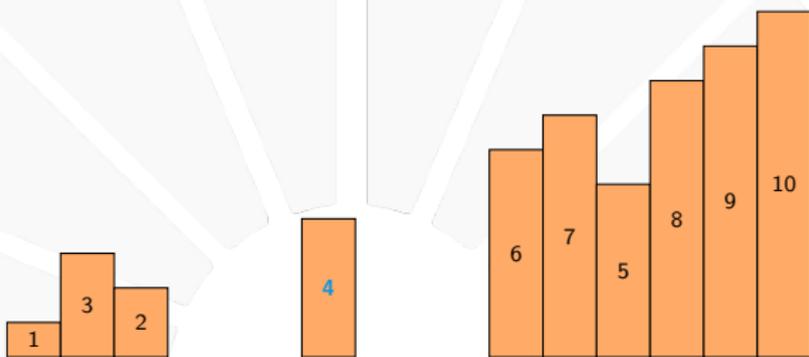
## Quicksort — Ideia



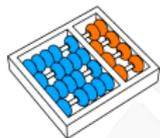
- ▶ Escolhemos um **pivô** (ex: 4).
- ▶ Colocamos:
  - ▶ Os elementos **menores** que o pivô **na esquerda**.
  - ▶ Os elementos **maiores** que o pivô **na direita**.



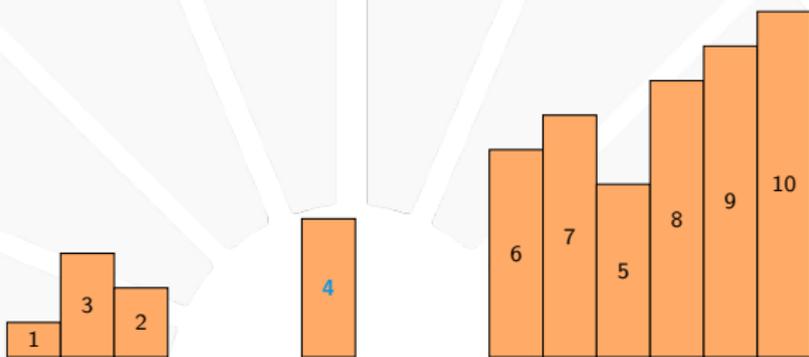
## Quicksort — Ideia



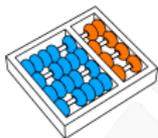
- ▶ Escolhemos um **pivô** (ex: 4).
- ▶ Colocamos:
  - ▶ Os elementos **menores** que o pivô **na esquerda**.
  - ▶ Os elementos **maiores** que o pivô **na direita**.
- ▶ O **pivô** está na posição **correta**.



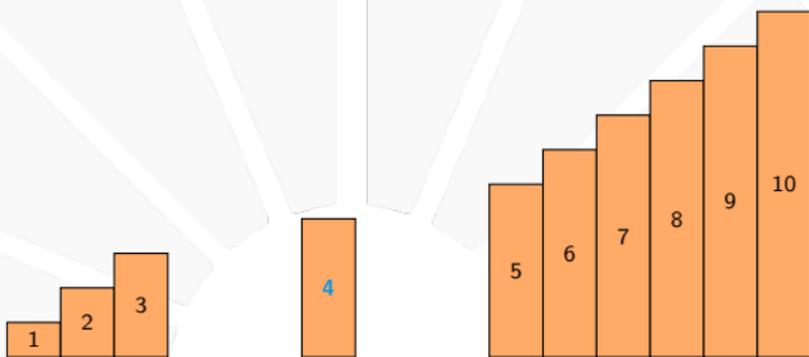
## Quicksort — Ideia



- ▶ Escolhemos um **pivô** (ex: 4).
- ▶ Colocamos:
  - ▶ Os elementos **menores** que o pivô **na esquerda**.
  - ▶ Os elementos **maiores** que o pivô **na direita**.
- ▶ O **pivô** está na posição **correta**.
- ▶ O lado esquerdo e o direito podem ser **ordenados independentemente**.



## Quicksort — Ideia



- ▶ Escolhemos um **pivô** (ex: 4).
- ▶ Colocamos:
  - ▶ Os elementos **menores** que o pivô **na esquerda**.
  - ▶ Os elementos **maiores** que o pivô **na direita**.
- ▶ O **pivô** está na posição **correta**.
- ▶ O lado esquerdo e o direito podem ser **ordenados independentemente**.



## Quicksort

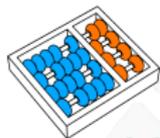
`partition(l, e, d):`



## Quicksort

`partition(l, e, d):`

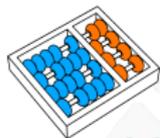
- ▶ Escolhe um **pivô**.



## Quicksort

**partition(l, e, d):**

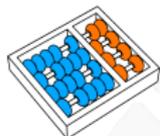
- ▶ Escolhe um **pivô**.
- ▶ Coloca os elementos **menores à esquerda** do pivô.



## Quicksort

**partition(l, e, d):**

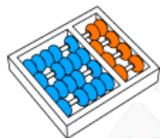
- ▶ Escolhe um **pivô**.
- ▶ Coloca os elementos **menores à esquerda** do pivô.
- ▶ Coloca os elementos **maiores à direita** do pivô.



## Quicksort

`partition(l, e, d):`

- ▶ Escolhe um **pivô**.
- ▶ Coloca os elementos **menores à esquerda** do pivô.
- ▶ Coloca os elementos **maiores à direita** do pivô.
- ▶ Devolve a **posição final do pivô**.

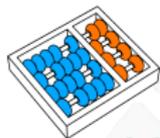


## Quicksort

`partition(l, e, d):`

- ▶ Escolhe um **pivô**.
- ▶ Coloca os elementos **menores à esquerda** do pivô.
- ▶ Coloca os elementos **maiores à direita** do pivô.
- ▶ Devolve a **posição final do pivô**.

```
1 def quicksort(l, e, d):  
2     if e < d:  
3         k = partition(l, e, d)  
4         quicksort(l, e, k - 1)  
5         quicksort(l, k + 1, d)
```



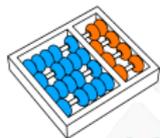
## Quicksort

`partition(l, e, d):`

- ▶ Escolhe um **pivô**.
- ▶ Coloca os elementos **menores à esquerda** do pivô.
- ▶ Coloca os elementos **maiores à direita** do pivô.
- ▶ Devolve a **posição final do pivô**.

```
1 def quicksort(l, e, d):  
2     if e < d:  
3         k = partition(l, e, d)  
4         quicksort(l, e, k - 1)  
5         quicksort(l, k + 1, d)
```

- ▶ Basta particionar a lista em duas.



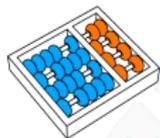
## Quicksort

`partition(l, e, d):`

- ▶ Escolhe um **pivô**.
- ▶ Coloca os elementos **menores à esquerda** do pivô.
- ▶ Coloca os elementos **maiores à direita** do pivô.
- ▶ Devolve a **posição final do pivô**.

```
1 def quicksort(l, e, d):  
2     if e < d:  
3         k = partition(l, e, d)  
4         quicksort(l, e, k - 1)  
5         quicksort(l, k + 1, d)
```

- ▶ Basta particionar a lista em duas.
- ▶ Depois, ordenar o lado esquerdo e o direito.



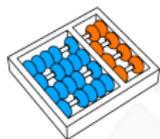
## Quicksort

`partition(l, e, d):`

- ▶ Escolhe um **pivô**.
- ▶ Coloca os elementos **menores à esquerda** do pivô.
- ▶ Coloca os elementos **maiores à direita** do pivô.
- ▶ Devolve a **posição final do pivô**.

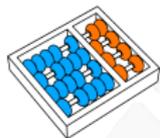
```
1 def quicksort(l, e, d):  
2     if e < d:  
3         k = partition(l, e, d)  
4         quicksort(l, e, k - 1)  
5         quicksort(l, k + 1, d)
```

- ▶ Basta particionar a lista em duas.
- ▶ Depois, ordenar o lado esquerdo e o direito.



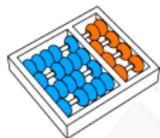
## Como particionar uma lista?

- ▶ Andamos da direita para a esquerda com um índice  $i$ .



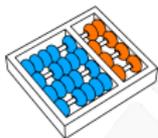
## Como particionar uma lista?

- ▶ Andamos da direita para a esquerda com um índice  $i$ .
- ▶ De  $i$  até  $\text{pos} - 1$  ficam os menores do que o pivô.



