MC102 — Algoritmos de Ordenação Recursivos

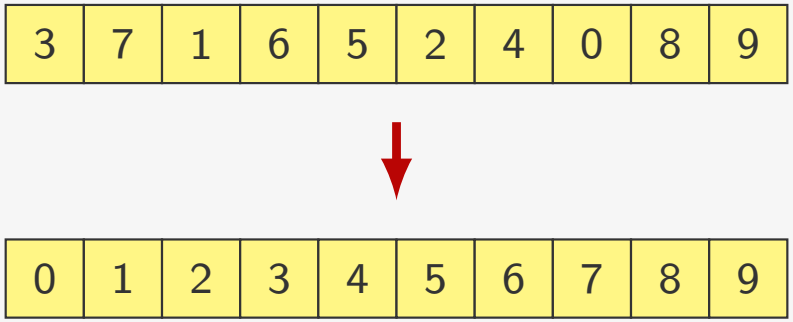
Rafael C. S. Schouery
rafael@ic.unicamp.br

Universidade Estadual de Campinas

Atualizado em: 2023-03-02 14:30

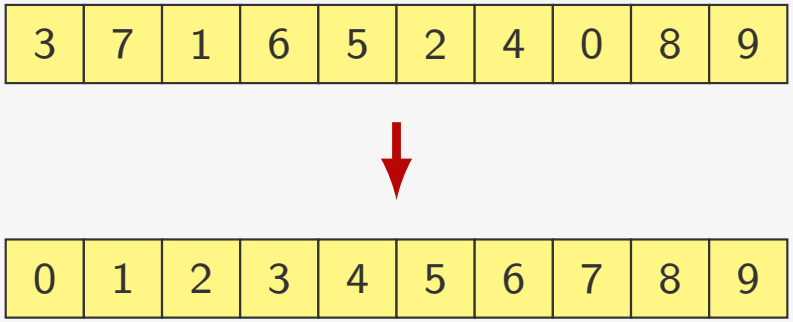
Ordenação

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



Ordenação

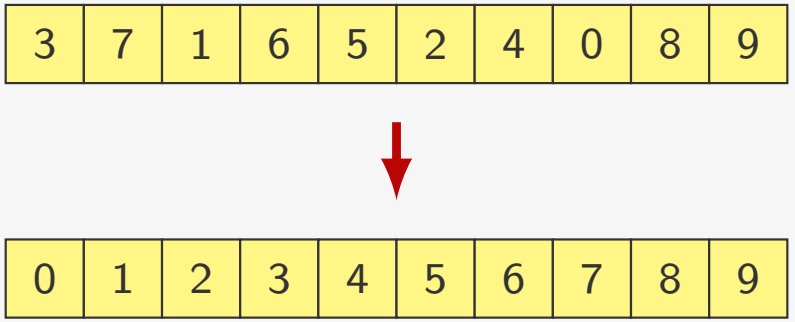
Ordenação: Dada uma lista l de n elementos, rearranjar os elementos de l de forma que $l[1] \leq l[2] \leq \cdots \leq l[n]$.



Vimos três algoritmos:

Ordenação

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

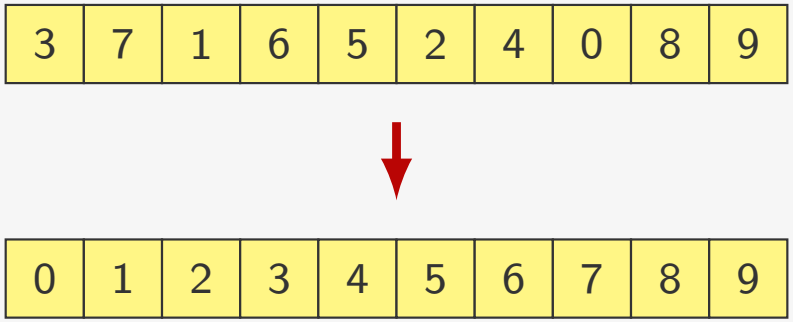


Vimos três algoritmos:

- `selectionsort`: seleciona o i-ésimo menor elemento e coloca na posição i

Ordenação

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

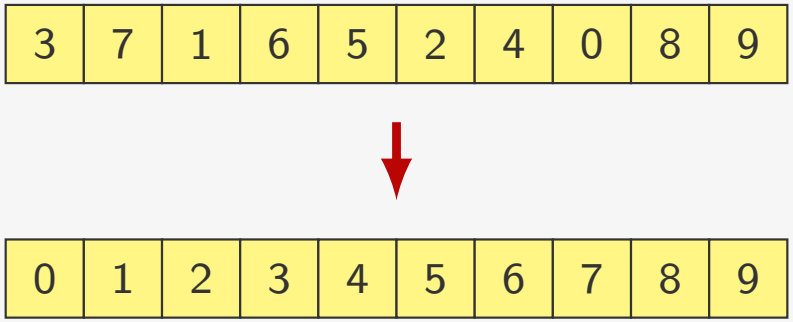


Vimos três algoritmos:

- `selectionsort`: seleciona o i-ésimo menor elemento e coloca na posição i
- `bubblesort`: fazemos varias 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 $l[1] \leq l[2] \leq \cdots \leq l[n]$.



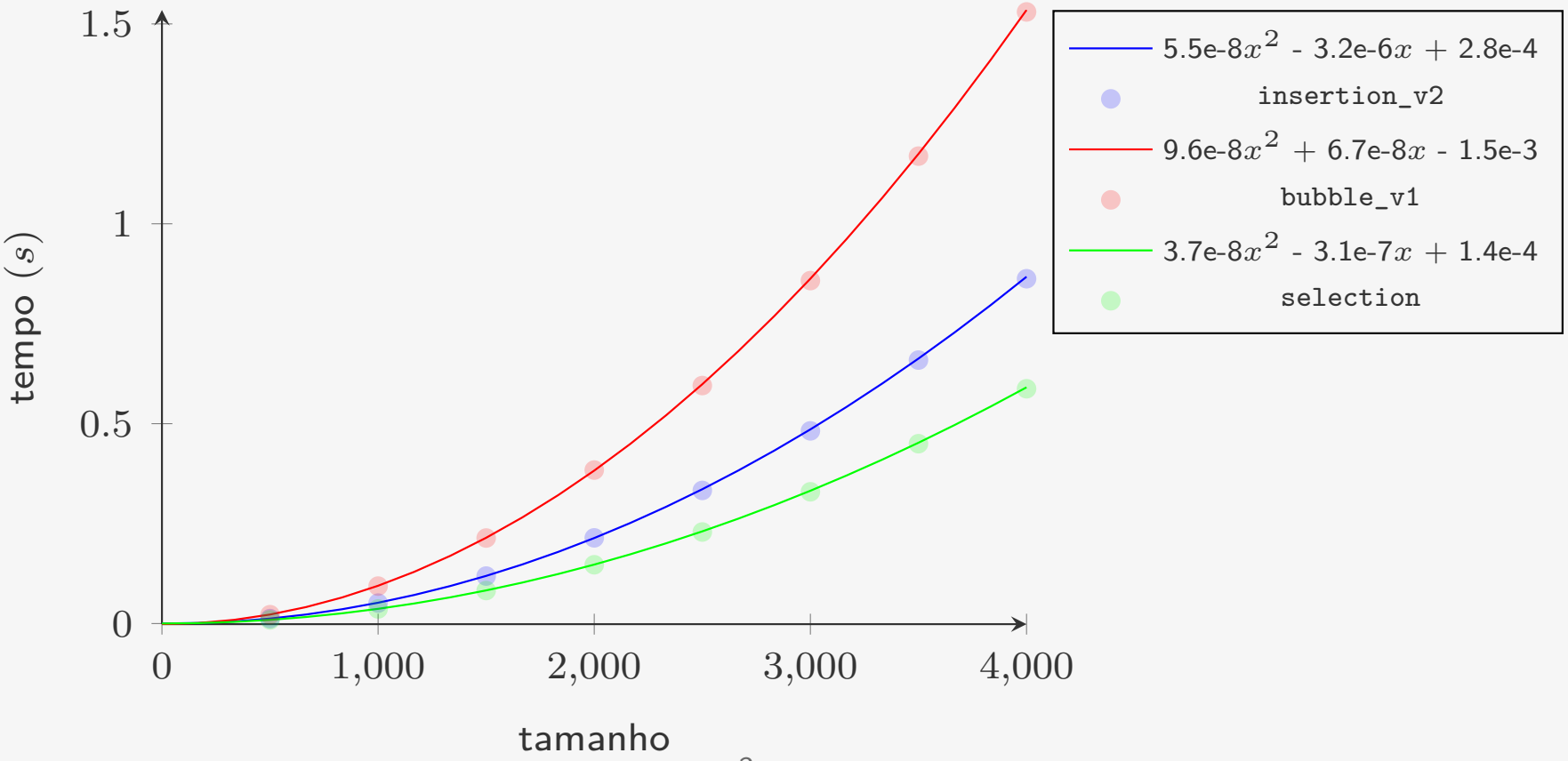
Vimos três algoritmos:

- `selectionsort`: seleciona o i-ésimo menor elemento e coloca na posição i
- `bubblesort`: fazemos varias passagens do final para o começo trocando pares invertidos
- `insertionsort`: inserimos o i-ésimo elemento na posição correta

Experimento

Tempo cresce quadraticamente com o tamanho da lista

- Listas de tamanho 100, 200, ..., 4000
- Elemento da lista escolhido aleatoriamente entre 0 e 1
- Tiramos a média do tempo de 10 execuções



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:

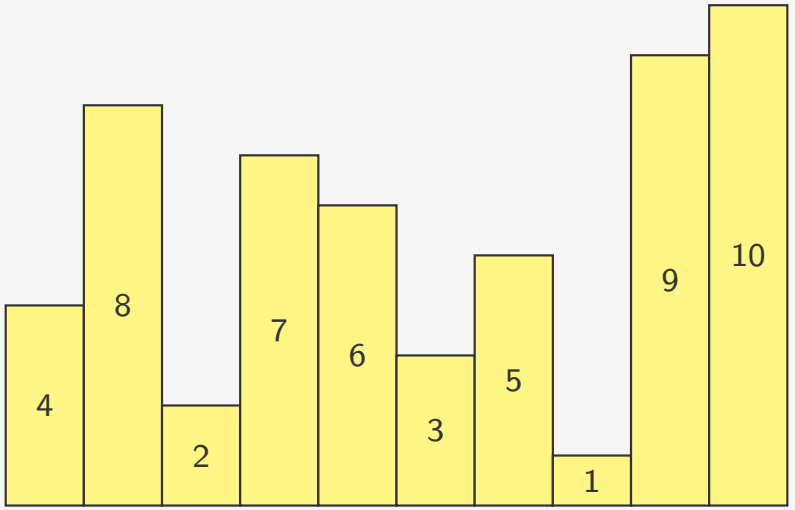
- Dois outros algoritmos de ordenação
- Baseados em recursão

Outros algoritmos

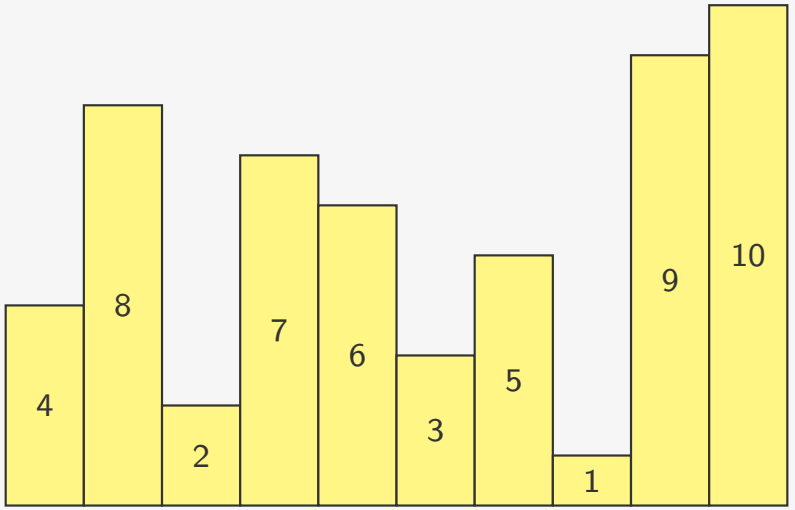
Na aula de hoje veremos:

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

Estratégia: Recursão

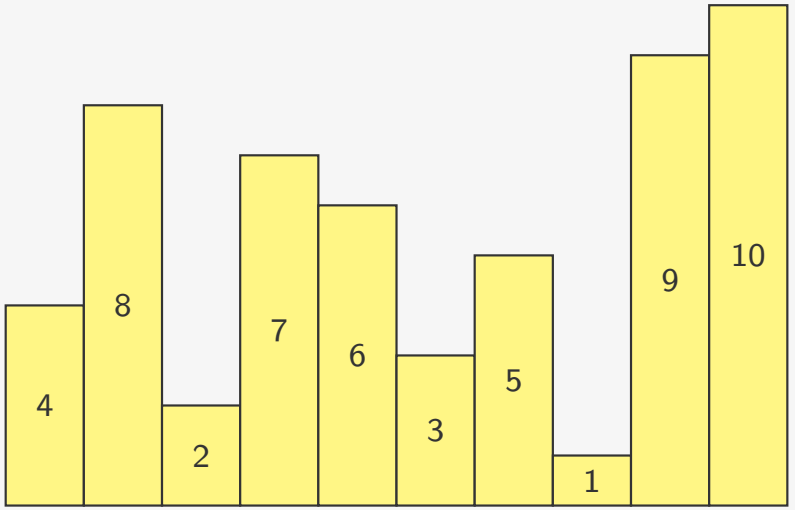


Estratégia: Recursão



Como ordenar a primeira metade da lista?

