Recursão e ordenação

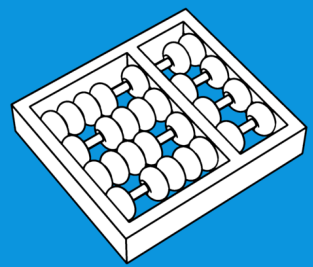
MC102 - Algoritmos e Programação de Computadores

06/24

22

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

UNICAMP

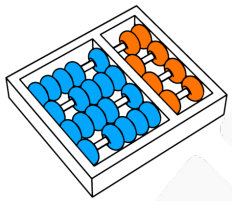


“Divide et impera”

Júlio César.

Lembrem de participar na OBI!!!

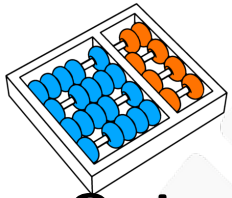
Dúvidas da aula anterior



Dúvidas selecionadas

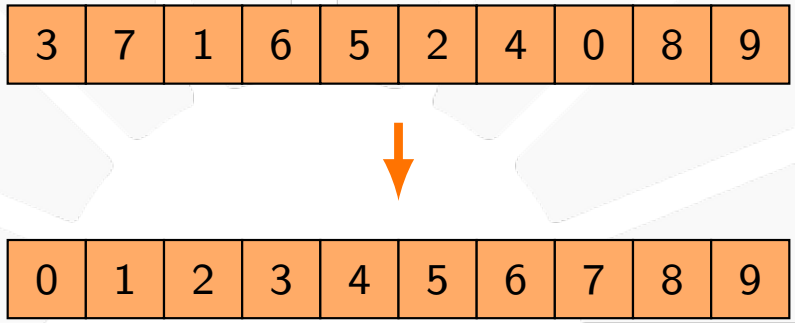
- Qual o mais rápido, a recursão ou o for.
- Esses programas com recursão, como o que calcula a sequência de Fibonacci, não seriam mais lentos que aqueles que usam for? Há alguma outra razão, além de deixar o código mais limpo, para escolher recursão em vez de um loop?
- Nessa aula em específico as funções que apareceram eram, em geral, mais eficientes se programadas com um loop, mas qual seria um exemplo de algoritmo que seria mais eficiente se pensado de forma recursiva?
- Como saber se é melhor/mais eficiente usar a recursão ao invés de uma iteração em um problema?
- Existem problemas que podem ser resolvidos somente por meio da recursão? Ou ela é só uma ferramenta pra conveniência do programador?
- Em quais cenários a recursão é inevitável ou mais natural do que a iteração?
- A recursão é muito exigente do ponto de vista da memória? Criar um novo "prato" na pilha ocupa mais memória, certo?
- Em um algoritmo recursivo pode ocorrer "stack over flow" mesmo que o algoritmo esteja correto? Por exemplo, se quiséssemos calcular o fatorial de 1 trilhão, poderia ocorrer "stack over flow" por conta da limitação da memória do computador que está rodando o código?
- Se eu faço uma recursão onde a função é chamada 1000 vezes eu tenho um recursion error. Como eu posso "burlar" esse máximo e fazer mais de 1000?
- Até onde eu sei, existe um tipo de recursão chamada recursão de cauda, qual a diferença entre ela e a recursão vista em aula?
- Além de programação dinâmica, que recursos otimizam uma recursão?
- Existem desvantagens em utilizar recursão?
- Poderia dar mais aplicações de recursividade? Por exemplo, quando eu usaria isso em um projeto real?

Lembrando



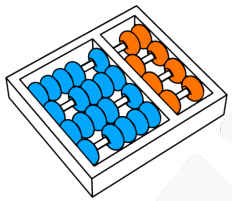
Ordenação

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



Vimos três algoritmos:

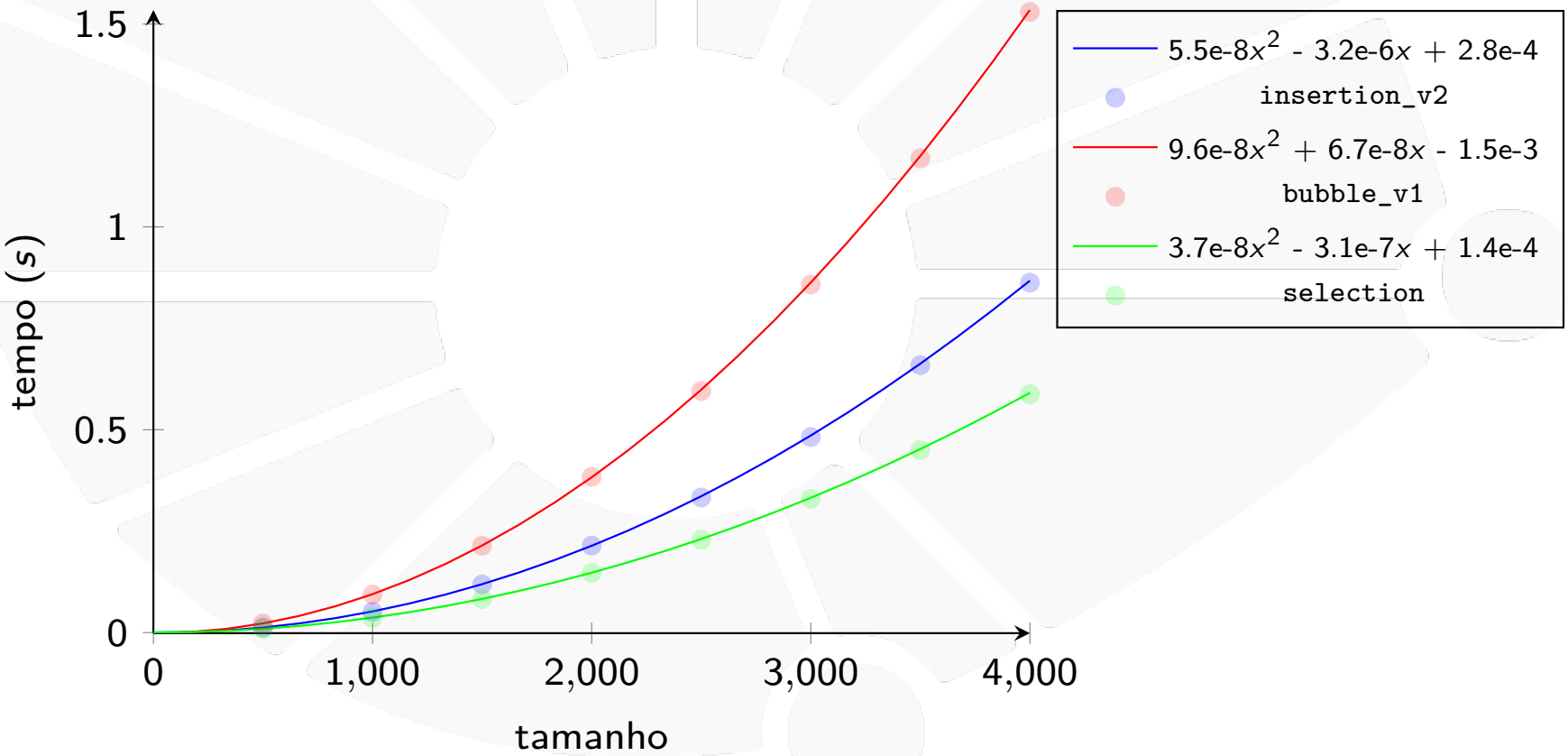
- **SelectionSort**: Seleciona o i-ésimo menor elemento e coloca na posição i.
- **BubbleSort**: Faz varias passagens do final para o começo trocando pares invertidos.
- **InsertionSort**: Insere o i-ésimo elemento na posição correta.