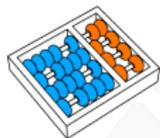
## Como particionar uma lista?

- ▶ Andamos da direita para a esquerda com um índice **i**.
- ▶ De **i** até **pos - 1** ficam os menores do que o pivô.
- ▶ De **pos** até **d** ficam os maiores ou iguais ao pivô.



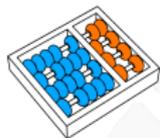
## Como particionar uma lista?

- ▶ Andamos da direita para a esquerda com um índice  $i$ .
- ▶ De  $i$  até  $pos - 1$  ficam os menores do que o pivô.
- ▶ De  $pos$  até  $d$  ficam os maiores ou iguais ao pivô.
- ▶ Se o elemento em  $i$  for maior ou igual ao pivô:



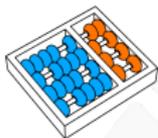
## Como particionar uma lista?

- ▶ Andamos da direita para a esquerda com um índice  $i$ .
- ▶ De  $i$  até  $pos - 1$  ficam os menores do que o pivô.
- ▶ De  $pos$  até  $d$  ficam os maiores ou iguais ao pivô.
- ▶ Se o elemento em  $i$  for maior ou igual ao pivô:
  - ▶ Diminuímos  $pos$  e realizamos uma troca de  $i$  com  $pos$ .



## Como particionar uma lista?

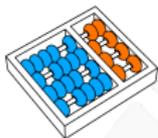
- ▶ Andamos da direita para a esquerda com um índice **i**.
- ▶ De **i** até **pos - 1** ficam os menores do que o pivô.
- ▶ De **pos** até **d** ficam os maiores ou iguais ao pivô.
- ▶ Se o elemento em **i** for maior ou igual ao pivô:
  - ▶ Diminuímos **pos** e realizamos uma troca de **i** com **pos**.
- ▶ No final, o pivô está em **pos**.



## Como particionar uma lista?

- ▶ Andamos da direita para a esquerda com um índice **i**.
- ▶ De **i** até **pos - 1** ficam os menores do que o pivô.
- ▶ De **pos** até **d** ficam os maiores ou iguais ao pivô.
- ▶ Se o elemento em **i** for maior ou igual ao pivô:
  - ▶ Diminuímos **pos** e realizamos uma troca de **i** com **pos**.
- ▶ No final, o pivô está em **pos**.

```
1 def partition(l, e, d):
2     pivo = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivo:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
```

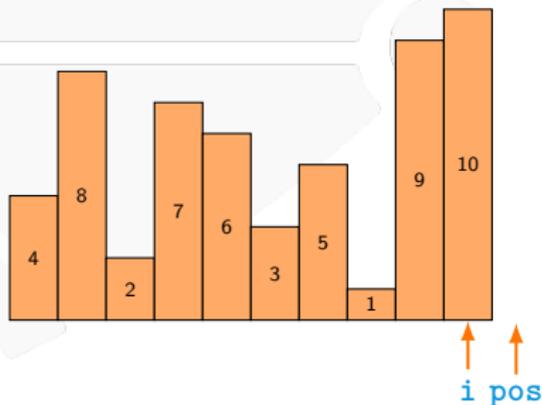


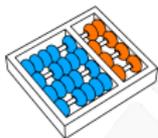
## Como particionar uma lista?

- ▶ Andamos da direita para a esquerda com um índice  $i$ .
- ▶ De  $i$  até  $pos - 1$  ficam os menores do que o pivô.
- ▶ De  $pos$  até  $d$  ficam os maiores ou iguais ao pivô.
- ▶ Se o elemento em  $i$  for maior ou igual ao pivô:
  - ▶ Diminuímos  $pos$  e realizamos uma troca de  $i$  com  $pos$ .
- ▶ No final, o pivô está em  $pos$ .

```

1 def partition(l, e, d):
2     pivô = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivô:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
  
```



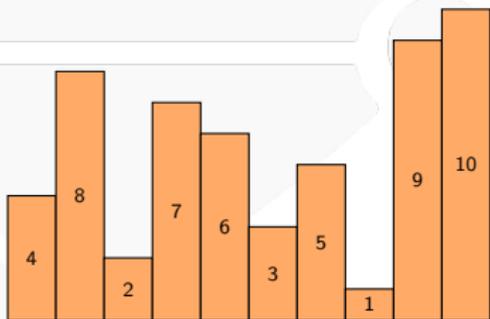


## Como particionar uma lista?

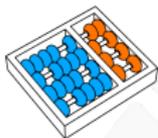
- ▶ Andamos da direita para a esquerda com um índice  $i$ .
- ▶ De  $i$  até  $\text{pos} - 1$  ficam os menores do que o pivô.
- ▶ De  $\text{pos}$  até  $d$  ficam os maiores ou iguais ao pivô.
- ▶ Se o elemento em  $i$  for maior ou igual ao pivô:
  - ▶ Diminuímos  $\text{pos}$  e realizamos uma troca de  $i$  com  $\text{pos}$ .
- ▶ No final, o pivô está em  $\text{pos}$ .

```

1 def partition(l, e, d):
2     pivô = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivô:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
  
```



$i/\text{pos}$

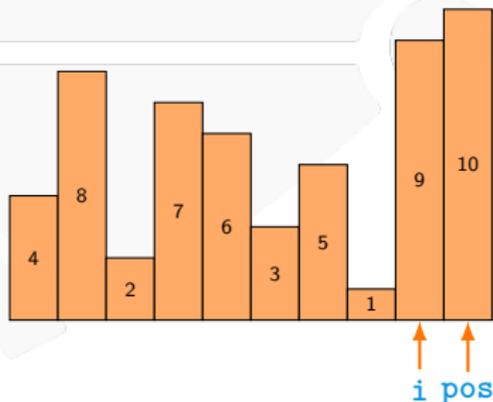


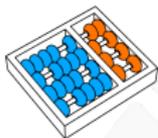
## Como particionar uma lista?

- ▶ Andamos da direita para a esquerda com um índice  $i$ .
- ▶ De  $i$  até  $pos - 1$  ficam os menores do que o pivô.
- ▶ De  $pos$  até  $d$  ficam os maiores ou iguais ao pivô.
- ▶ Se o elemento em  $i$  for maior ou igual ao pivô:
  - ▶ Diminuímos  $pos$  e realizamos uma troca de  $i$  com  $pos$ .
- ▶ No final, o pivô está em  $pos$ .

```

1 def partition(l, e, d):
2     pivô = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivô:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
  
```



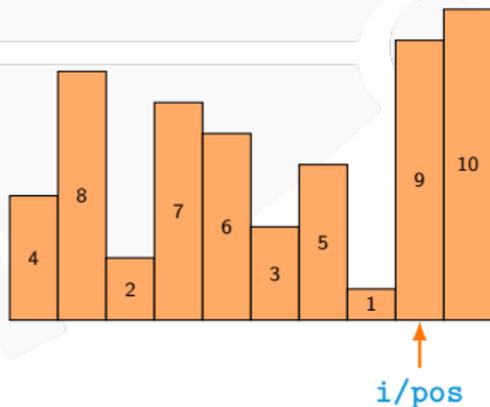


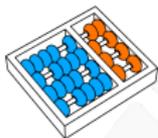
## Como particionar uma lista?

- ▶ Andamos da direita para a esquerda com um índice  $i$ .
- ▶ De  $i$  até  $\text{pos} - 1$  ficam os menores do que o pivô.
- ▶ De  $\text{pos}$  até  $d$  ficam os maiores ou iguais ao pivô.
- ▶ Se o elemento em  $i$  for maior ou igual ao pivô:
  - ▶ Diminuímos  $\text{pos}$  e realizamos uma troca de  $i$  com  $\text{pos}$ .
- ▶ No final, o pivô está em  $\text{pos}$ .

```

1 def partition(l, e, d):
2     pivô = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivô:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
  
```



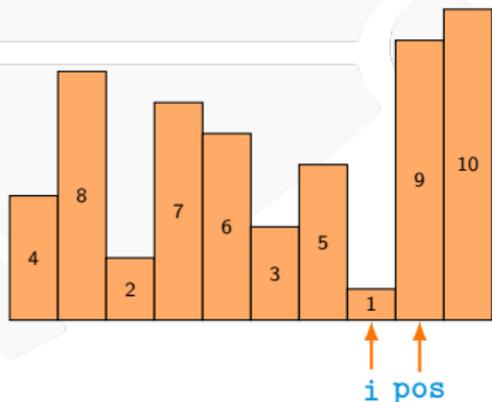


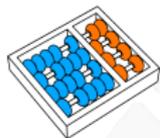
## Como particionar uma lista?