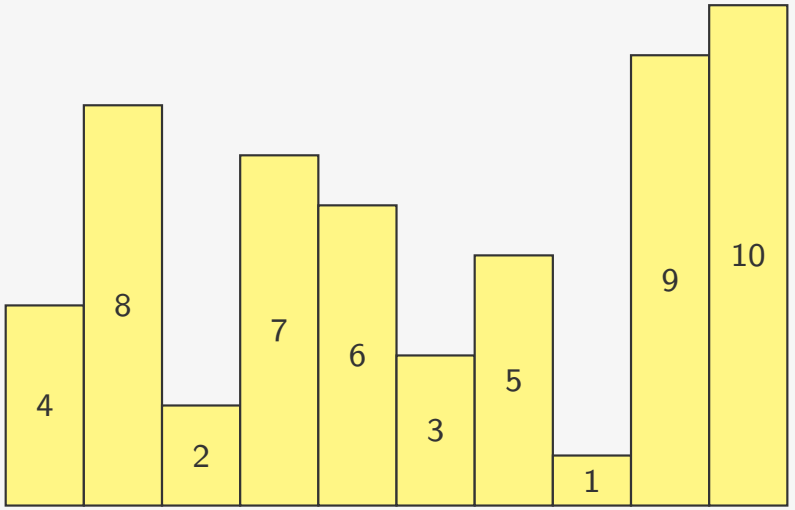
Estratégia: Recursão



Como ordenar a primeira metade da lista?

- usamos uma função `ordenar(l, e, d)`

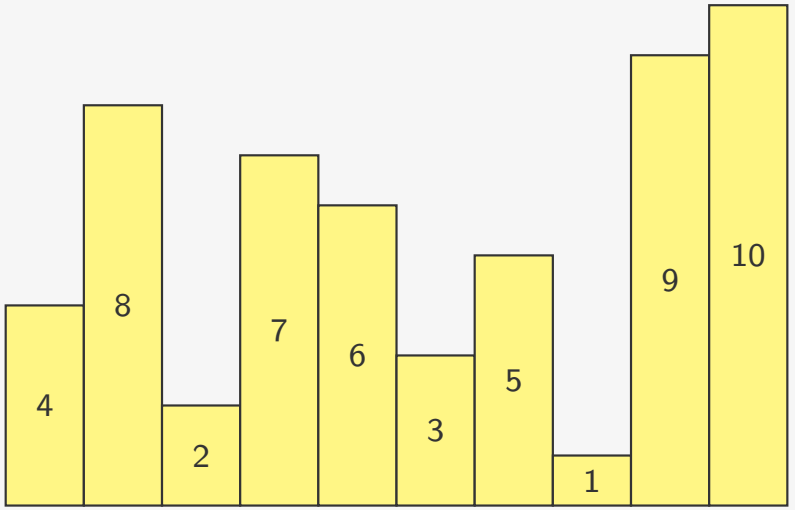
Estratégia: Recursão



Como ordenar a primeira metade da lista?

- usamos uma função `ordenar(l, e, d)`
  - ordena a lista `l` das posições `e` a `d` (inclusive)

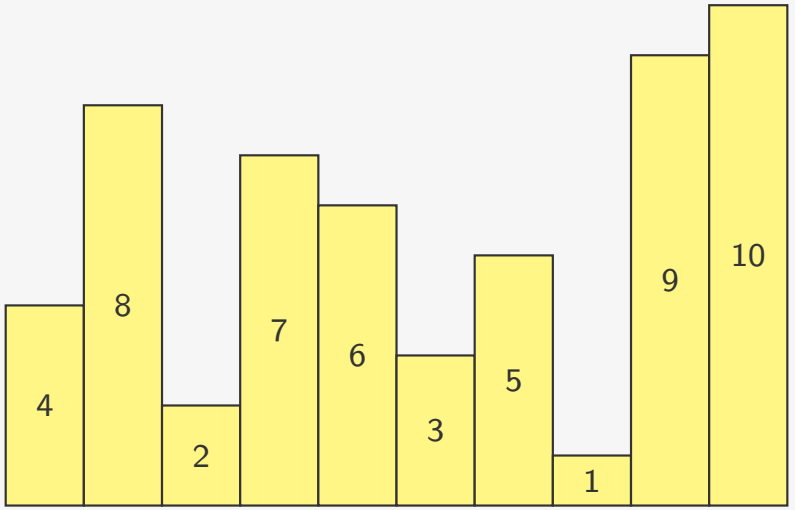
Estratégia: Recursão



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

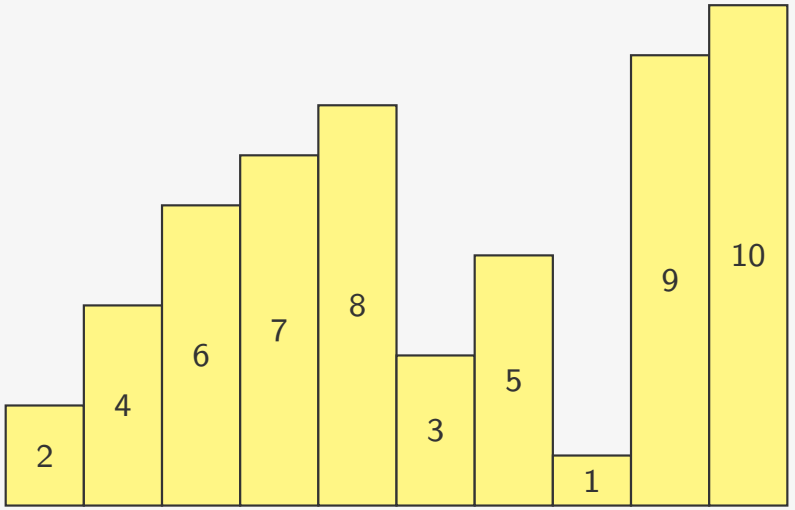
Estratégia: Recursão



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!

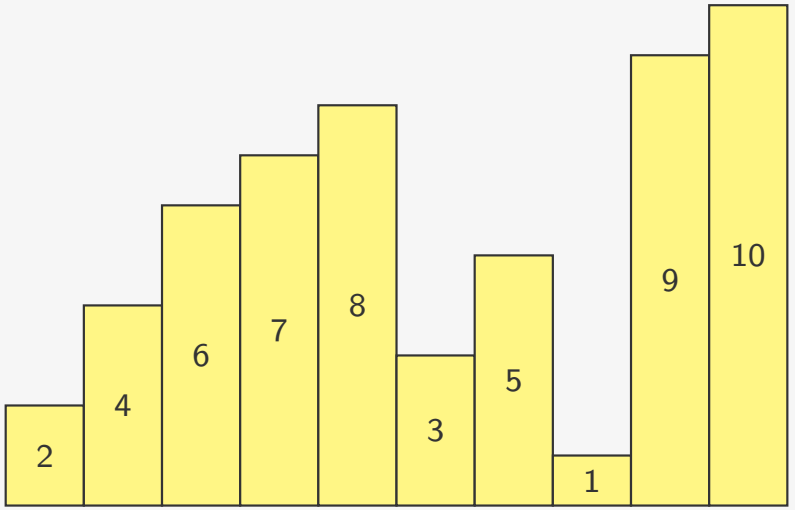
Estratégia: Recursão



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

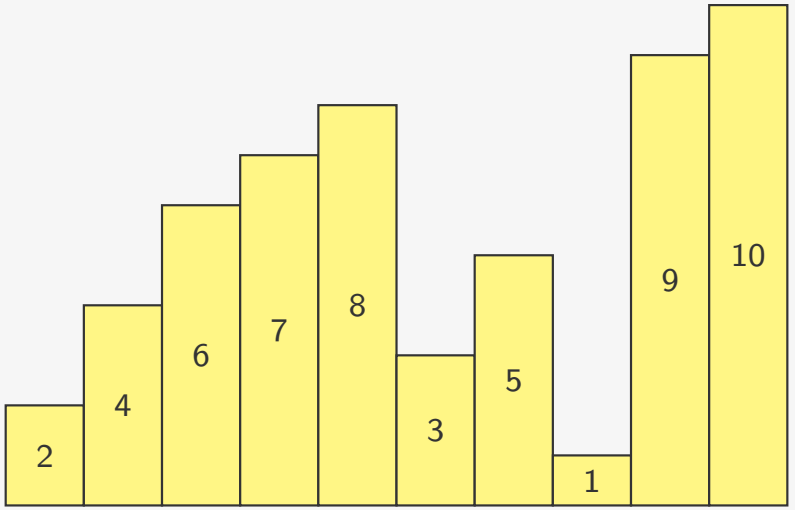


Como ordenar a primeira metade da lista?

- 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?

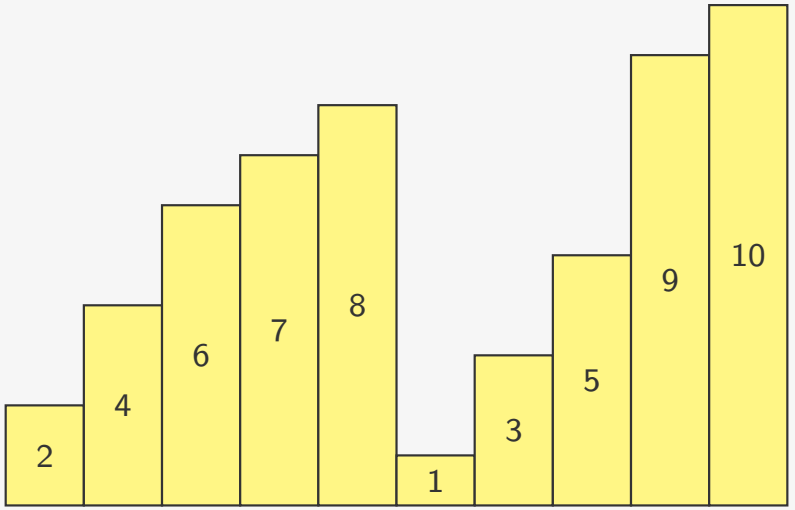
Ordenando a segunda parte



Para ordenar a segunda metade:

- executamos `ordenar(l, 5, 9)`

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

Ordenando toda a lista

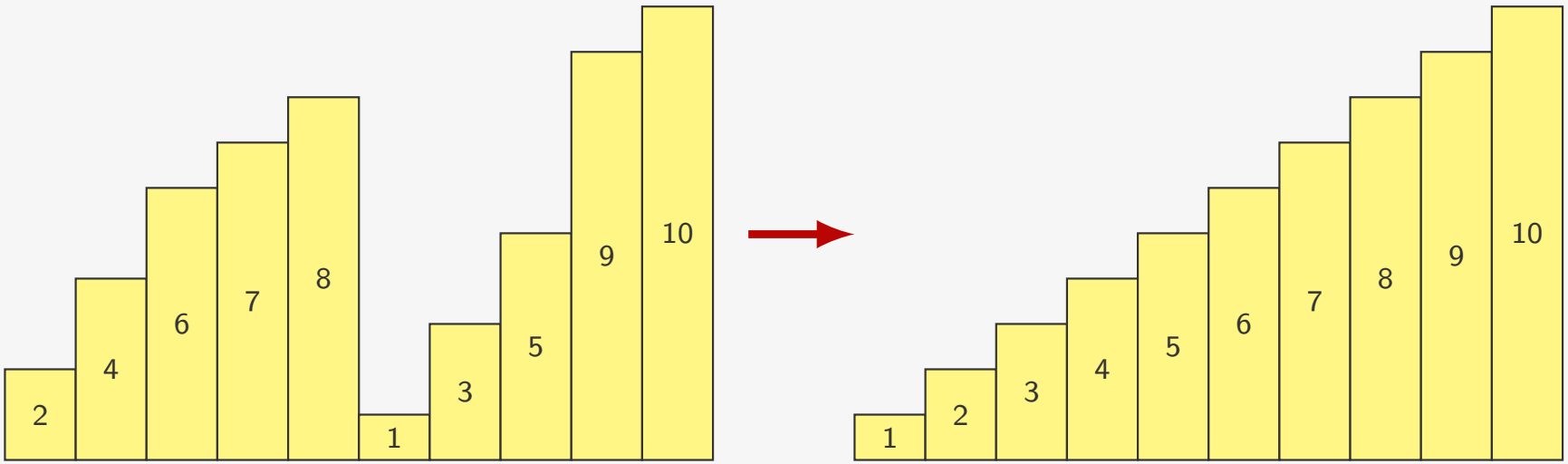
Se temos um lista com as suas duas metades já ordenadas

- Como ordenar toda a lista?