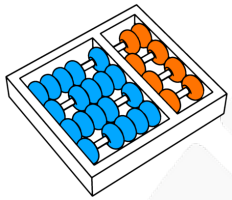
Experimento

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.



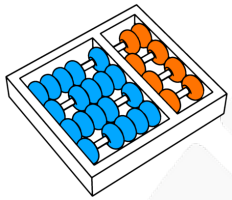
Ordenação recursiva



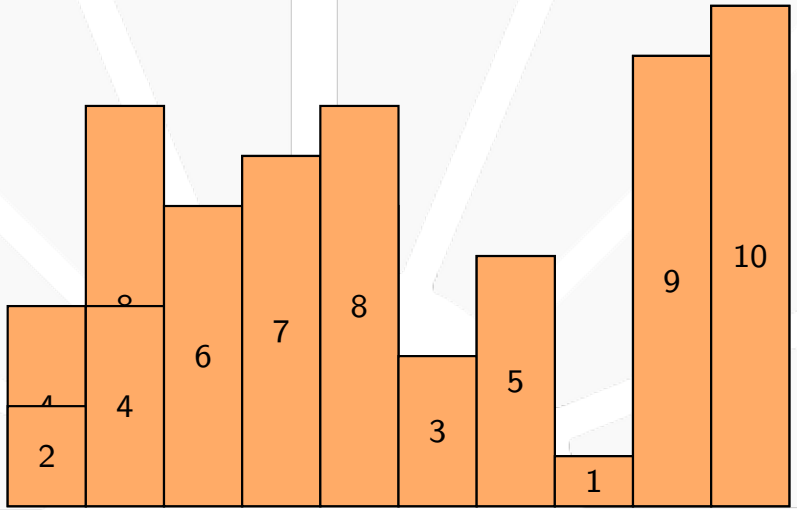
Outros algoritmos

Na aula de hoje veremos:

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



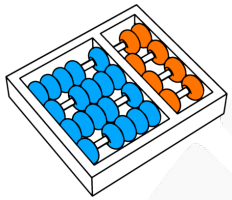
Estratégia: Recursão:



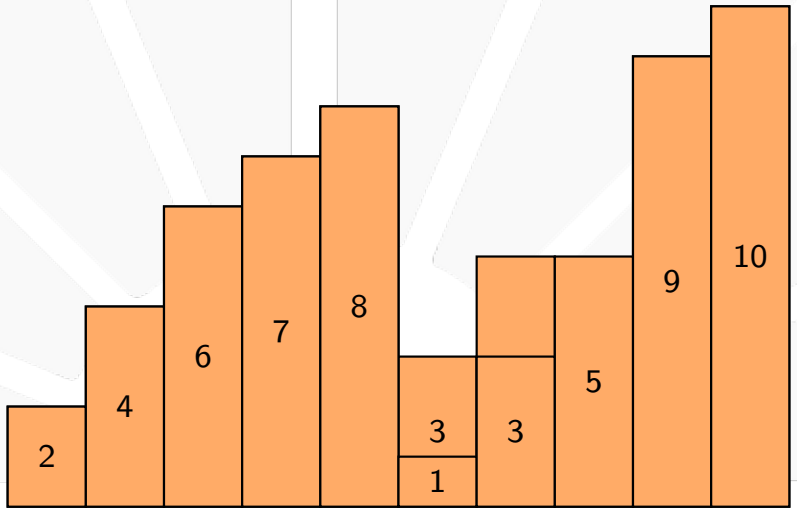
Como ordenar a primeira metade da lista?

- 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?

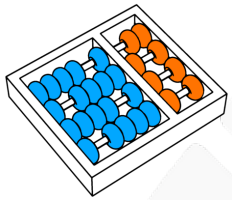


Ordenando a segunda parte



Para ordenar a segunda metade:

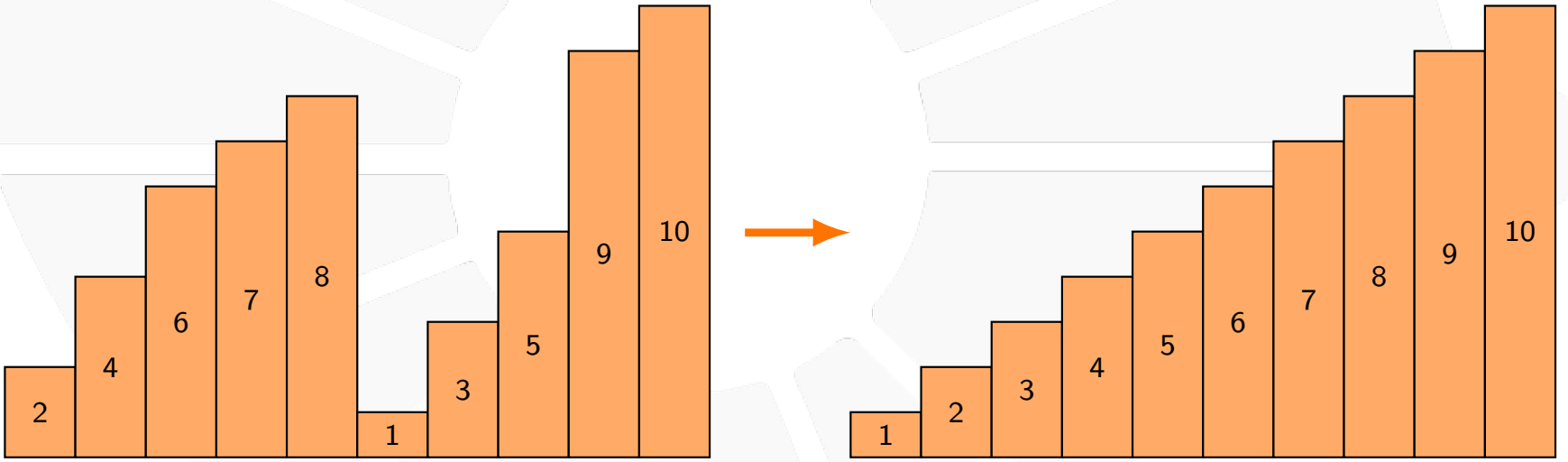
- Executamos **ordenar(l, 5, 9)**.

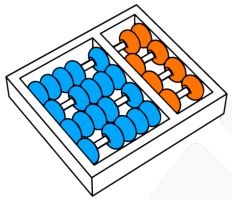


Ordenando toda a lista

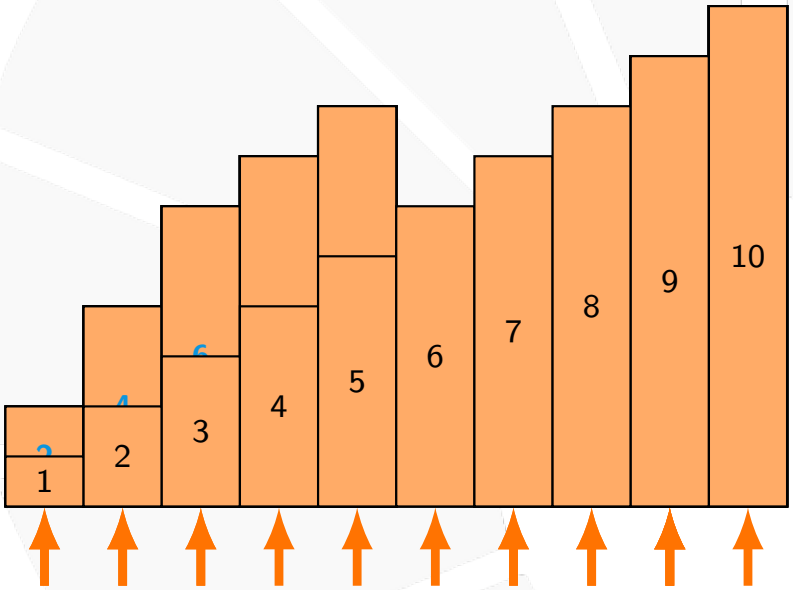
Se temos um lista com as suas duas metades já ordenadas:

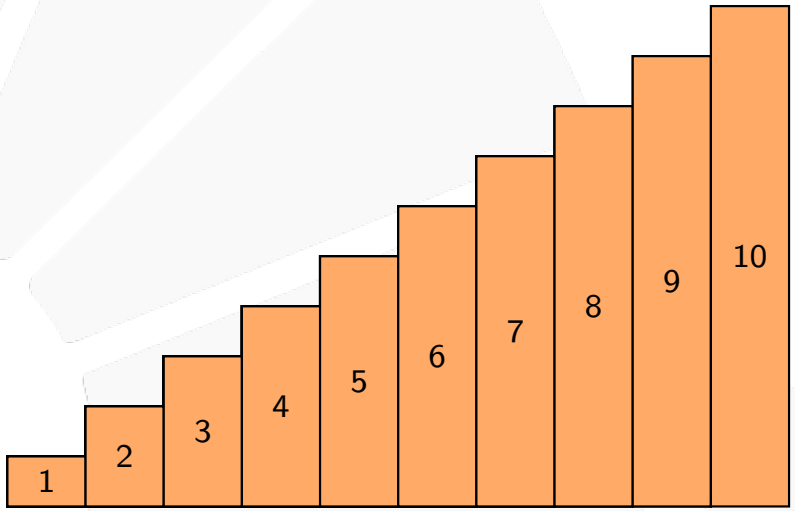
- Como ordenar toda a lista?



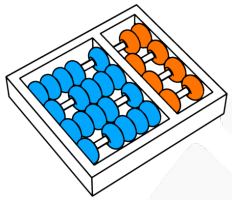


Intercalando





- 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

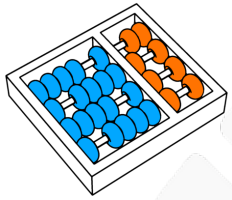
Observação:

- 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.