- ▶ Andamos da direita para a esquerda com um índice  $i$ .
- ▶ De  $i$  até  $pos - 1$  ficam os menores do que o pivô.
- ▶ De  $pos$  até  $d$  ficam os maiores ou iguais ao pivô.
- ▶ Se o elemento em  $i$  for maior ou igual ao pivô:
  - ▶ Diminuímos  $pos$  e realizamos uma troca de  $i$  com  $pos$ .
- ▶ No final, o pivô está em  $pos$ .

```

1 def partition(l, e, d):
2     pivô = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivô:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
  
```



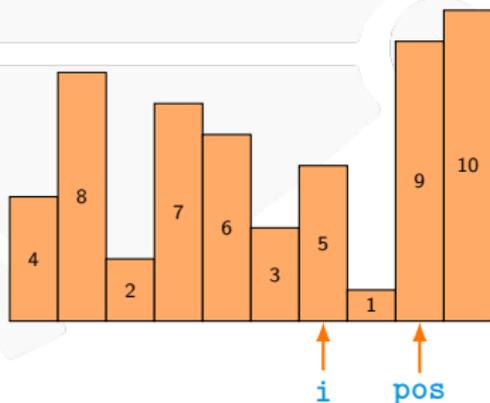


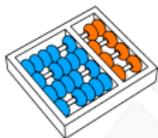
## Como particionar uma lista?

- ▶ Andamos da direita para a esquerda com um índice **i**.
- ▶ De **i** até **pos - 1** ficam os menores do que o pivô.
- ▶ De **pos** até **d** ficam os maiores ou iguais ao pivô.
- ▶ Se o elemento em **i** for maior ou igual ao pivô:
  - ▶ Diminuímos **pos** e realizamos uma troca de **i** com **pos**.
- ▶ No final, o pivô está em **pos**.

```

1 def partition(l, e, d):
2     pivô = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivô:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
  
```



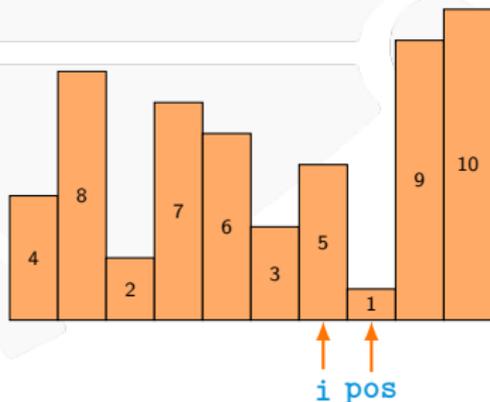


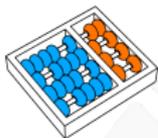
## Como particionar uma lista?

- ▶ Andamos da direita para a esquerda com um índice  $i$ .
- ▶ De  $i$  até  $pos - 1$  ficam os menores do que o pivô.
- ▶ De  $pos$  até  $d$  ficam os maiores ou iguais ao pivô.
- ▶ Se o elemento em  $i$  for maior ou igual ao pivô:
  - ▶ Diminuímos  $pos$  e realizamos uma troca de  $i$  com  $pos$ .
- ▶ No final, o pivô está em  $pos$ .

```

1 def partition(l, e, d):
2     pivô = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivô:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
  
```



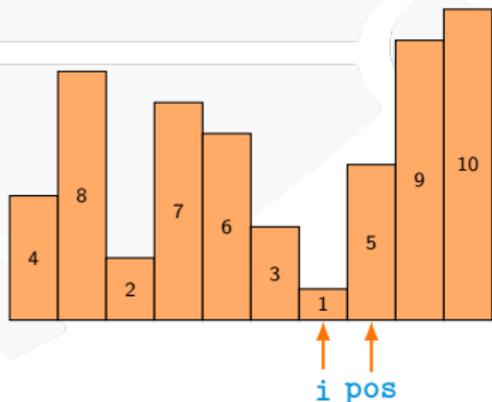


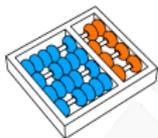
## Como particionar uma lista?

- ▶ Andamos da direita para a esquerda com um índice  $i$ .
- ▶ De  $i$  até  $pos - 1$  ficam os menores do que o pivô.
- ▶ De  $pos$  até  $d$  ficam os maiores ou iguais ao pivô.
- ▶ Se o elemento em  $i$  for maior ou igual ao pivô:
  - ▶ Diminuímos  $pos$  e realizamos uma troca de  $i$  com  $pos$ .
- ▶ No final, o pivô está em  $pos$ .

```

1 def partition(l, e, d):
2     pivô = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivô:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
  
```



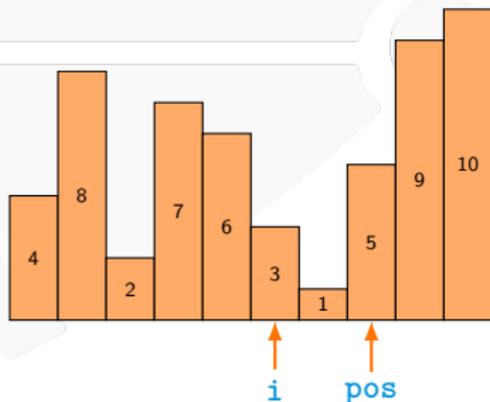


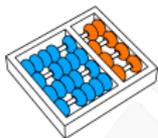
## Como particionar uma lista?

- ▶ Andamos da direita para a esquerda com um índice  $i$ .
- ▶ De  $i$  até  $pos - 1$  ficam os menores do que o pivô.
- ▶ De  $pos$  até  $d$  ficam os maiores ou iguais ao pivô.
- ▶ Se o elemento em  $i$  for maior ou igual ao pivô:
  - ▶ Diminuímos  $pos$  e realizamos uma troca de  $i$  com  $pos$ .
- ▶ No final, o pivô está em  $pos$ .

```

1 def partition(l, e, d):
2     pivô = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivô:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
  
```



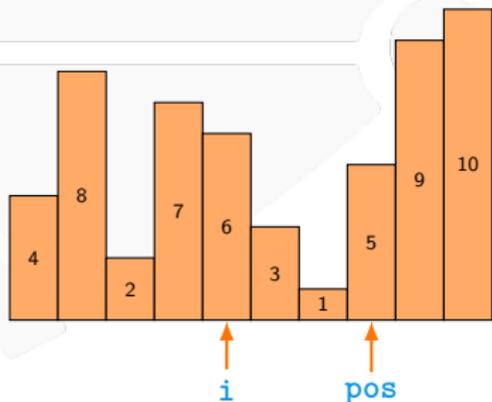


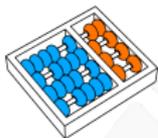
## Como particionar uma lista?

- ▶ Andamos da direita para a esquerda com um índice  $i$ .
- ▶ De  $i$  até  $pos - 1$  ficam os menores do que o pivô.
- ▶ De  $pos$  até  $d$  ficam os maiores ou iguais ao pivô.
- ▶ Se o elemento em  $i$  for maior ou igual ao pivô:
  - ▶ Diminuímos  $pos$  e realizamos uma troca de  $i$  com  $pos$ .
- ▶ No final, o pivô está em  $pos$ .

```

1 def partition(l, e, d):
2     pivô = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivô:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
  
```



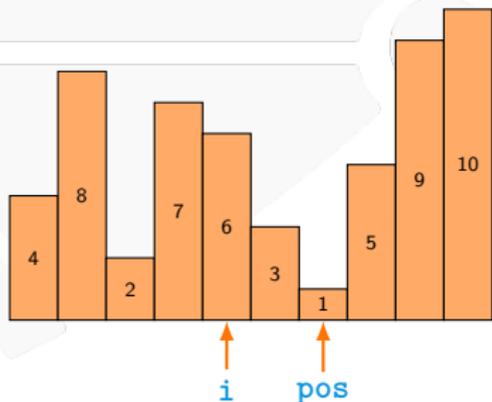


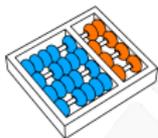
## Como particionar uma lista?