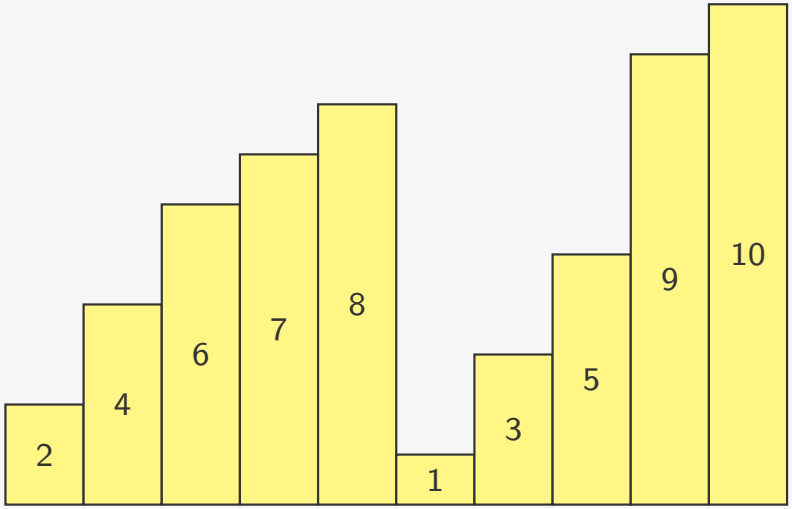
Ordenando toda a lista

Se temos um lista com as suas duas metades já ordenadas

- Como ordenar toda a lista?

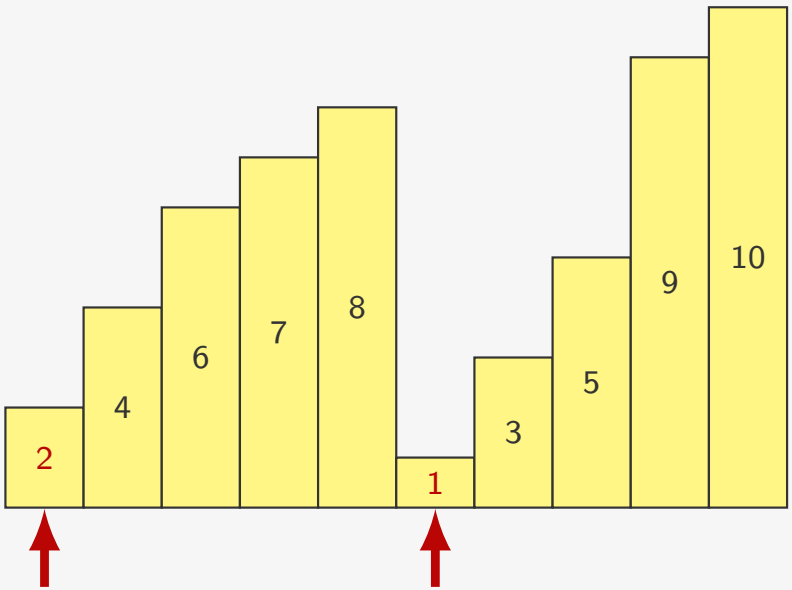


Intercalando



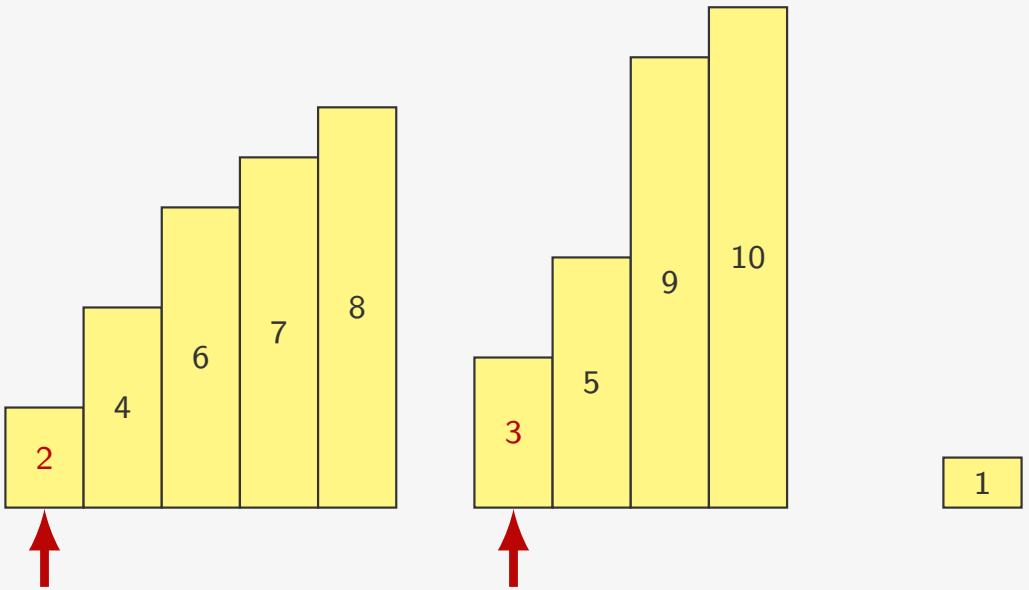
- Percorremos as duas sub-listas

Intercalando



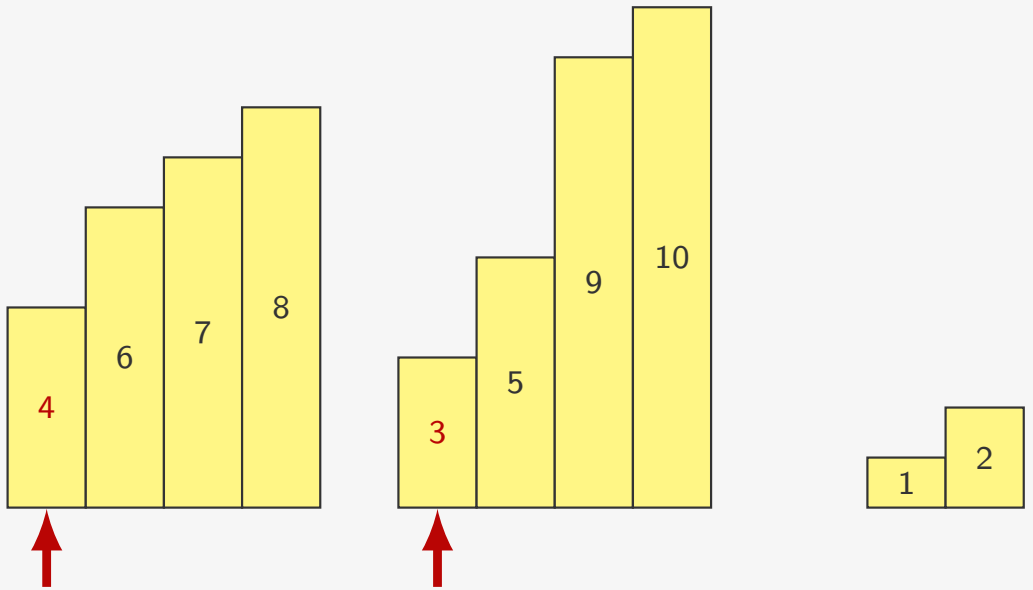
- Percorremos as duas sub-listas
- Pegamos o mínimo e inserimos em uma lista auxiliar

Intercalando



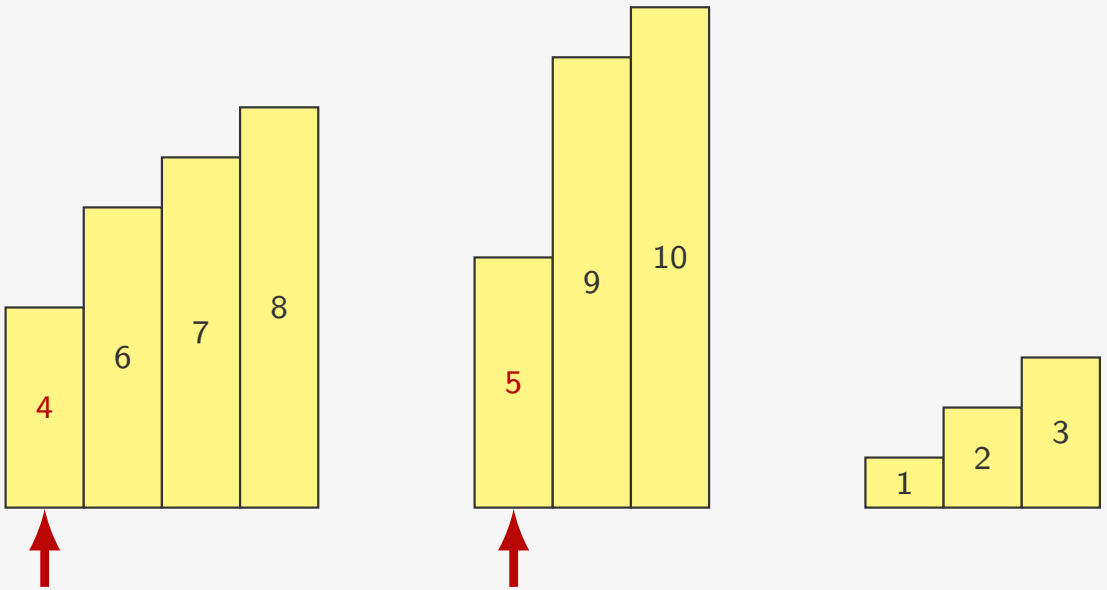
- Percorremos as duas sub-listas
- Pegamos o mínimo e inserimos em uma lista auxiliar

Intercalando



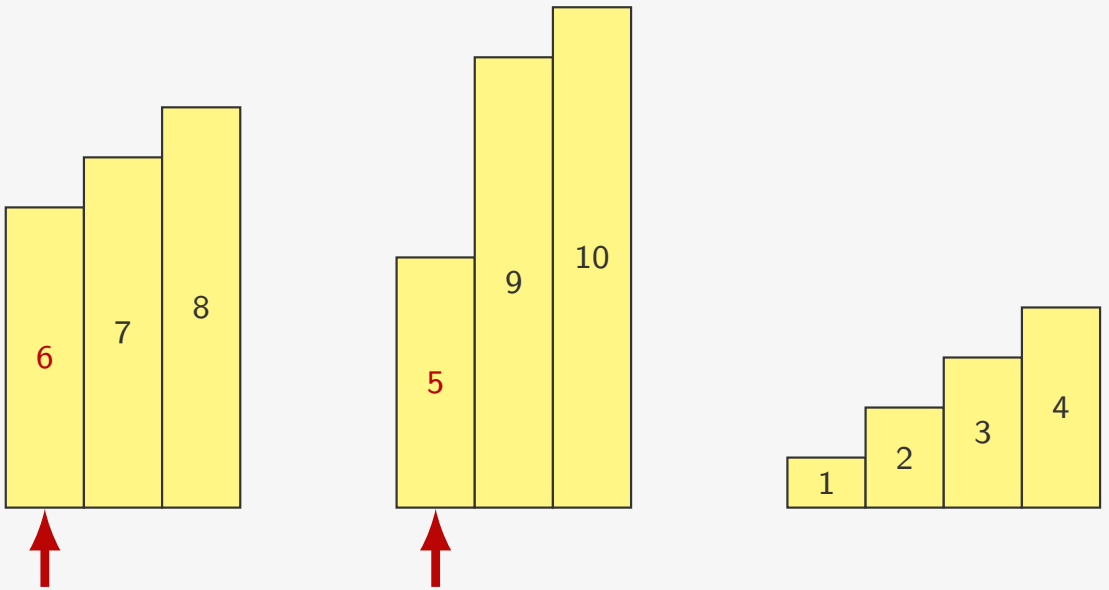
- Percorremos as duas sub-listas
- Pegamos o mínimo e inserimos em uma lista auxiliar