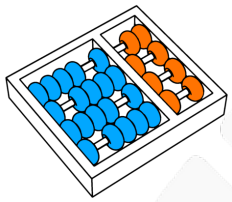
MergeSort



Ordenação por intercalação (MergeSort)

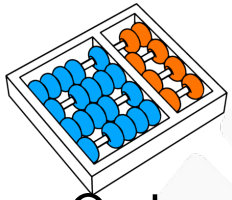
Intercalação:

- 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:  # Copia o restante da primeira metade
12         aux.append(l[i])
13         i += 1
14     while j <= d:  # Copia o restante da segunda metade
15         aux.append(l[j])
16         j += 1
17     for i in range(e, d + 1): # Copia de volta
18         l[i] = aux[i - e]
```

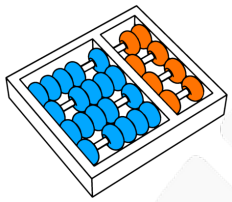


Ordenação por intercalação (MergeSort)

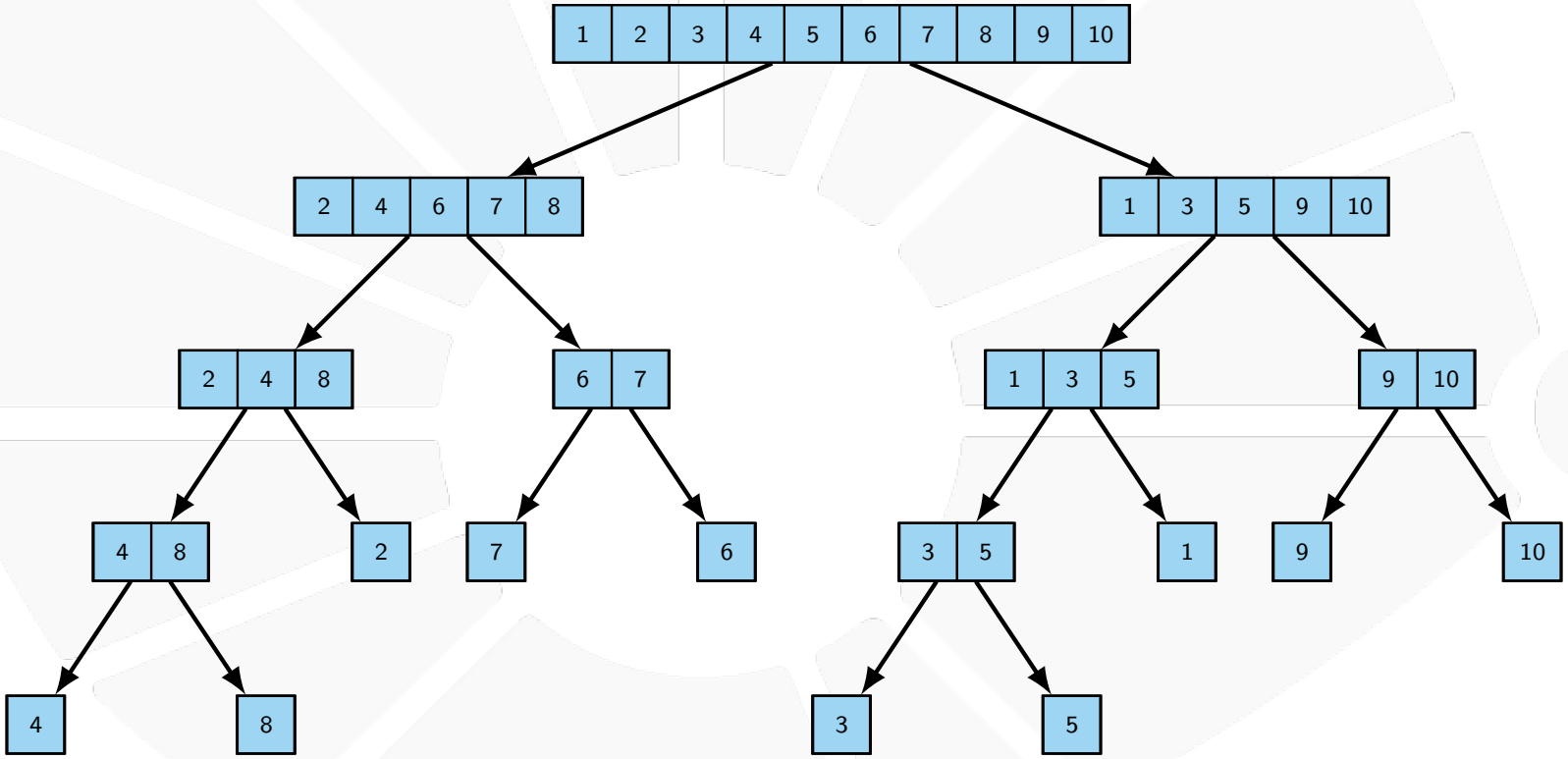
Ordenação:

- 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!

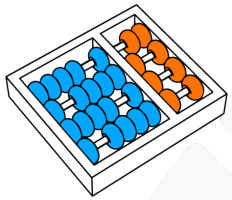
```
1 def mergesort(l, e, d):
2     if e < d:
3         m = (e + d) // 2
4         mergesort(l, e, m)
5         mergesort(l, m + 1, d)
6         merge(l, e, m, d)
```



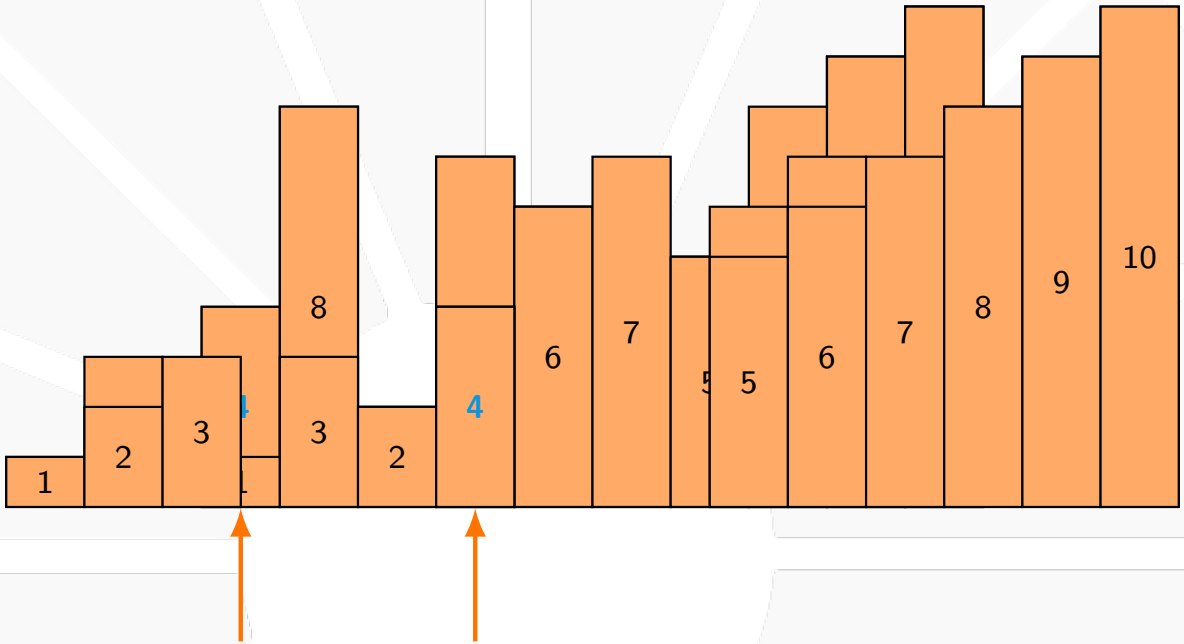
Simulação



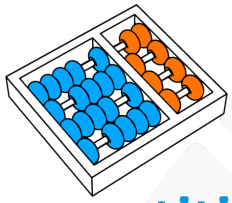
QuickSort



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**.



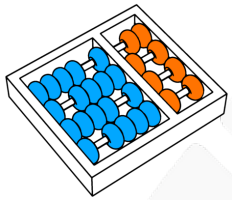
Quicksort

partition(l, e, d):

- 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)
```