- ▶ Andamos da direita para a esquerda com um índice **i**.
- ▶ De **i** até **pos - 1** ficam os menores do que o pivô.
- ▶ De **pos** até **d** ficam os maiores ou iguais ao pivô.
- ▶ Se o elemento em **i** for maior ou igual ao pivô:
  - ▶ Diminuímos **pos** e realizamos uma troca de **i** com **pos**.
- ▶ No final, o pivô está em **pos**.

```

1 def partition(l, e, d):
2     pivô = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivô:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
  
```



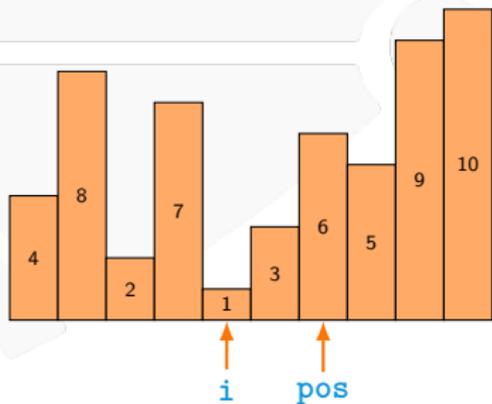


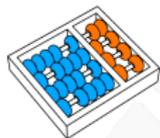
## Como particionar uma lista?

- ▶ Andamos da direita para a esquerda com um índice  $i$ .
- ▶ De  $i$  até  $pos - 1$  ficam os menores do que o pivô.
- ▶ De  $pos$  até  $d$  ficam os maiores ou iguais ao pivô.
- ▶ Se o elemento em  $i$  for maior ou igual ao pivô:
  - ▶ Diminuímos  $pos$  e realizamos uma troca de  $i$  com  $pos$ .
- ▶ No final, o pivô está em  $pos$ .

```

1 def partition(l, e, d):
2     pivô = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivô:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
  
```



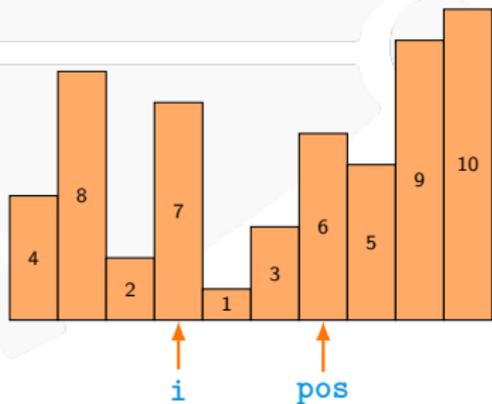


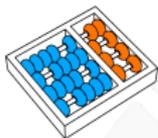
## Como particionar uma lista?

- ▶ Andamos da direita para a esquerda com um índice  $i$ .
- ▶ De  $i$  até  $pos - 1$  ficam os menores do que o pivô.
- ▶ De  $pos$  até  $d$  ficam os maiores ou iguais ao pivô.
- ▶ Se o elemento em  $i$  for maior ou igual ao pivô:
  - ▶ Diminuímos  $pos$  e realizamos uma troca de  $i$  com  $pos$ .
- ▶ No final, o pivô está em  $pos$ .

```

1 def partition(l, e, d):
2     pivô = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivô:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
  
```



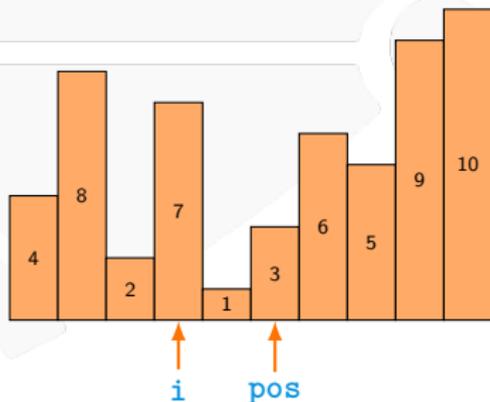


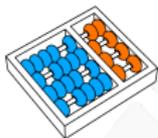
## Como particionar uma lista?

- ▶ Andamos da direita para a esquerda com um índice  $i$ .
- ▶ De  $i$  até  $\text{pos} - 1$  ficam os menores do que o pivô.
- ▶ De  $\text{pos}$  até  $d$  ficam os maiores ou iguais ao pivô.
- ▶ Se o elemento em  $i$  for maior ou igual ao pivô:
  - ▶ Diminuímos  $\text{pos}$  e realizamos uma troca de  $i$  com  $\text{pos}$ .
- ▶ No final, o pivô está em  $\text{pos}$ .

```

1 def partition(l, e, d):
2     pivô = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivô:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
  
```



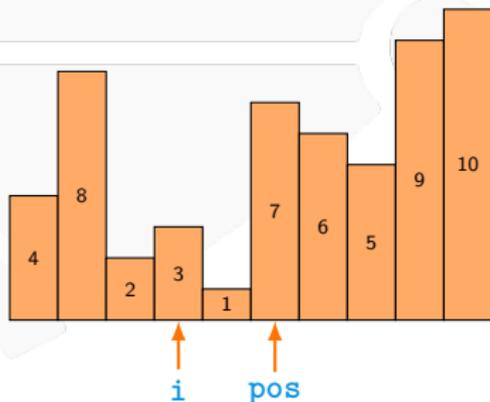


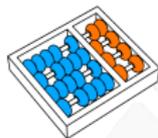
## Como particionar uma lista?

- ▶ Andamos da direita para a esquerda com um índice  $i$ .
- ▶ De  $i$  até  $pos - 1$  ficam os menores do que o pivô.
- ▶ De  $pos$  até  $d$  ficam os maiores ou iguais ao pivô.
- ▶ Se o elemento em  $i$  for maior ou igual ao pivô:
  - ▶ Diminuímos  $pos$  e realizamos uma troca de  $i$  com  $pos$ .
- ▶ No final, o pivô está em  $pos$ .

```

1 def partition(l, e, d):
2     pivô = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivô:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
  
```



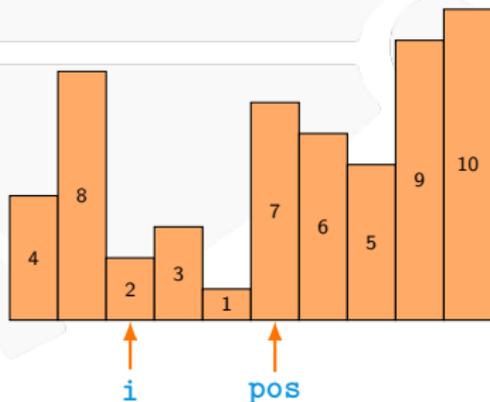


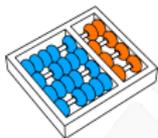
## Como particionar uma lista?

- ▶ Andamos da direita para a esquerda com um índice  $i$ .
- ▶ De  $i$  até  $pos - 1$  ficam os menores do que o pivô.
- ▶ De  $pos$  até  $d$  ficam os maiores ou iguais ao pivô.
- ▶ Se o elemento em  $i$  for maior ou igual ao pivô:
  - ▶ Diminuímos  $pos$  e realizamos uma troca de  $i$  com  $pos$ .
- ▶ No final, o pivô está em  $pos$ .

```

1 def partition(l, e, d):
2     pivô = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivô:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
  
```



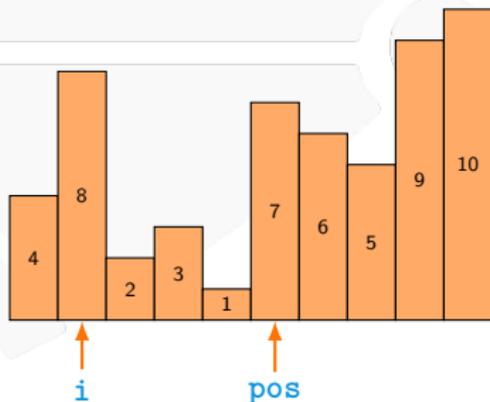


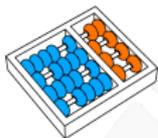
## Como particionar uma lista?

- ▶ Andamos da direita para a esquerda com um índice  $i$ .
- ▶ De  $i$  até  $pos - 1$  ficam os menores do que o pivô.
- ▶ De  $pos$  até  $d$  ficam os maiores ou iguais ao pivô.
- ▶ Se o elemento em  $i$  for maior ou igual ao pivô:
  - ▶ Diminuímos  $pos$  e realizamos uma troca de  $i$  com  $pos$ .
- ▶ No final, o pivô está em  $pos$ .

```

1 def partition(l, e, d):
2     pivô = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivô:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
  
```



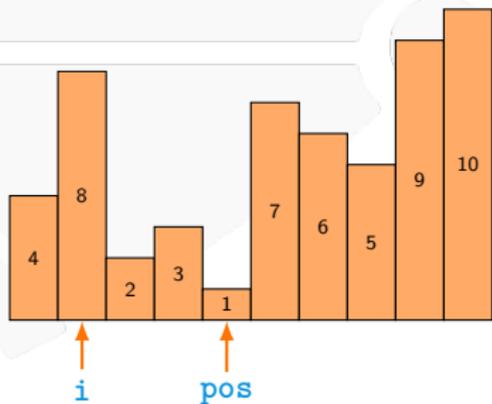


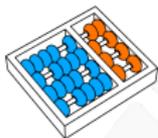
## Como particionar uma lista?