Intercalando



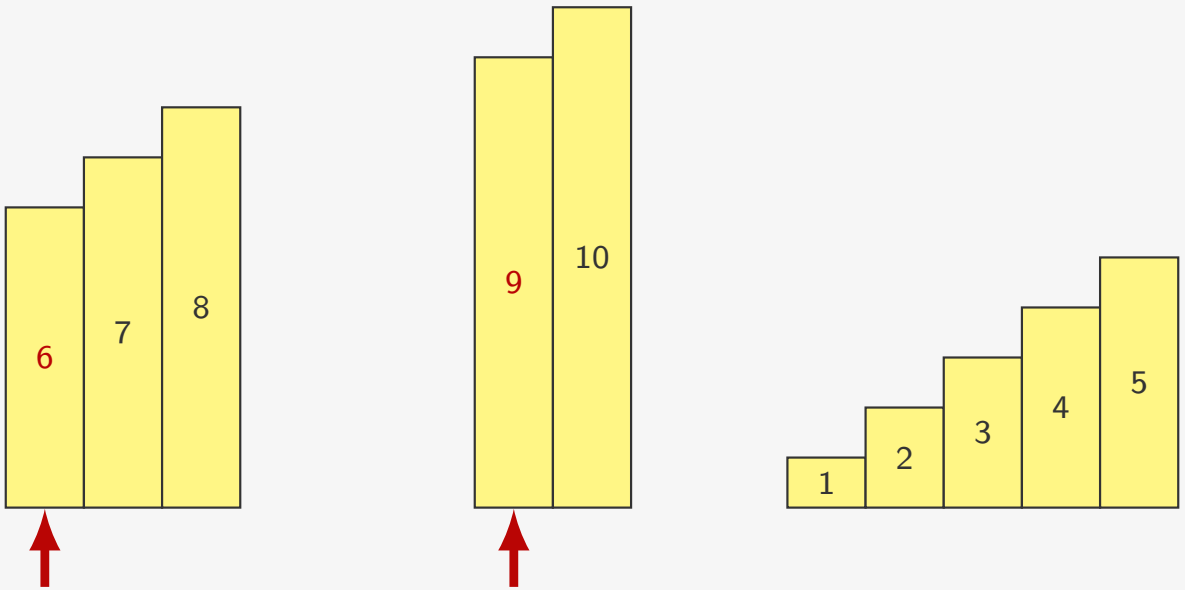
- Percorremos as duas sub-listas
- Pegamos o mínimo e inserimos em uma lista auxiliar

Intercalando



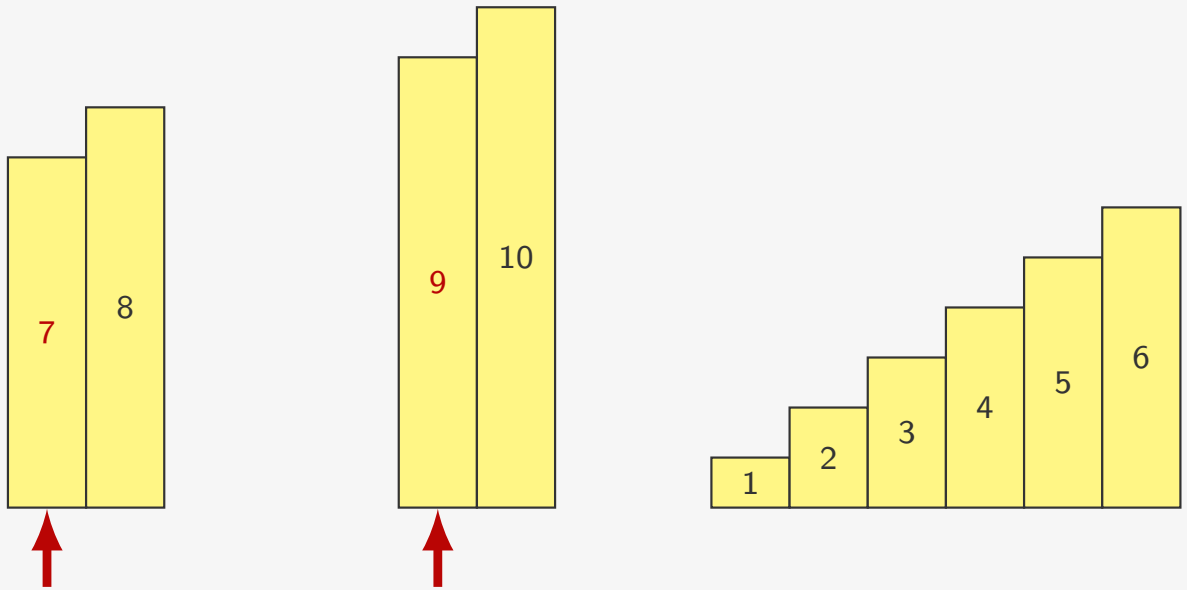
- Percorremos as duas sub-listas
- Pegamos o mínimo e inserimos em uma lista auxiliar

Intercalando



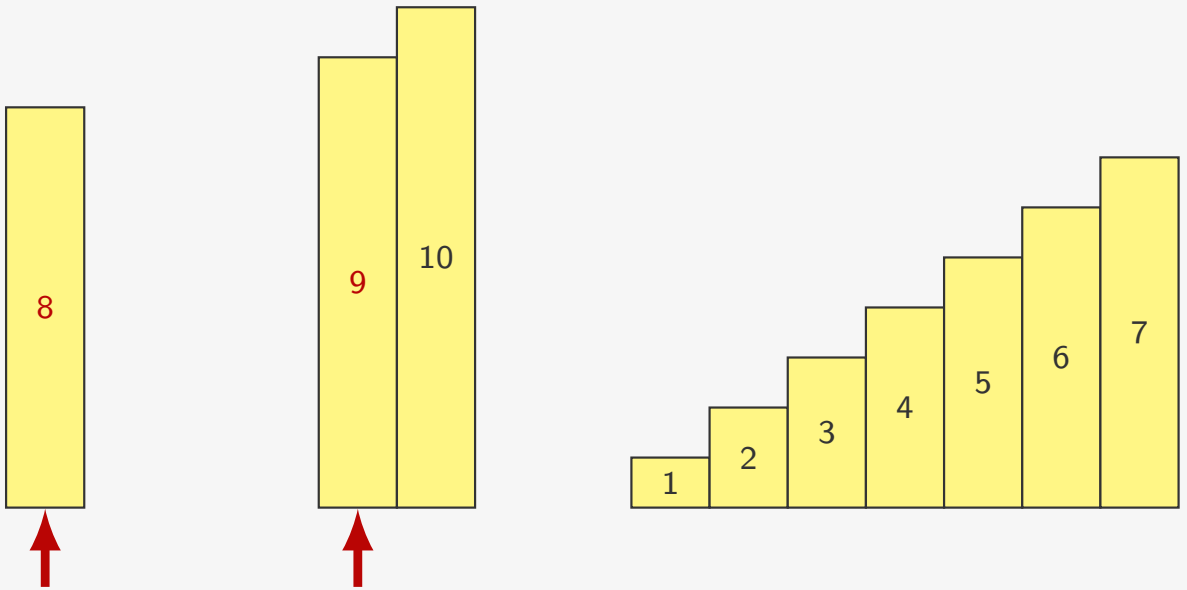
- Percorremos as duas sub-listas
- Pegamos o mínimo e inserimos em uma lista auxiliar

Intercalando



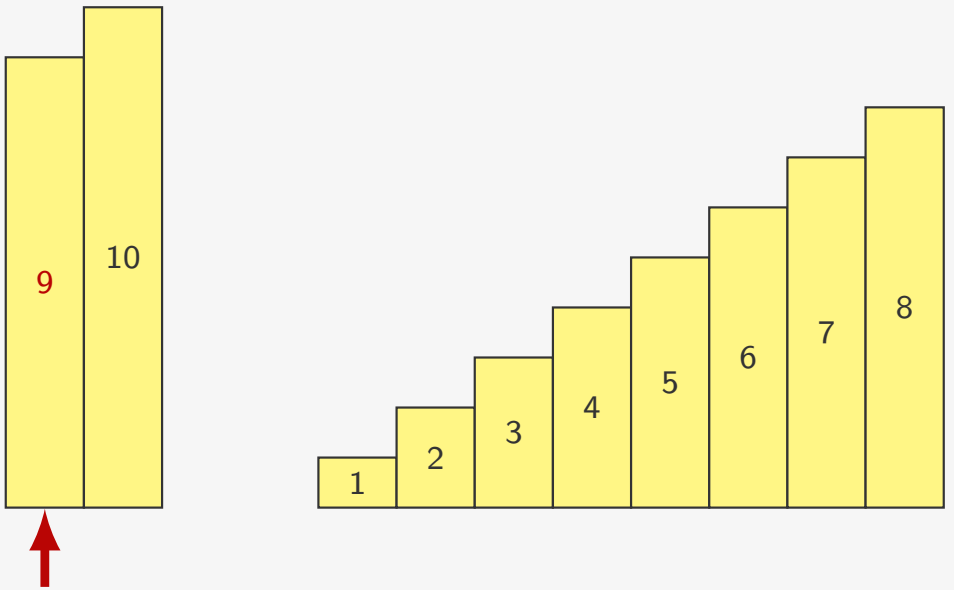
- Percorremos as duas sub-listas
- Pegamos o mínimo e inserimos em uma lista auxiliar

Intercalando



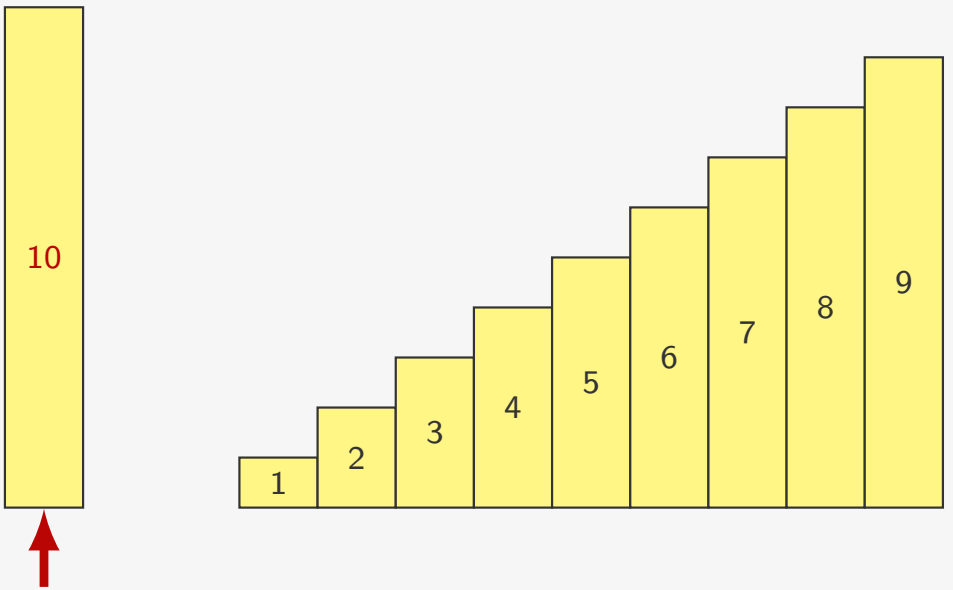
- Percorremos as duas sub-listas
- Pegamos o mínimo e inserimos em uma lista auxiliar

Intercalando



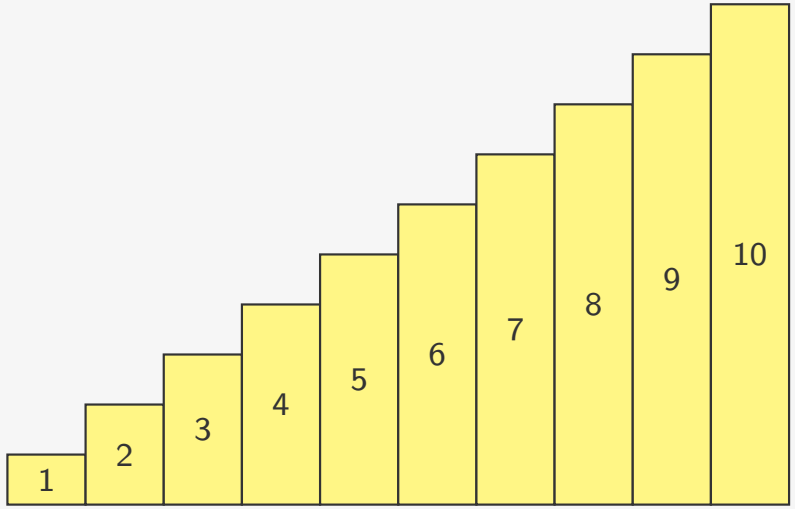
- Percorremos as duas sub-listas
- Pegamos o mínimo e inserimos em uma lista auxiliar
- Depois copiamos o restante

Intercalando



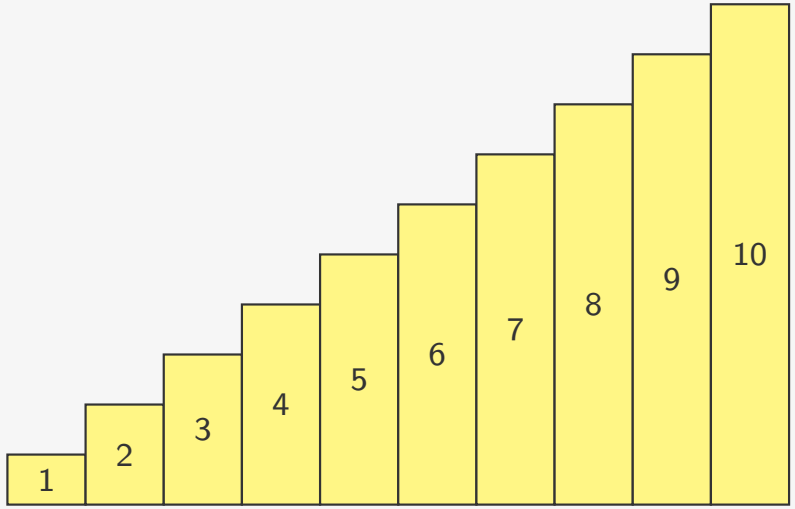
- Percorremos as duas sub-listas
- Pegamos o mínimo e inserimos em uma lista auxiliar
- Depois copiamos o restante