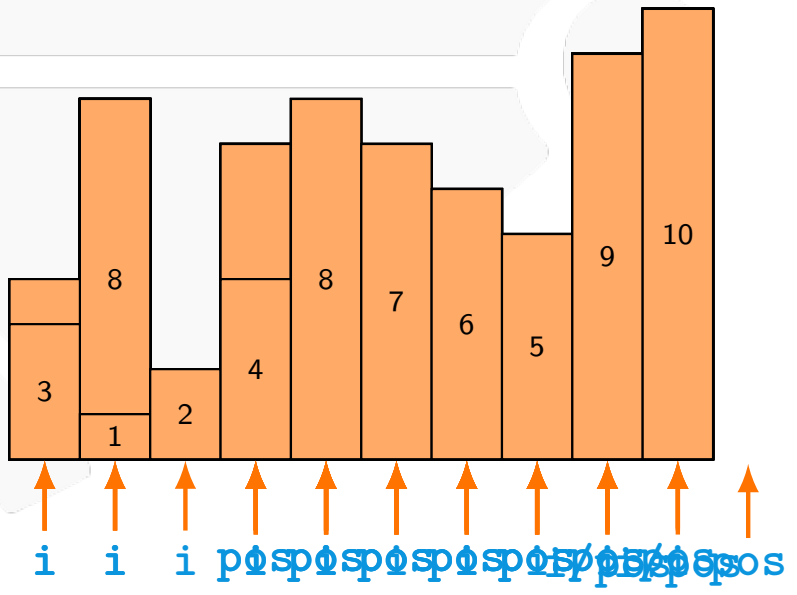
- Basta particionar a lista em duas.
- Depois, ordenar o lado esquerdo e o direito.

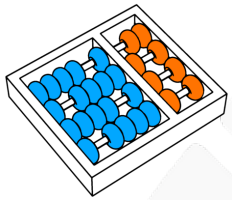


Como particionar uma lista?

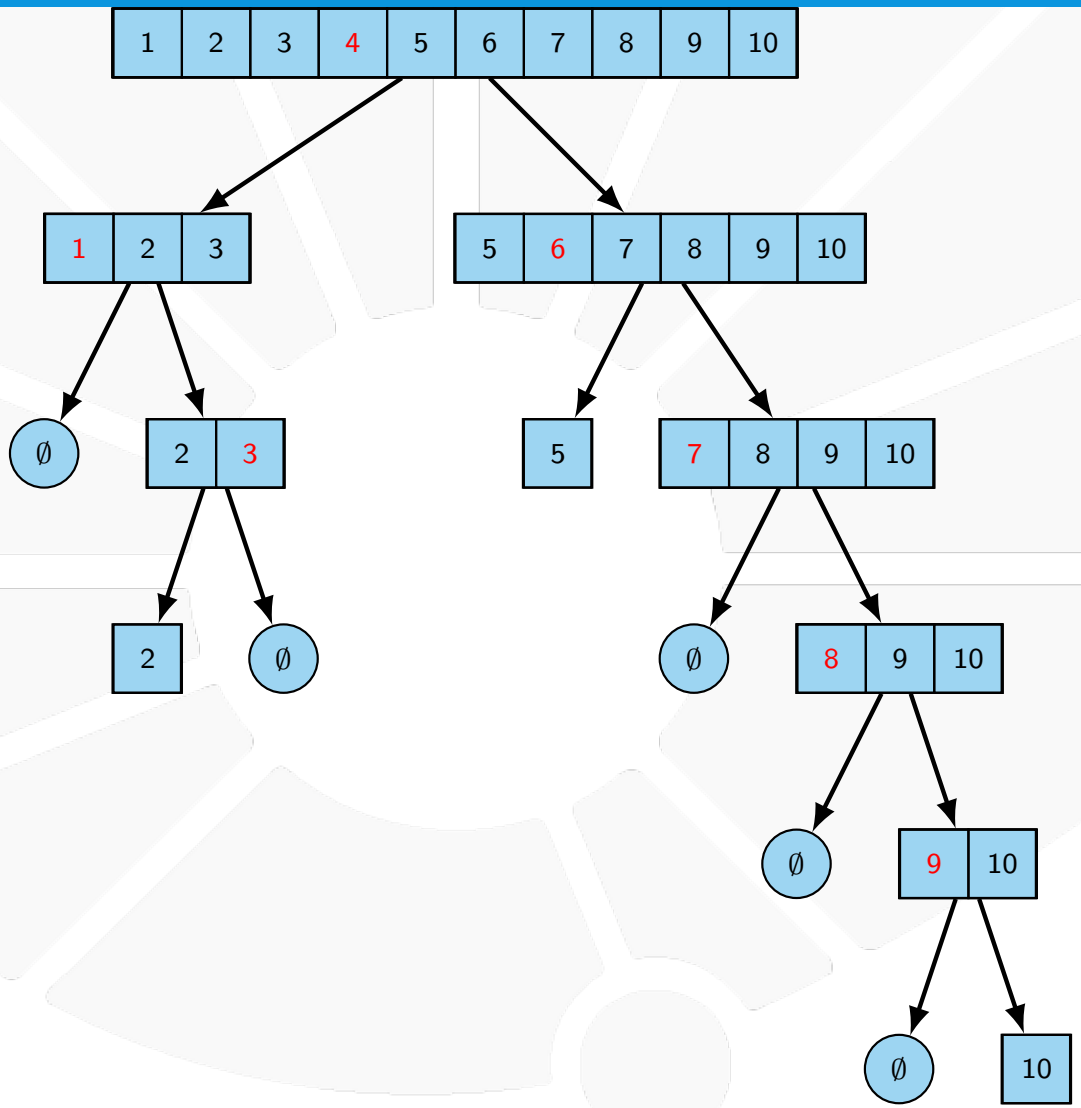
- Andamos da direita para a esquerda com um índice **i**.
- De **i** até **pos - 1** ficam os menores do que o pivô.
- De **pos** até **d** ficam os maiores ou iguais ao pivô.
- Se o elemento em **i** for maior ou igual ao pivô:
  - Diminuímos **pos** e realizamos uma troca de **i** com **pos**.
- No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