- ▶ Andamos da direita para a esquerda com um índice  $i$ .
- ▶ De  $i$  até  $pos - 1$  ficam os menores do que o pivô.
- ▶ De  $pos$  até  $d$  ficam os maiores ou iguais ao pivô.
- ▶ Se o elemento em  $i$  for maior ou igual ao pivô:
  - ▶ Diminuímos  $pos$  e realizamos uma troca de  $i$  com  $pos$ .
- ▶ No final, o pivô está em  $pos$ .

```

1 def partition(l, e, d):
2     pivô = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivô:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
  
```



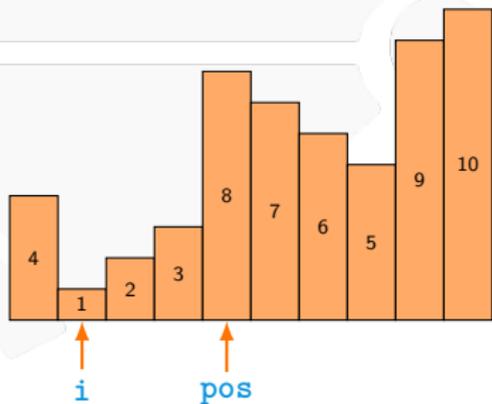


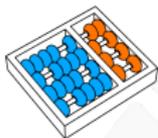
## Como particionar uma lista?

- ▶ Andamos da direita para a esquerda com um índice  $i$ .
- ▶ De  $i$  até  $pos - 1$  ficam os menores do que o pivô.
- ▶ De  $pos$  até  $d$  ficam os maiores ou iguais ao pivô.
- ▶ Se o elemento em  $i$  for maior ou igual ao pivô:
  - ▶ Diminuímos  $pos$  e realizamos uma troca de  $i$  com  $pos$ .
- ▶ No final, o pivô está em  $pos$ .

```

1 def partition(l, e, d):
2     pivô = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivô:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
  
```



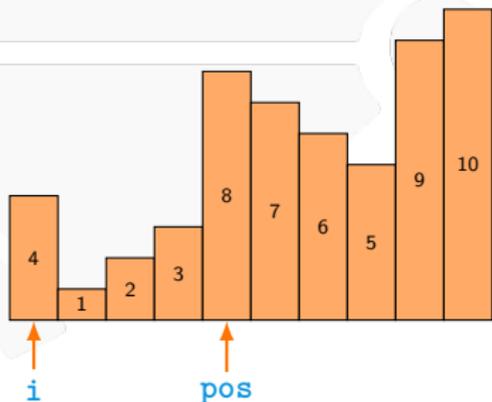


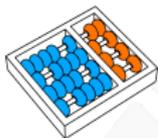
## Como particionar uma lista?

- ▶ Andamos da direita para a esquerda com um índice  $i$ .
- ▶ De  $i$  até  $pos - 1$  ficam os menores do que o pivô.
- ▶ De  $pos$  até  $d$  ficam os maiores ou iguais ao pivô.
- ▶ Se o elemento em  $i$  for maior ou igual ao pivô:
  - ▶ Diminuímos  $pos$  e realizamos uma troca de  $i$  com  $pos$ .
- ▶ No final, o pivô está em  $pos$ .

```

1 def partition(l, e, d):
2     pivô = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivô:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
  
```



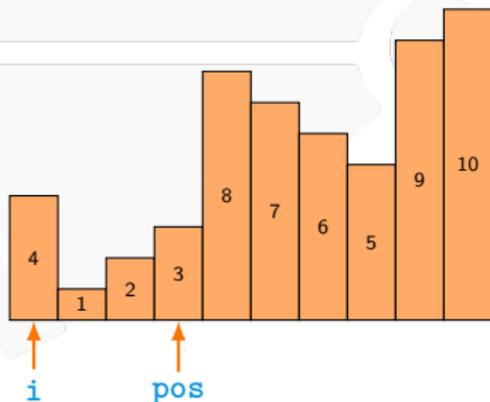


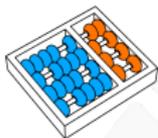
## Como particionar uma lista?

- ▶ Andamos da direita para a esquerda com um índice  $i$ .
- ▶ De  $i$  até  $pos - 1$  ficam os menores do que o pivô.
- ▶ De  $pos$  até  $d$  ficam os maiores ou iguais ao pivô.
- ▶ Se o elemento em  $i$  for maior ou igual ao pivô:
  - ▶ Diminuímos  $pos$  e realizamos uma troca de  $i$  com  $pos$ .
- ▶ No final, o pivô está em  $pos$ .

```

1 def partition(l, e, d):
2     pivô = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivô:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
  
```



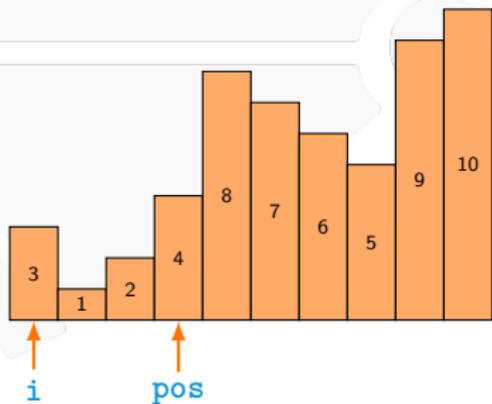


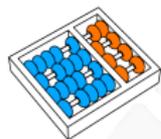
## Como particionar uma lista?

- ▶ Andamos da direita para a esquerda com um índice  $i$ .
- ▶ De  $i$  até  $pos - 1$  ficam os menores do que o pivô.
- ▶ De  $pos$  até  $d$  ficam os maiores ou iguais ao pivô.
- ▶ Se o elemento em  $i$  for maior ou igual ao pivô:
  - ▶ Diminuímos  $pos$  e realizamos uma troca de  $i$  com  $pos$ .
- ▶ No final, o pivô está em  $pos$ .

```

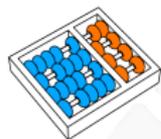
1 def partition(l, e, d):
2     pivô = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivô:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
  
```





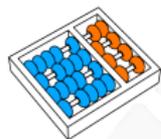
## Simulação do Quicksort

4	8	2	7	6	3	5	1	9	10
---	---	---	---	---	---	---	---	---	----



## Simulação do Quicksort

1	3	2	4	6	7	5	8	9	10
---	---	---	---	---	---	---	---	---	----

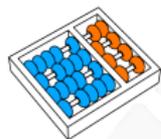


## Simulação do Quicksort

1	3	2	4	6	7	5	8	9	10
---	---	---	---	---	---	---	---	---	----



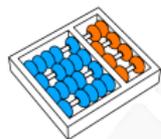
1	3	2
---	---	---



## Simulação do Quicksort

1	3	2	4	6	7	5	8	9	10
---	---	---	---	---	---	---	---	---	----

1	3	2
---	---	---

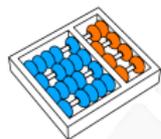


## Simulação do Quicksort

1	3	2	4	6	7	5	8	9	10
---	---	---	---	---	---	---	---	---	----

1	3	2
---	---	---

∅
---



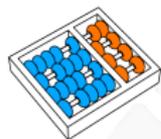
## Simulação do Quicksort

1	3	2	4	6	7	5	8	9	10
---	---	---	---	---	---	---	---	---	----

1	3	2
---	---	---

∅
---

3	2
---	---



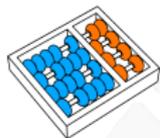
## Simulação do Quicksort

1	2	3	4	6	7	5	8	9	10
---	---	---	---	---	---	---	---	---	----

1	2	3
---	---	---

∅

2	3
---	---



## Simulação do Quicksort

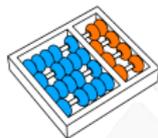
1	2	3	4	6	7	5	8	9	10
---	---	---	---	---	---	---	---	---	----