Intercalando



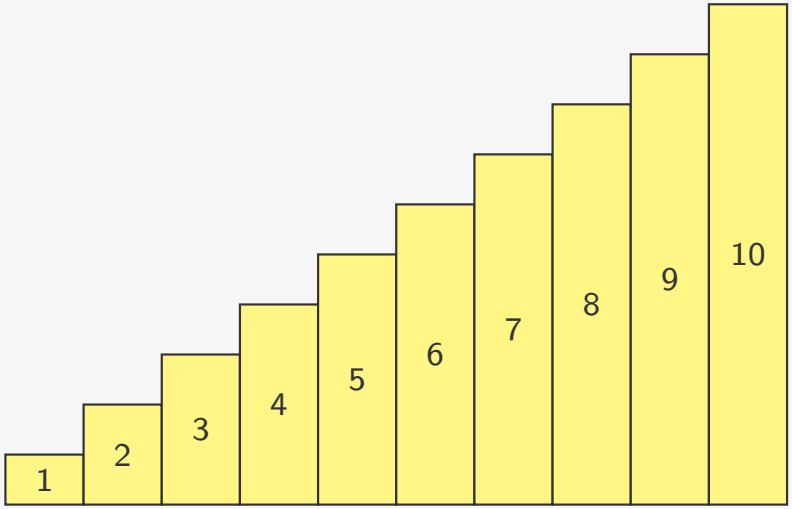
- Percorremos as duas sub-listas
- Pegamos o mínimo e inserimos em uma lista auxiliar
- Depois copiamos o restante

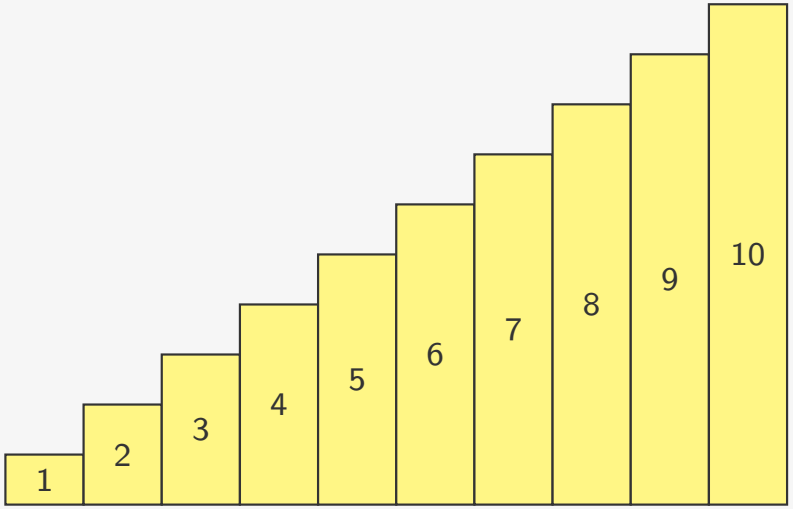
Intercalando



- 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

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

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:  # 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:

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!

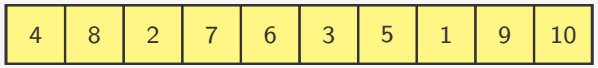
Ordenação por intercalação (*MergeSort*)

Ordenação:

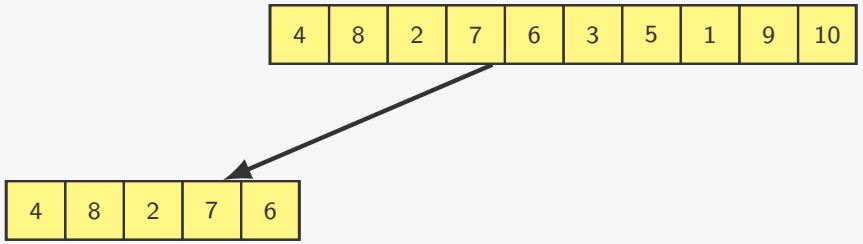
- Recebemos uma faixa da lista `l`:
  - A faixa começa na posição `e`
  - A faixa termina na posição `d`
- Dividimos a faixa em duas
- O caso base é uma faixa de tamanho 0 ou 1
  - Já está ordenada!

```
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)
```