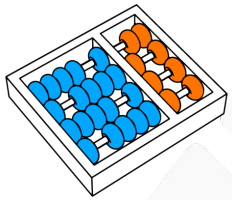




Simulação do Quicksort

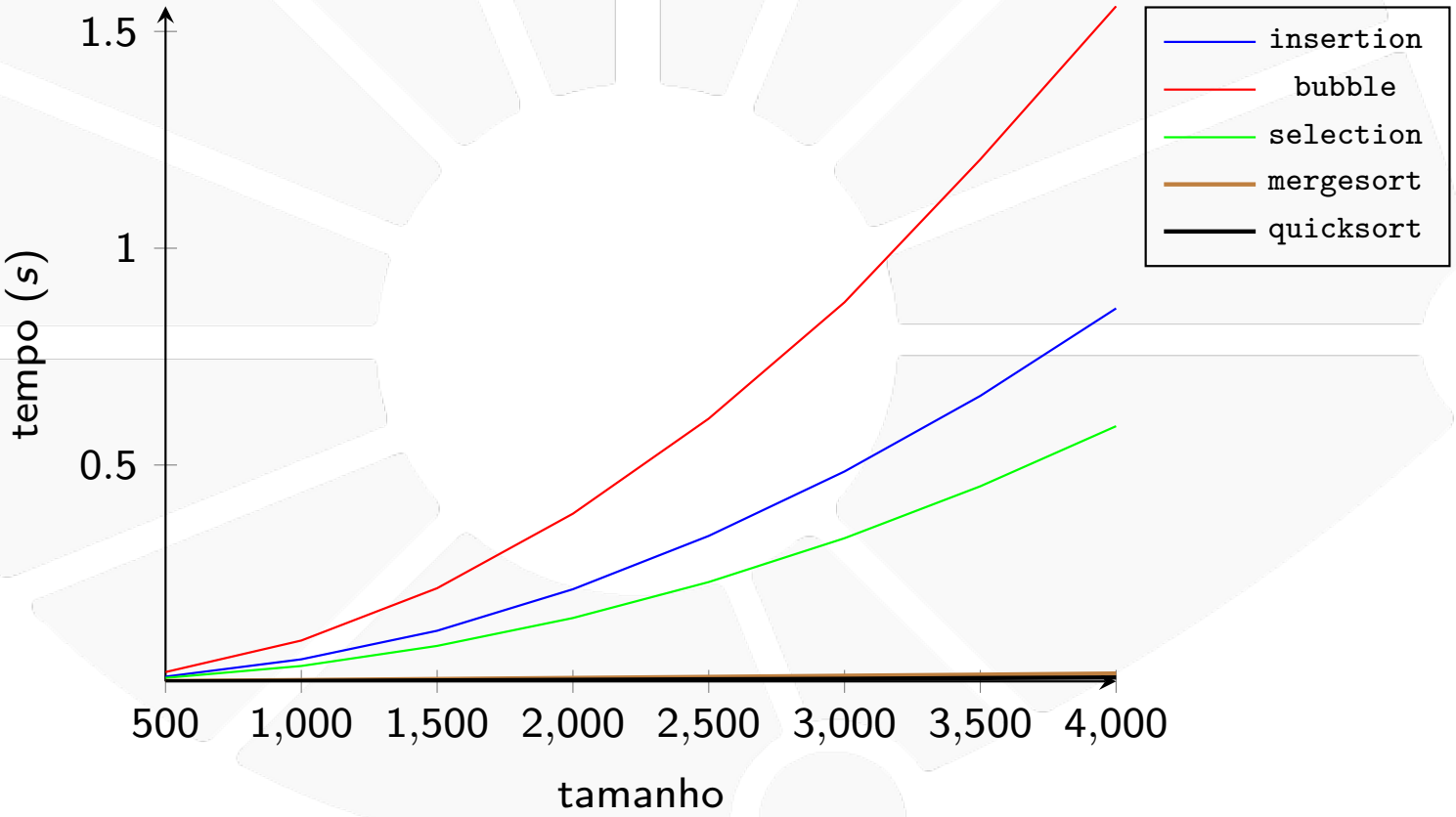


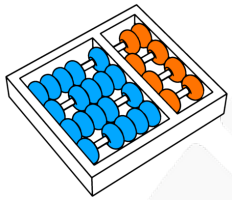
Experimentos



Experimento

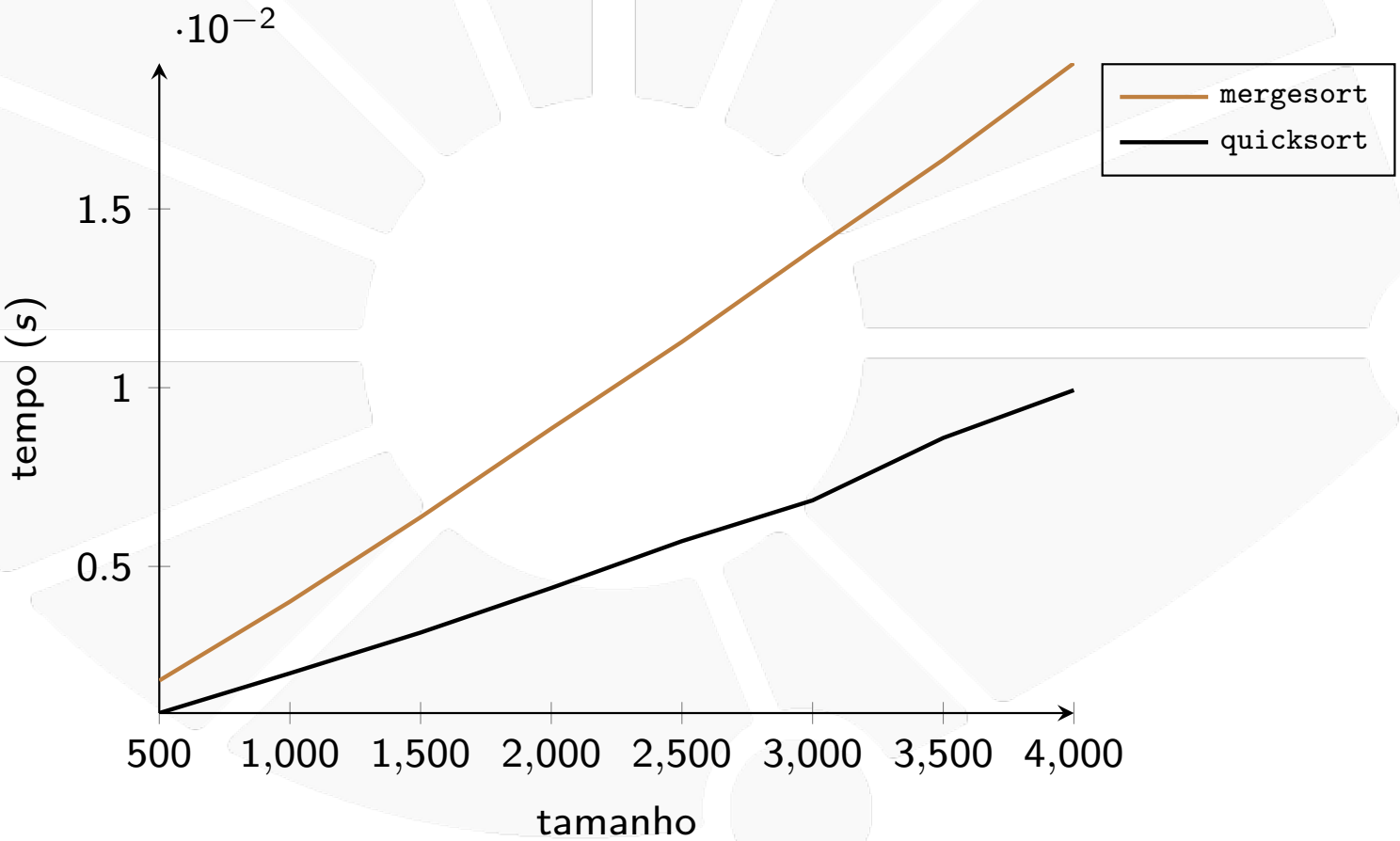
- Listas de tamanho 500, 1000, . . . , 4000, com elementos aleatórios entre 0 e 1.