1	2	3
---	---	---

∅

2	3
---	---

2
---



## Simulação do Quicksort

1	2	3	4	6	7	5	8	9	10
---	---	---	---	---	---	---	---	---	----

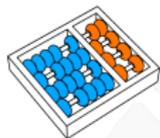
1	2	3
---	---	---

 $\emptyset$ 

2	3
---	---

2
---

 $\emptyset$



## Simulação do Quicksort

1	2	3	4	6	7	5	8	9	10
---	---	---	---	---	---	---	---	---	----

1	2	3
---	---	---

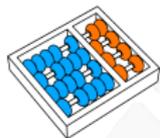
6	7	5	8	9	10
---	---	---	---	---	----

∅

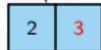
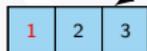
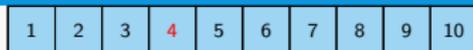
2	3
---	---

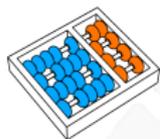
2
---

∅

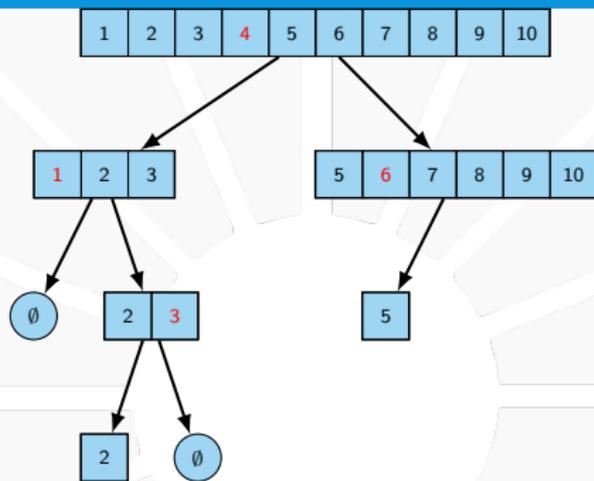


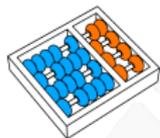
## Simulação do Quicksort



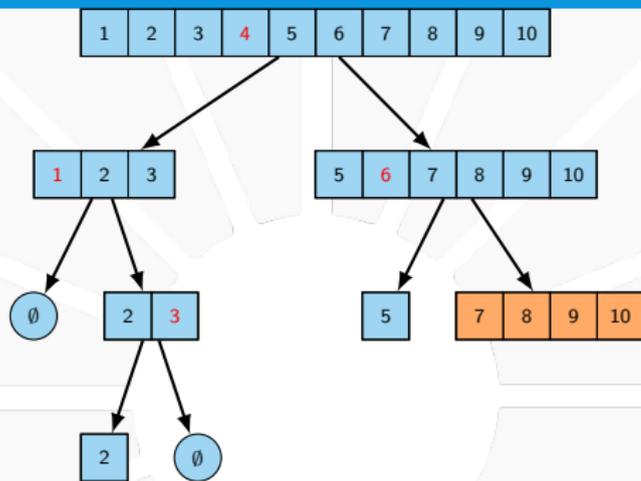


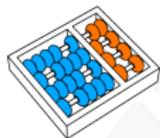
## Simulação do Quicksort



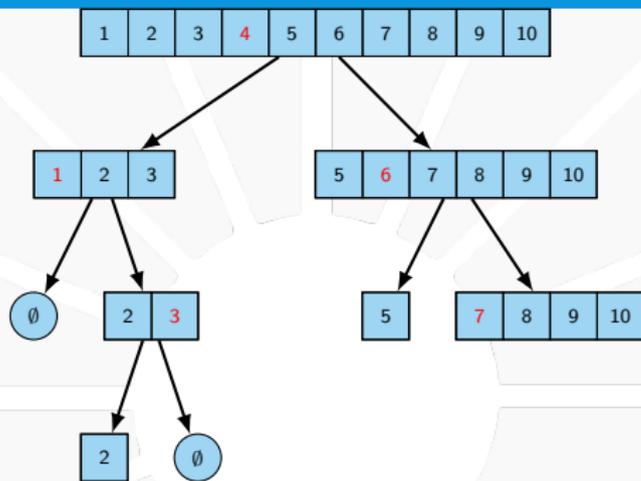


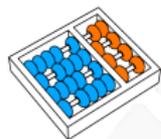
## Simulação do Quicksort



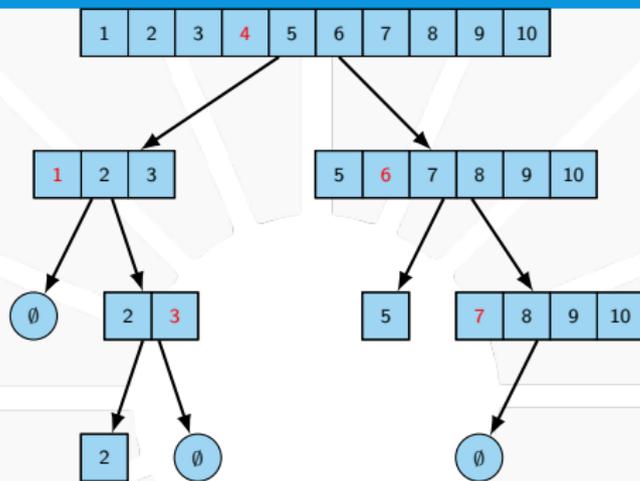