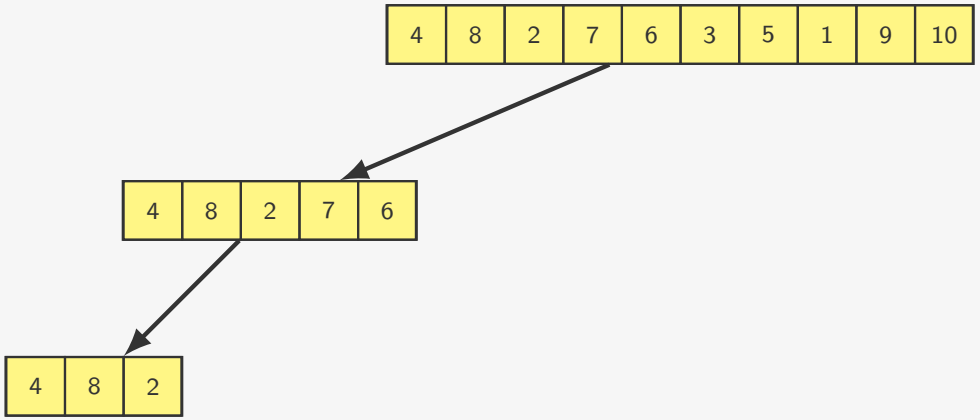
Simulação



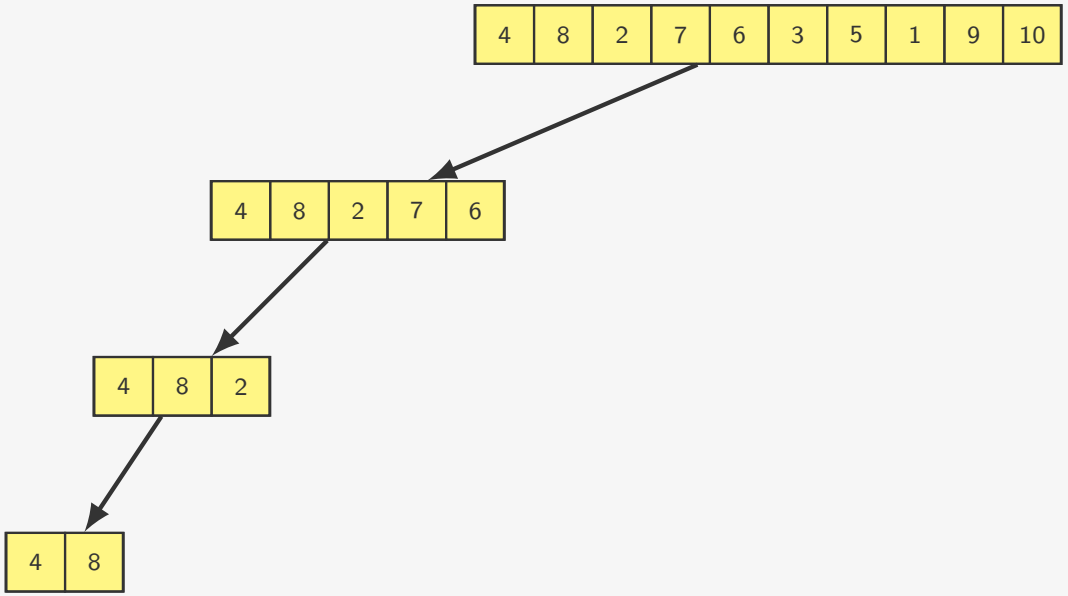
Simulação



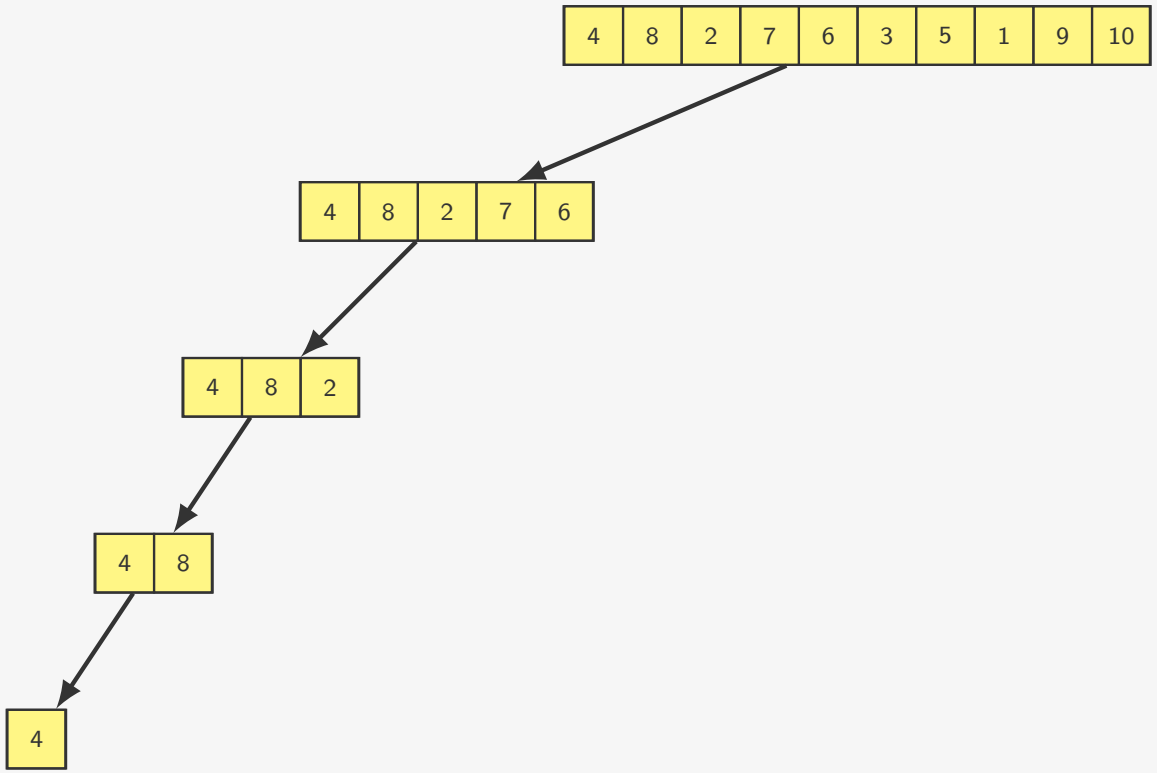
Simulação



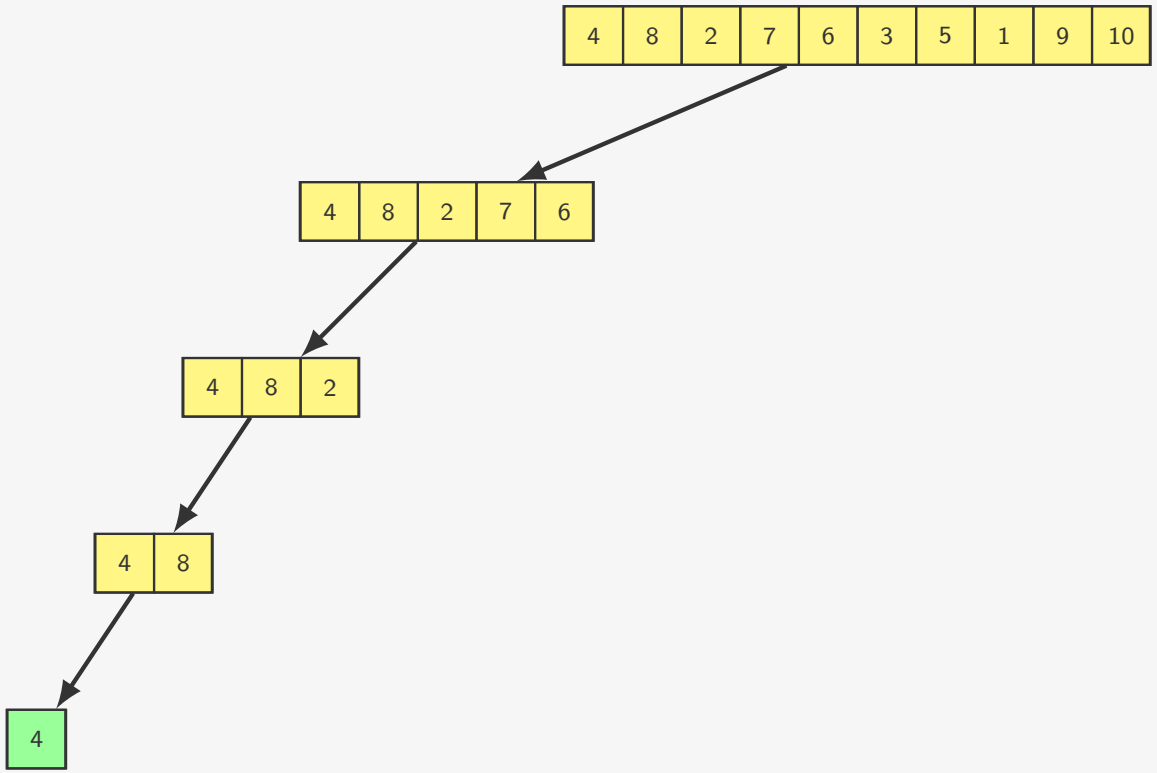
Simulação



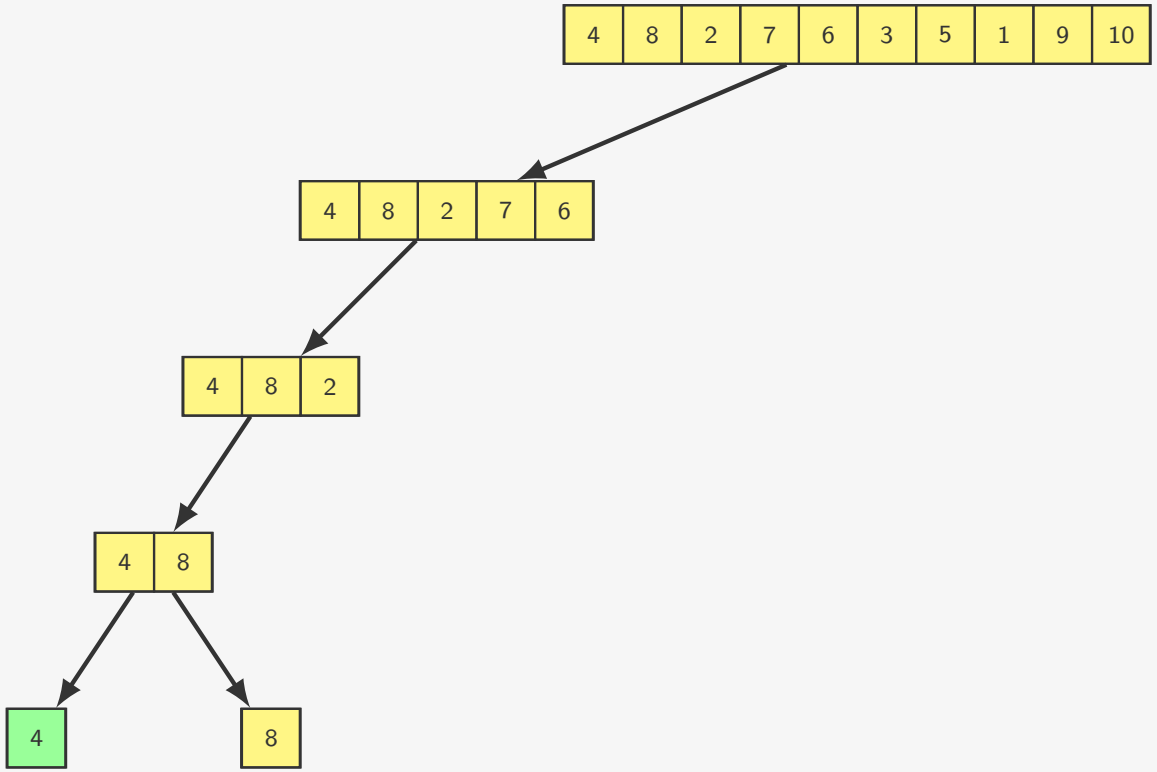
Simulação



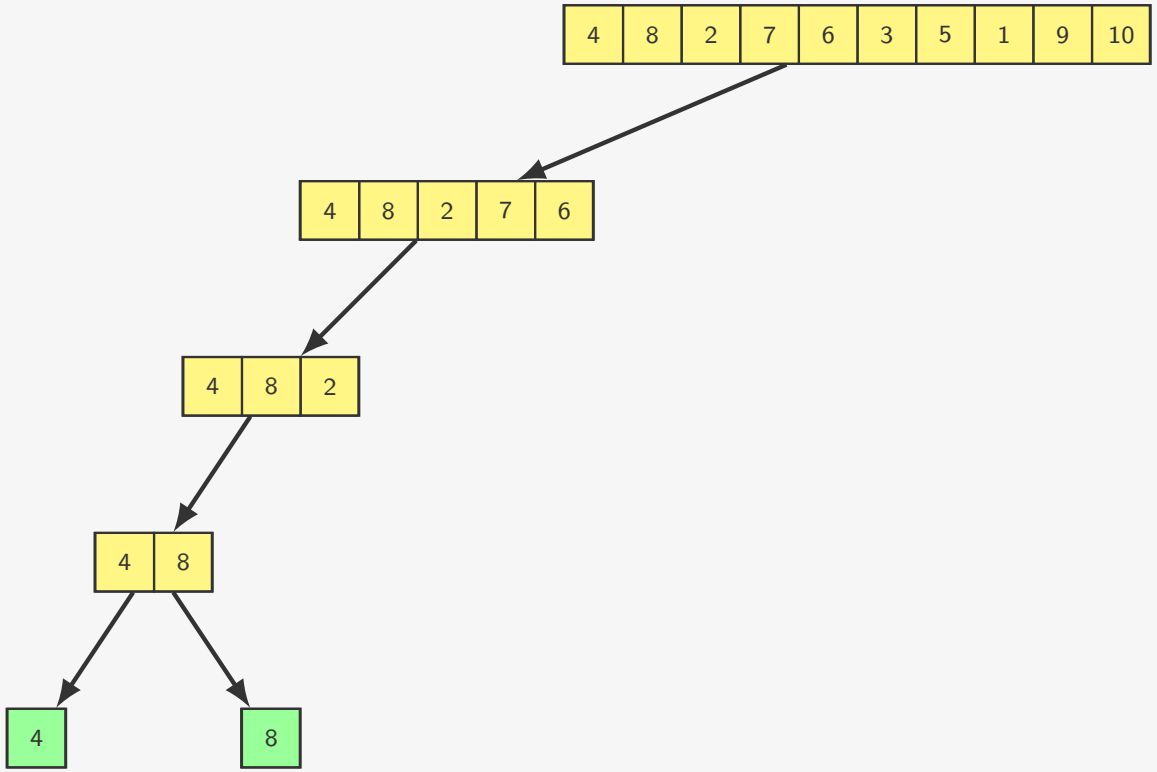