- Tiramos a média do tempo de 10 execuções.

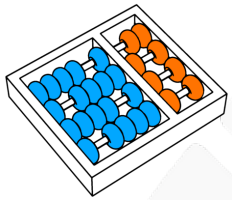




Experimento — Apenas quick e mergesort

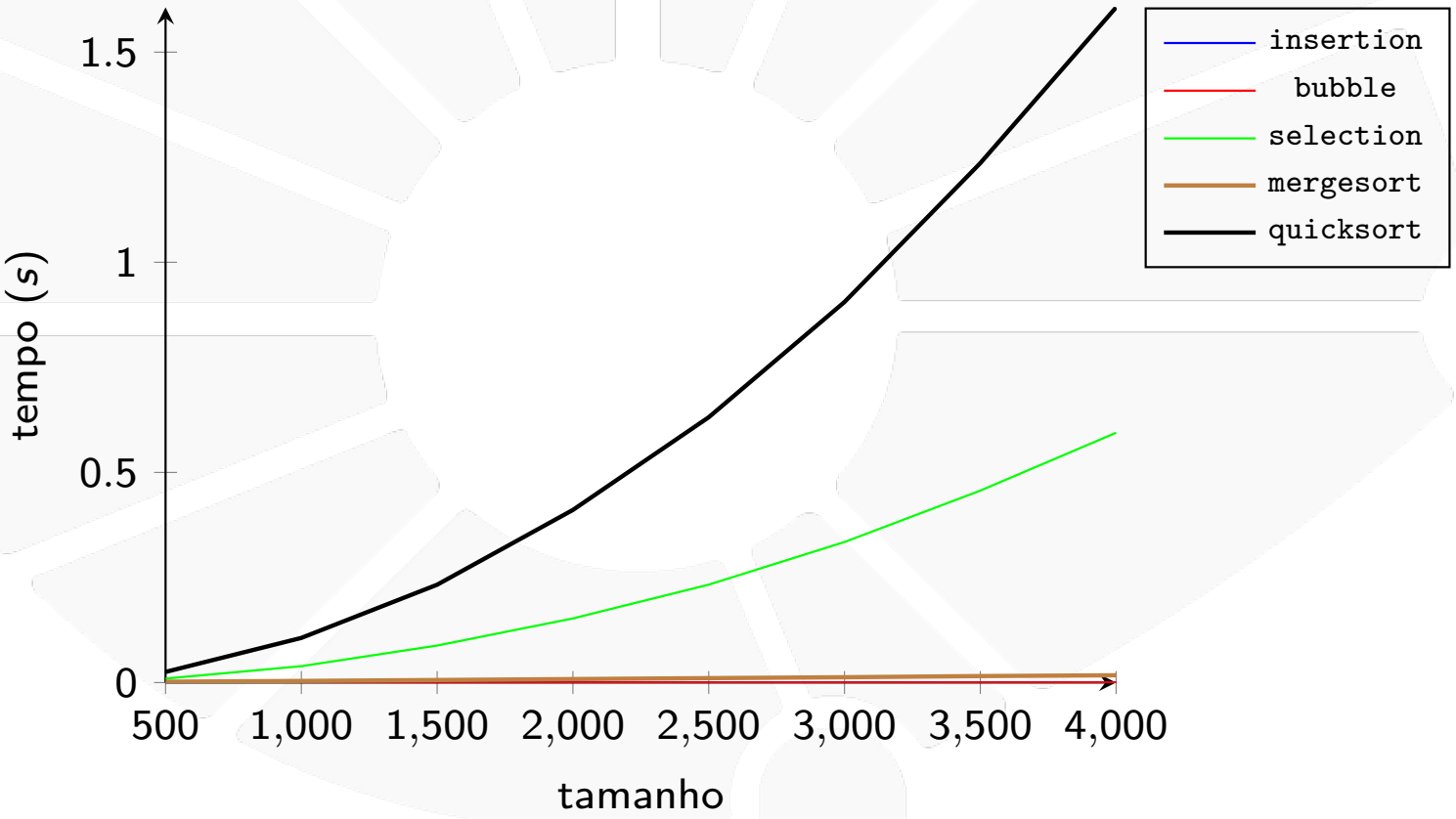
- Listas de tamanho 500, 1000, . . . , 4000, com elementos aleatórios entre 0 e 1.
- Tiramos a média do tempo de 10 execuções.