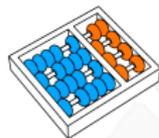
## Simulação do Quicksort



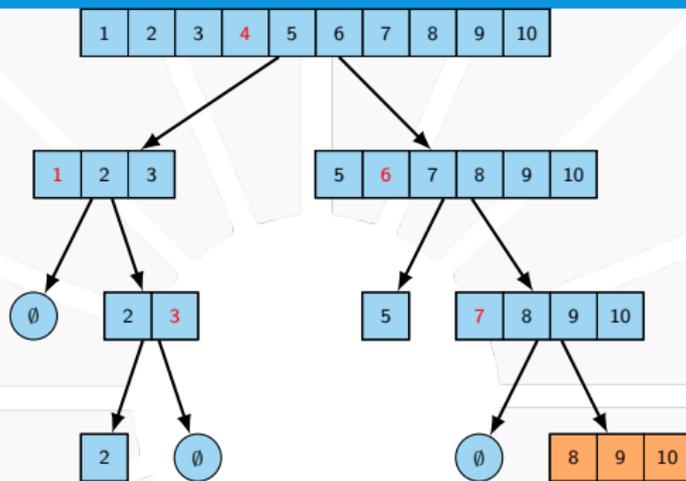


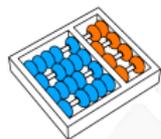
## Simulação do Quicksort



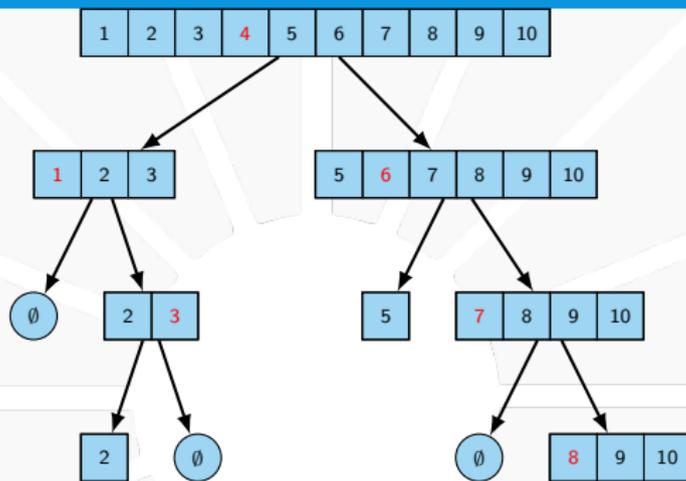


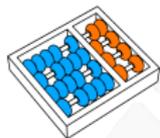
## Simulação do Quicksort



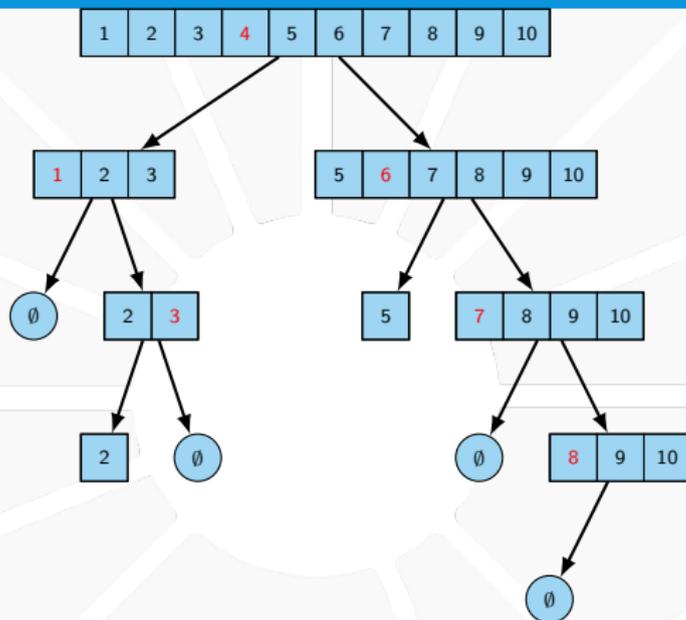


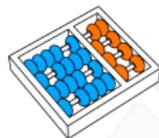
## Simulação do Quicksort



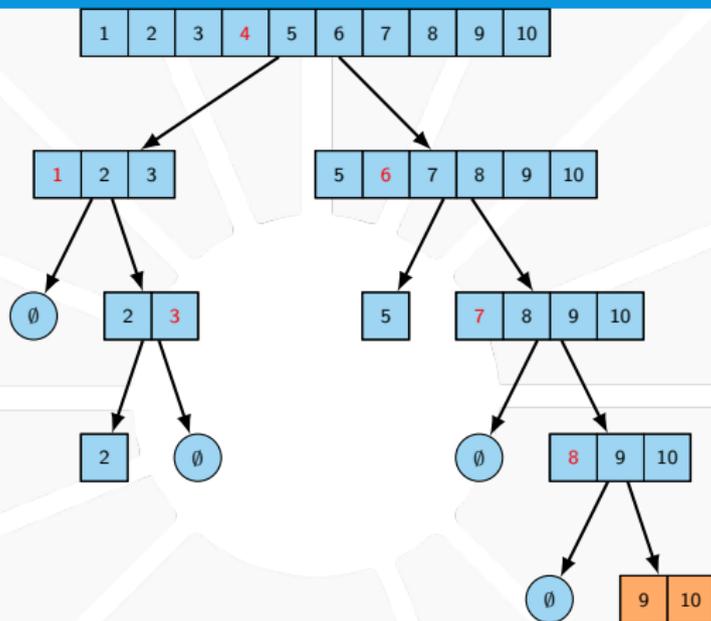


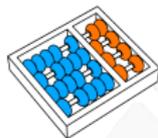
## Simulação do Quicksort



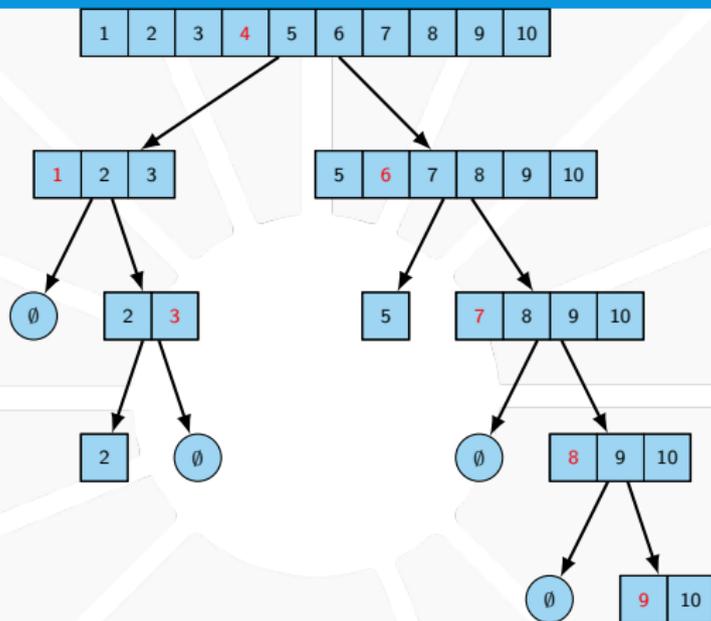


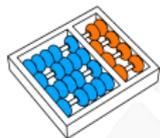
## Simulação do Quicksort



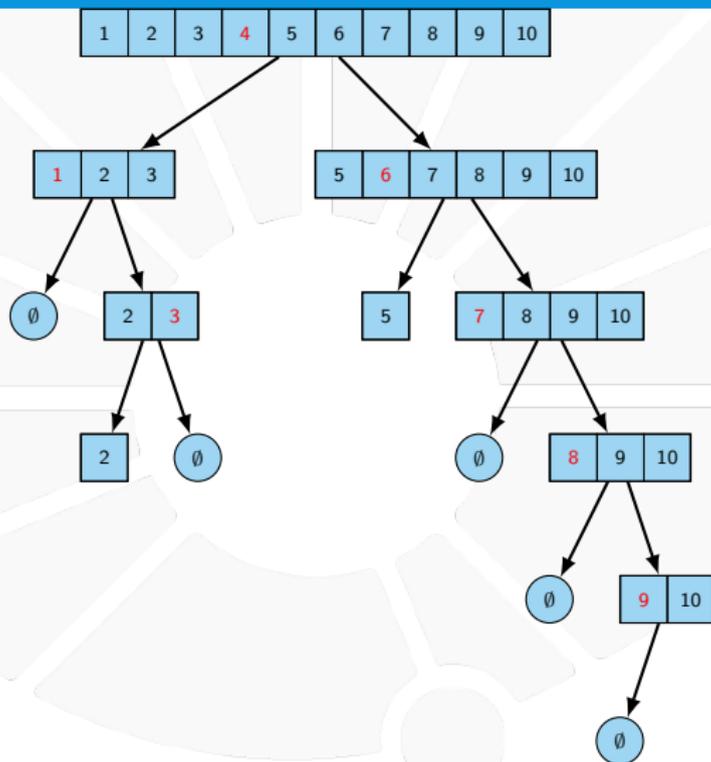


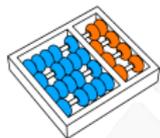
## Simulação do Quicksort



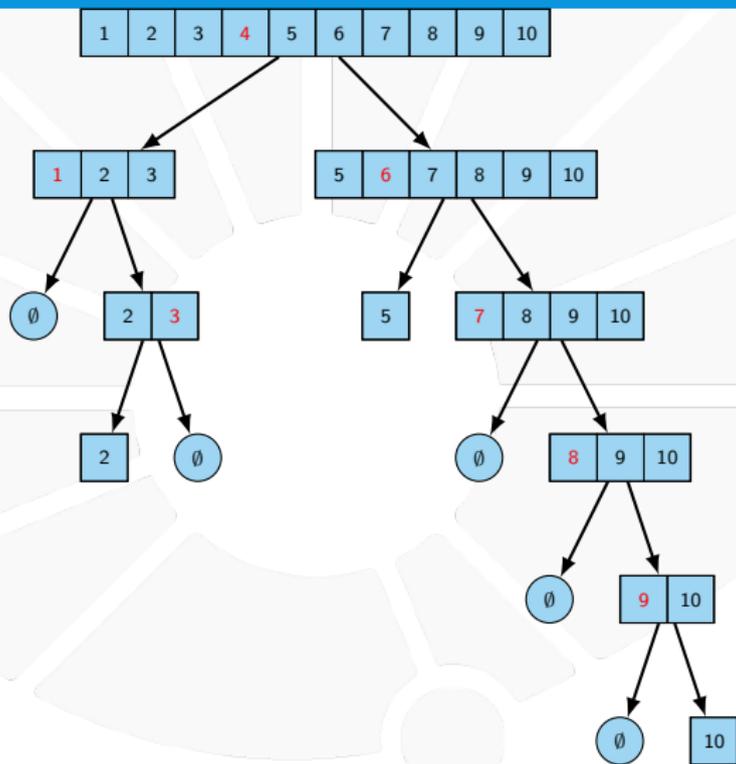


## Simulação do Quicksort



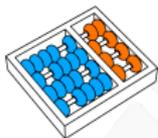


## Simulação do Quicksort



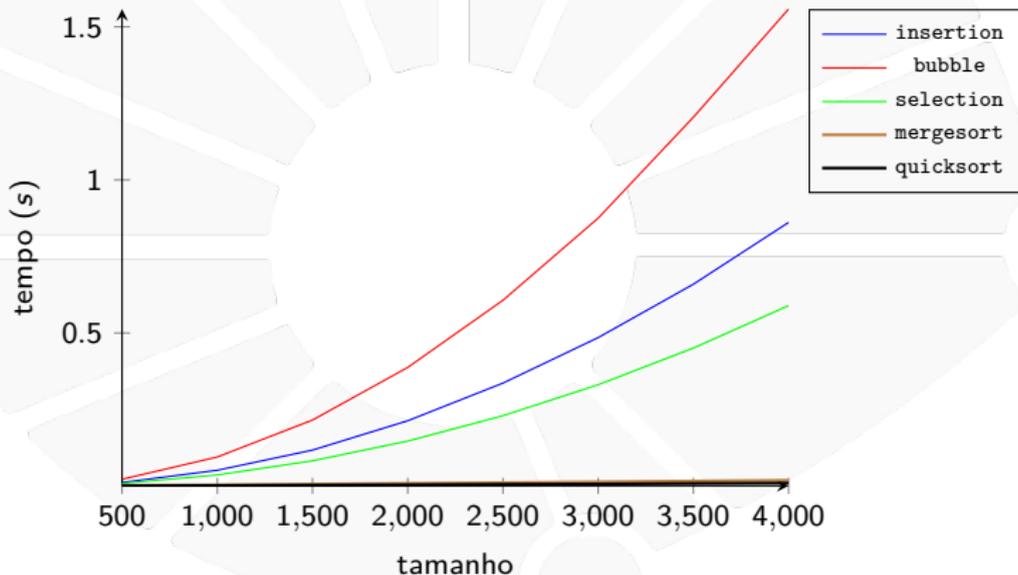


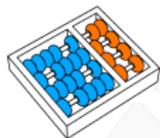