Simulação



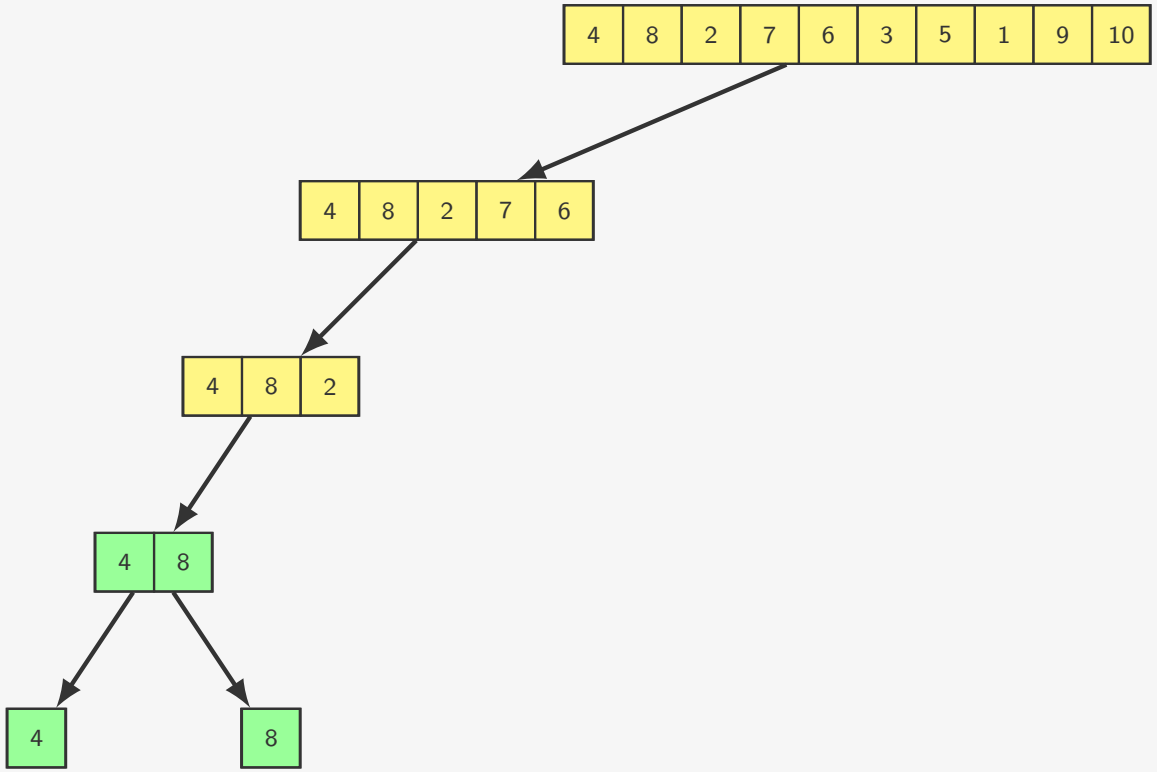
Simulação



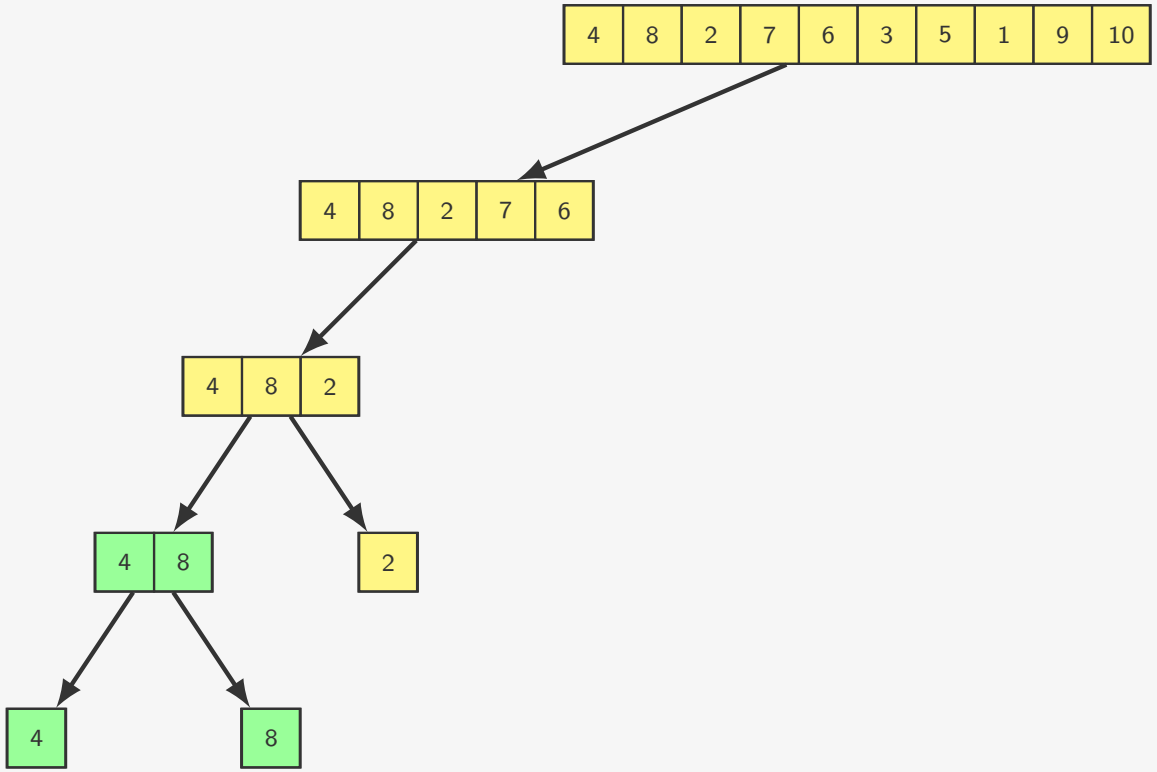
Simulação



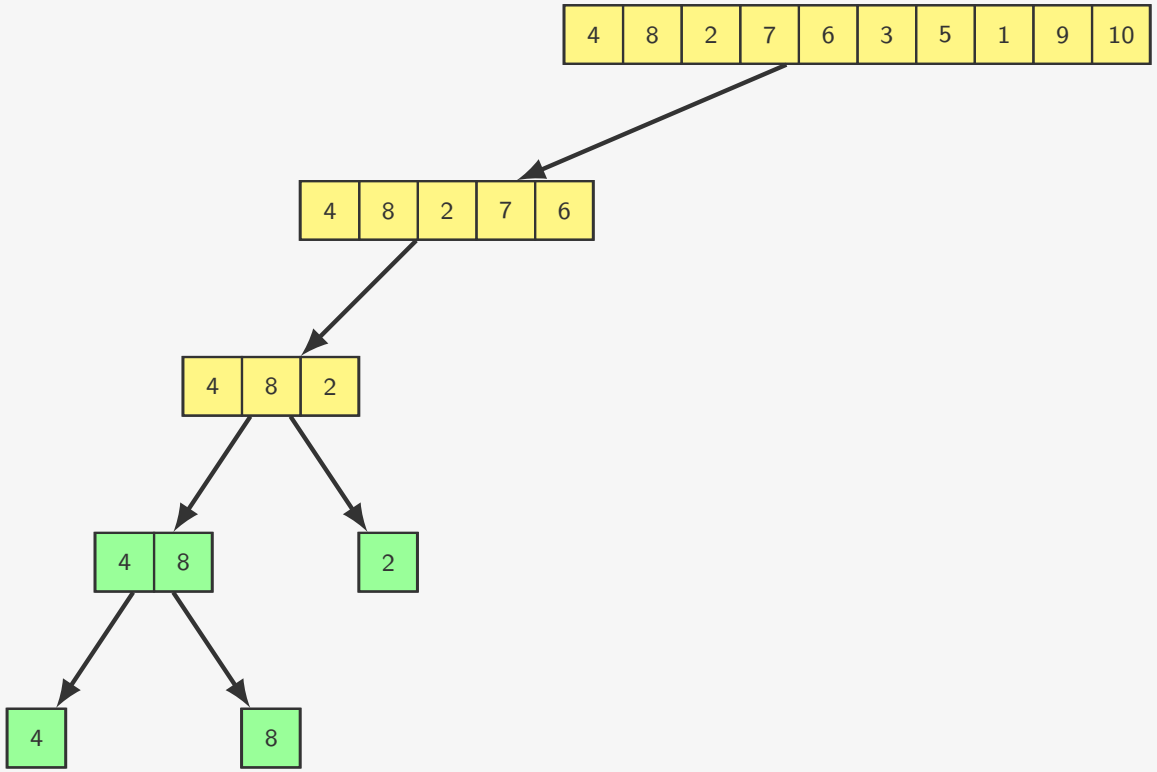
Simulação



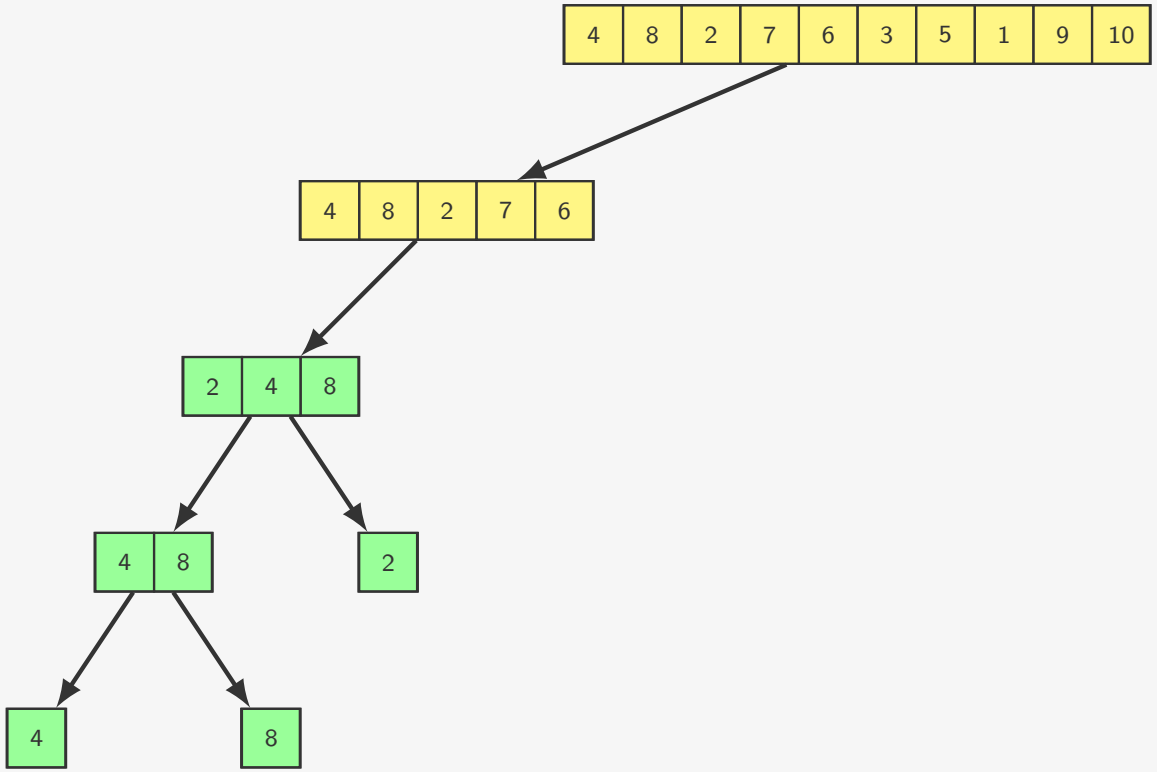
Simulação



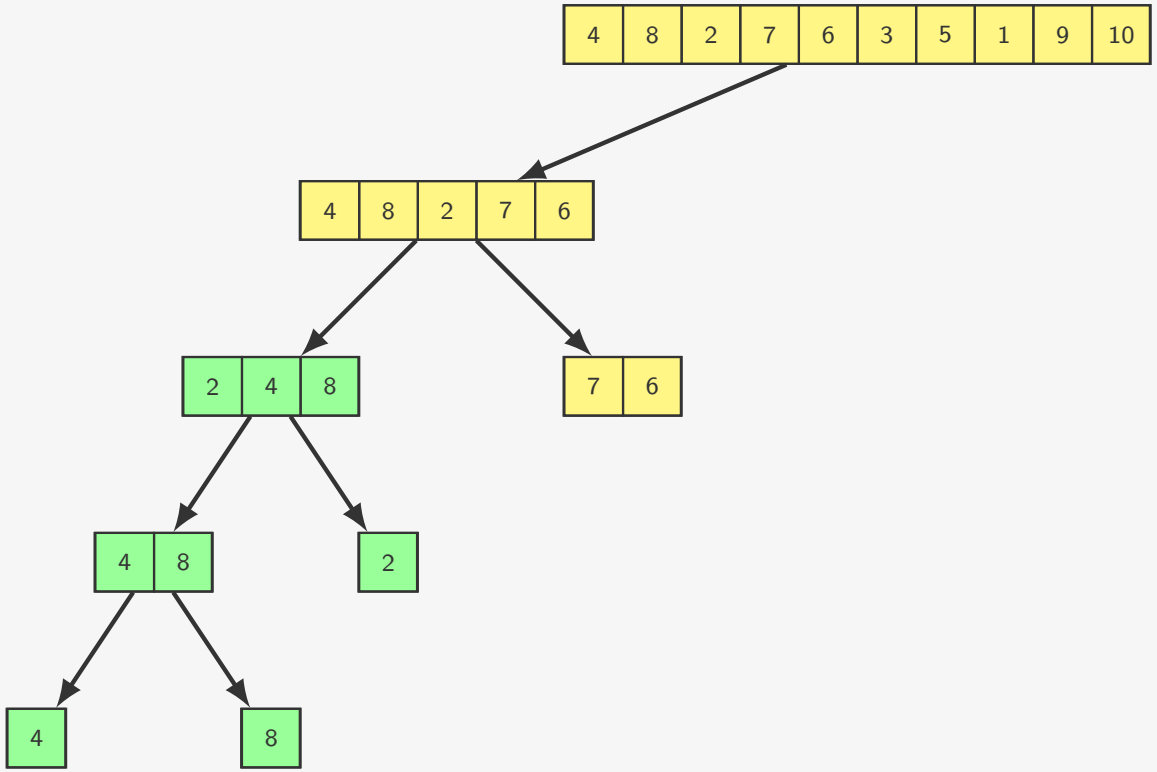
Simulação