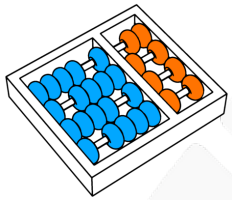




Experimento 2

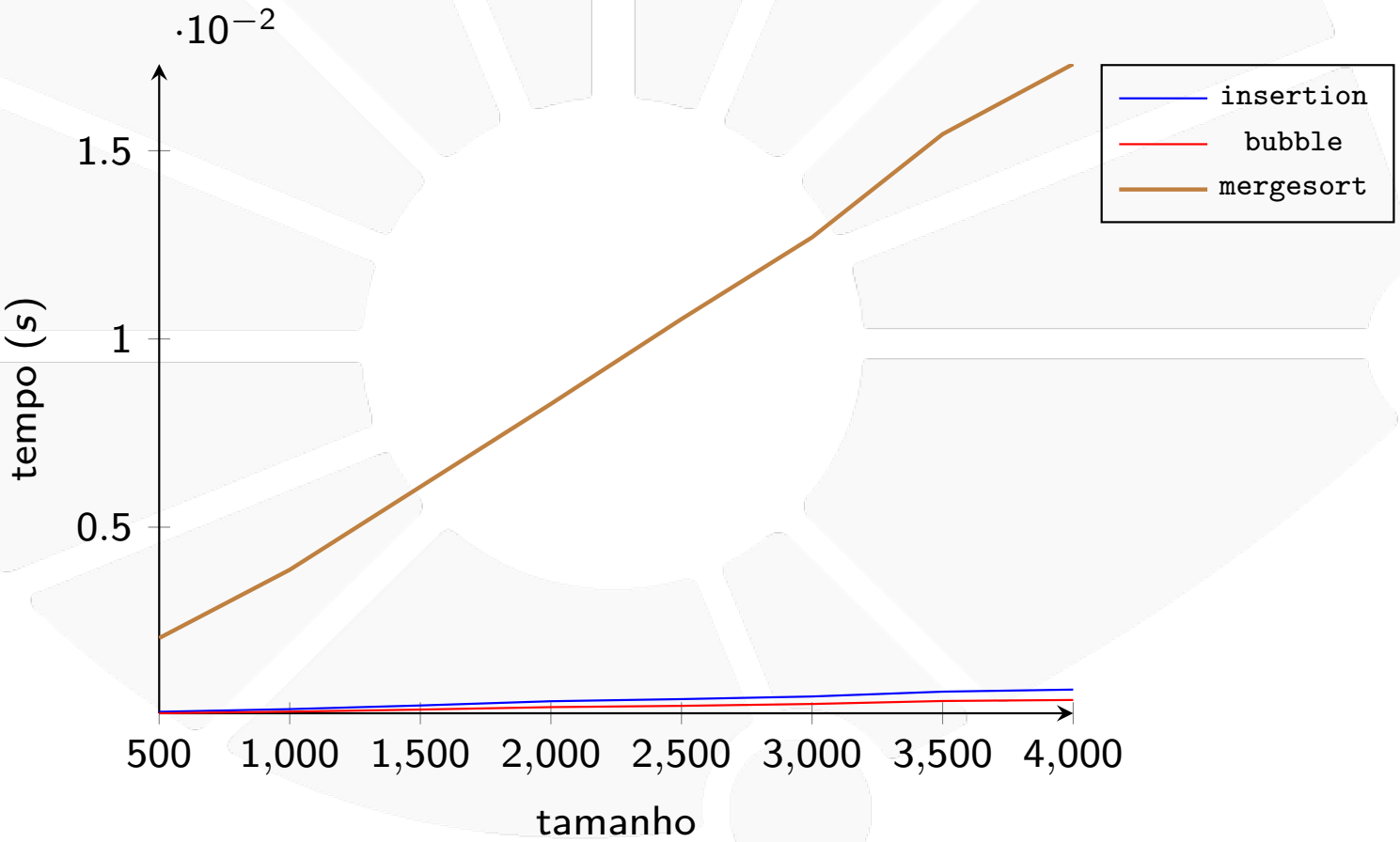
- Listas **ordenadas** de tamanho 500, 1000, . . . , 4000.
- Tiramos a média do tempo de 10 execuções.





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

06/24

22

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

UNICAMP