# EXPERIMENTOS



## Experimento

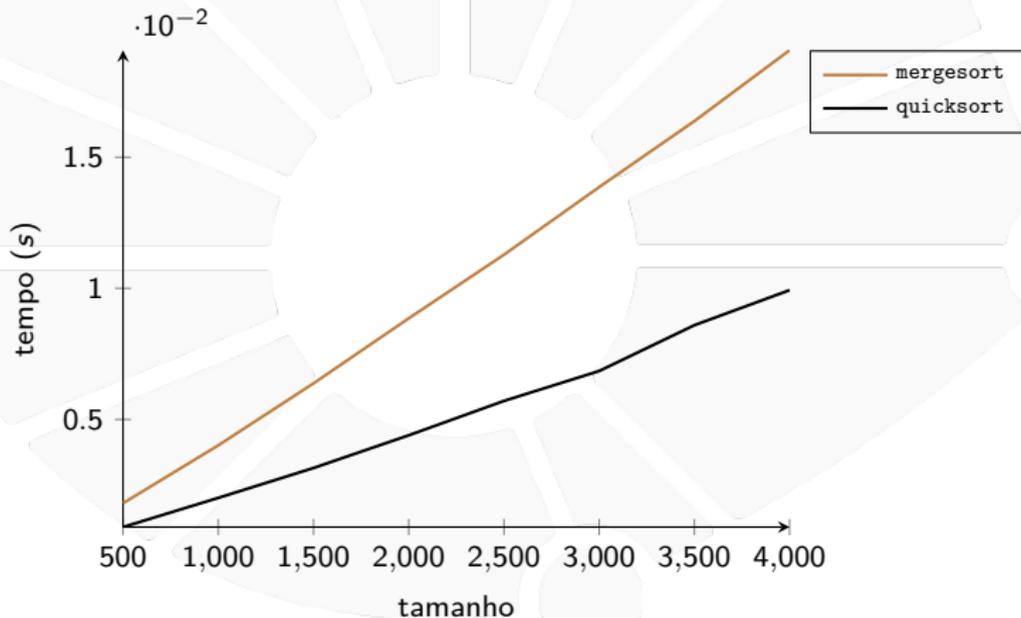
- ▶ Listas de tamanho 500, 1000, ..., 4000, com elementos aleatórios entre 0 e 1.
- ▶ Tiramos a média do tempo de 10 execuções.

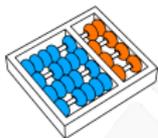




## Experimento — Apenas quick e mergesort

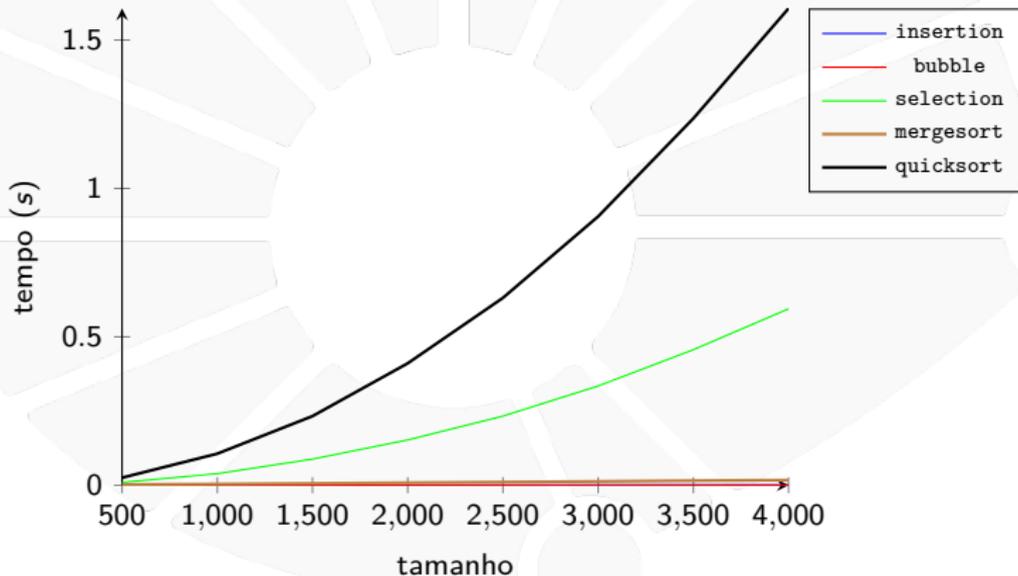
- ▶ Listas de tamanho 500, 1000, ..., 4000, com elementos aleatórios entre 0 e 1.
- ▶ Tiramos a média do tempo de 10 execuções.

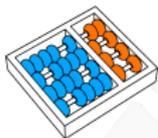




## Experimento 2

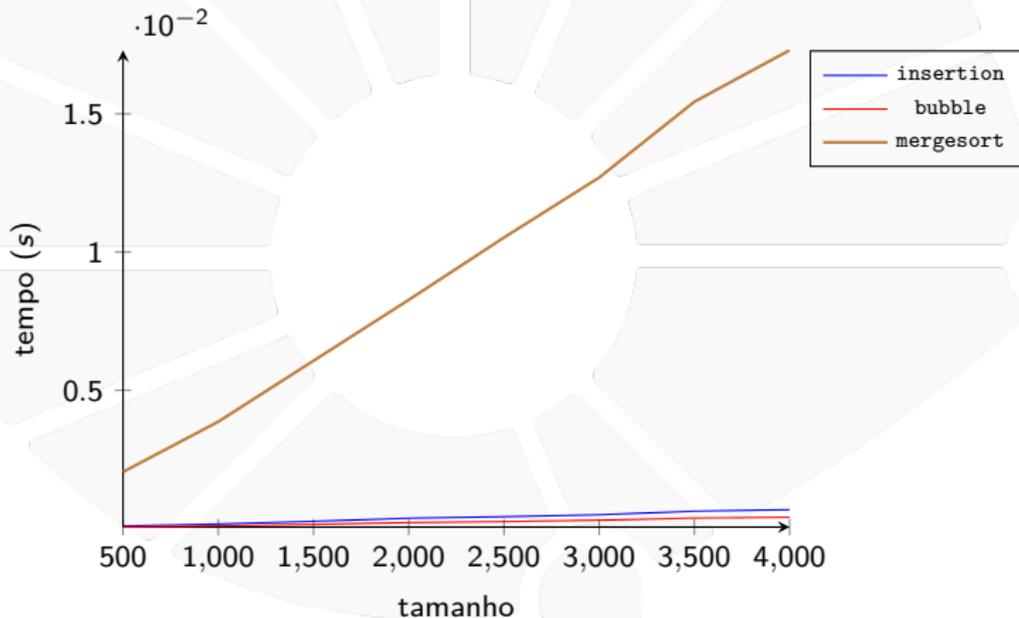
- ▶ Listas **ordenadas** de tamanho 500, 1000, ..., 4000.
- ▶ Tiramos a média do tempo de 10 execuções.





## Experimento 2

- ▶ Listas **ordenadas** de tamanho 500, 1000, ..., 4000.
- ▶ Tiramos a média do tempo de 10 execuções.



# RECURSÃO E ORDENAÇÃO

MC102 - Algoritmos e  
Programação de  
Computadores

Santiago Valdés Ravelo  
[https://ic.unicamp.br/~santiago/  
ravelo@unicamp.br](https://ic.unicamp.br/~santiago/ravelo@unicamp.br)

06/25

24



UNICAMP