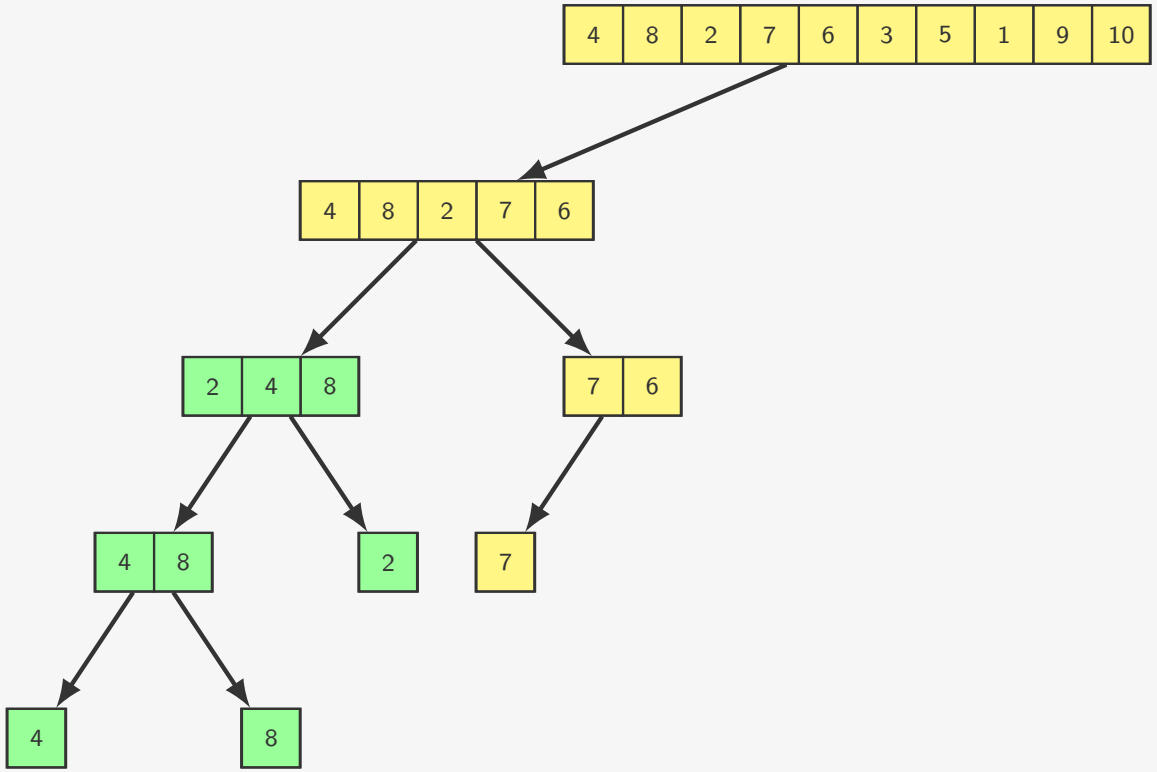
Simulação



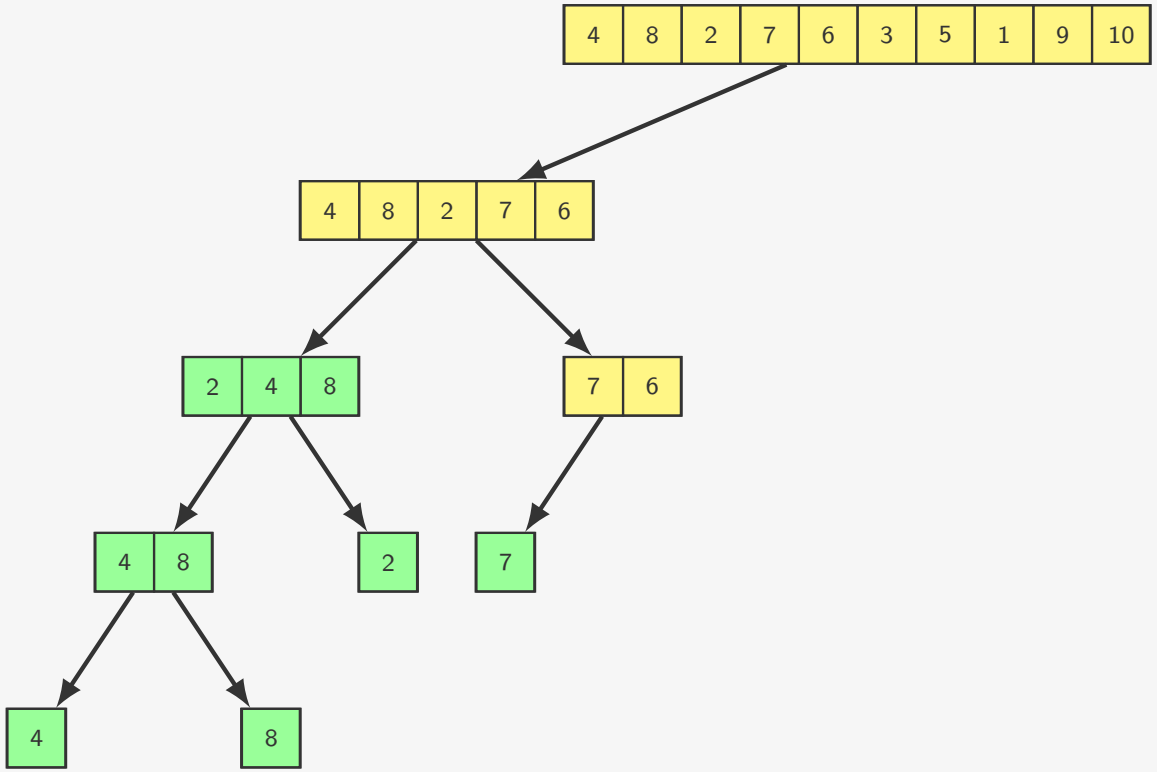
Simulação



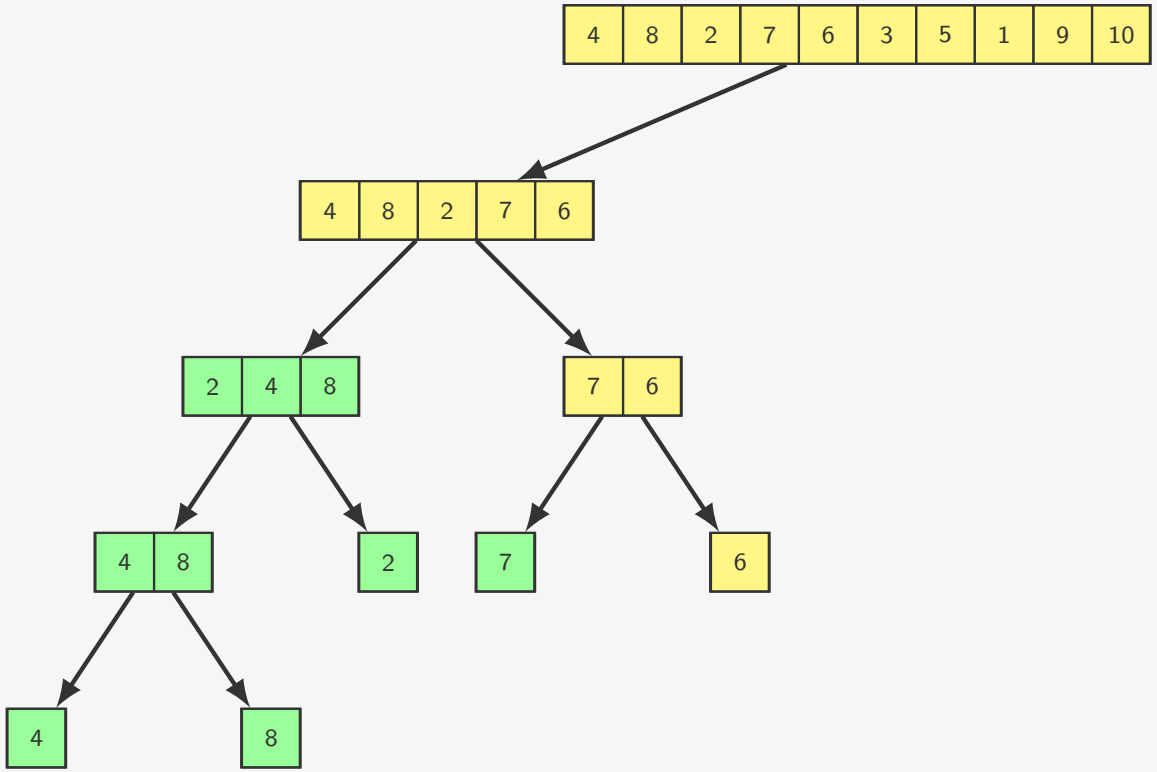
Simulação



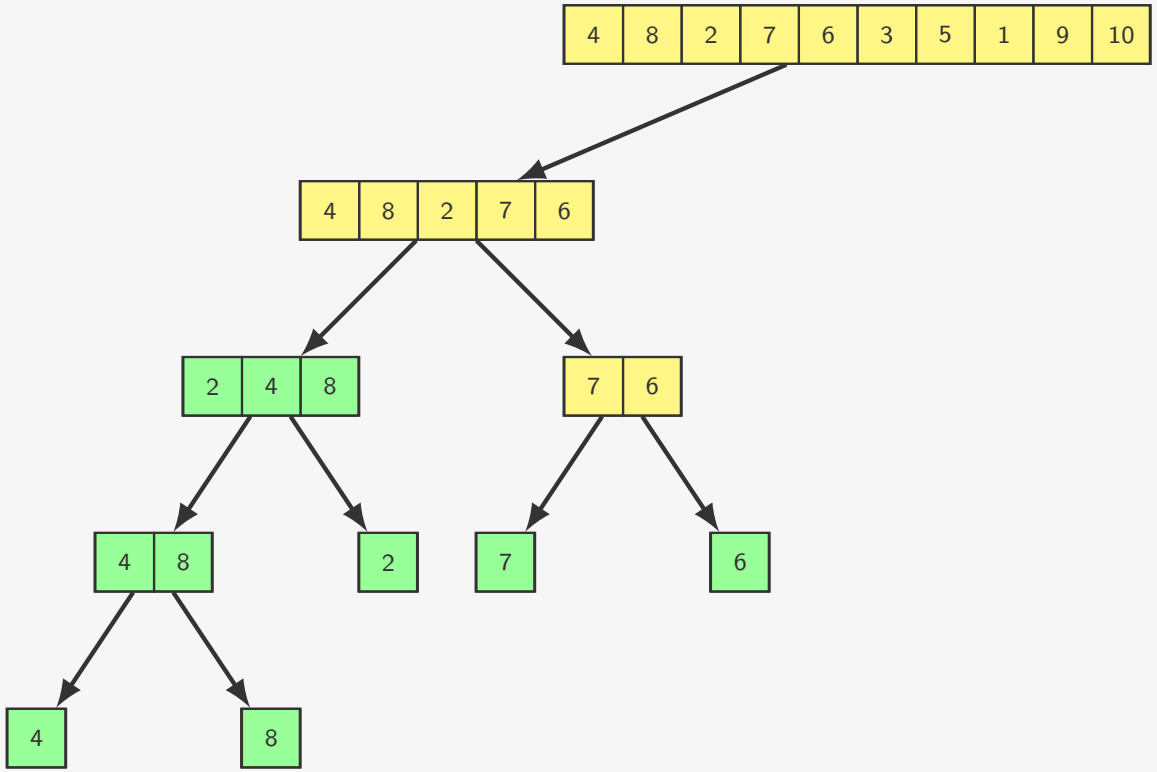
Simulação



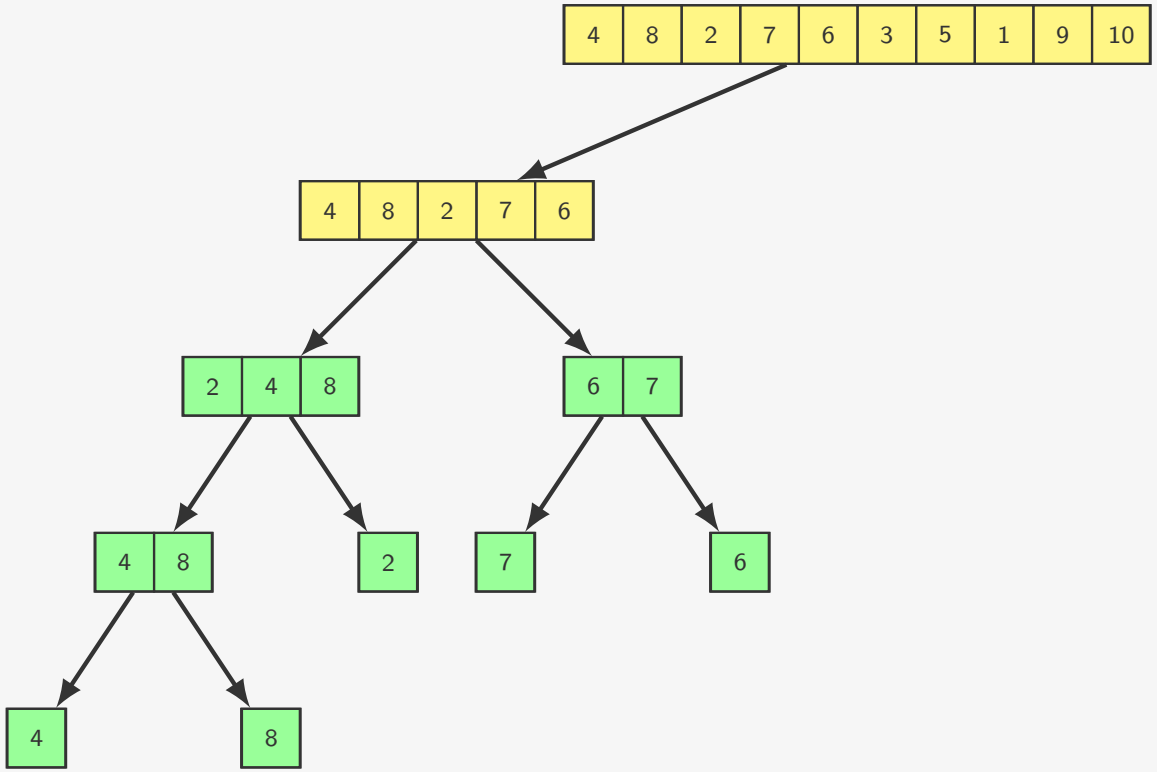
Simulação



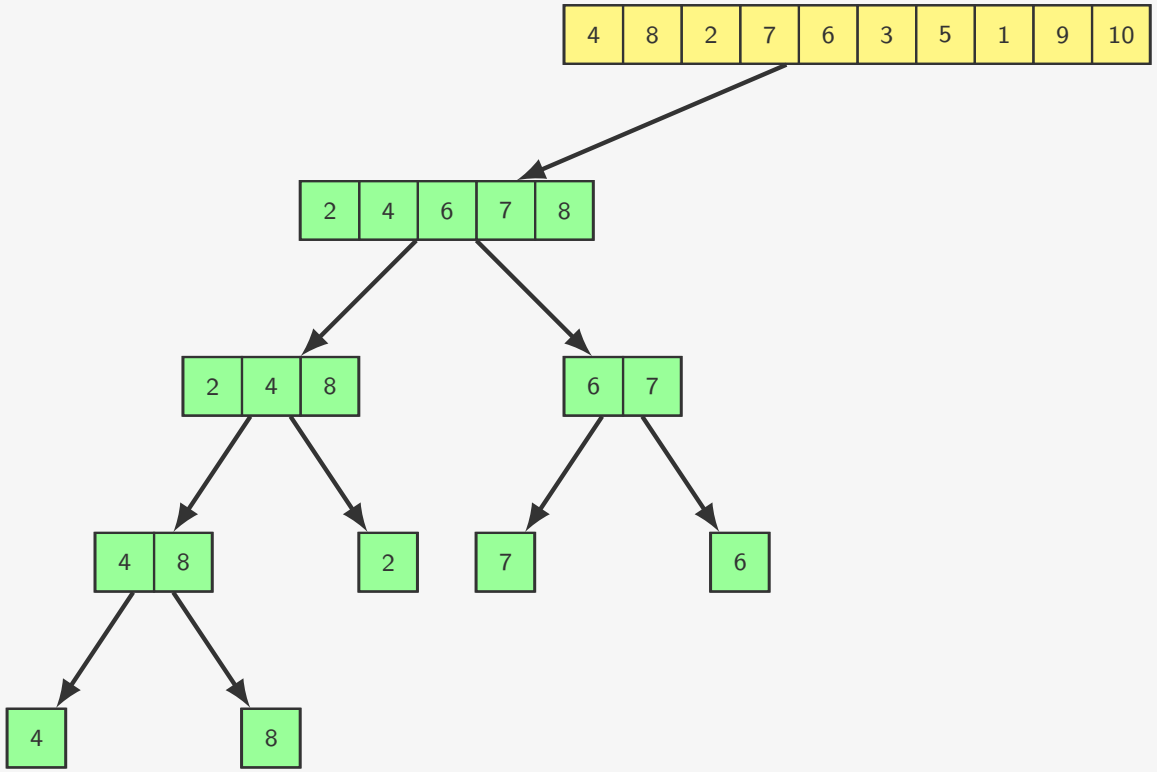
Simulação



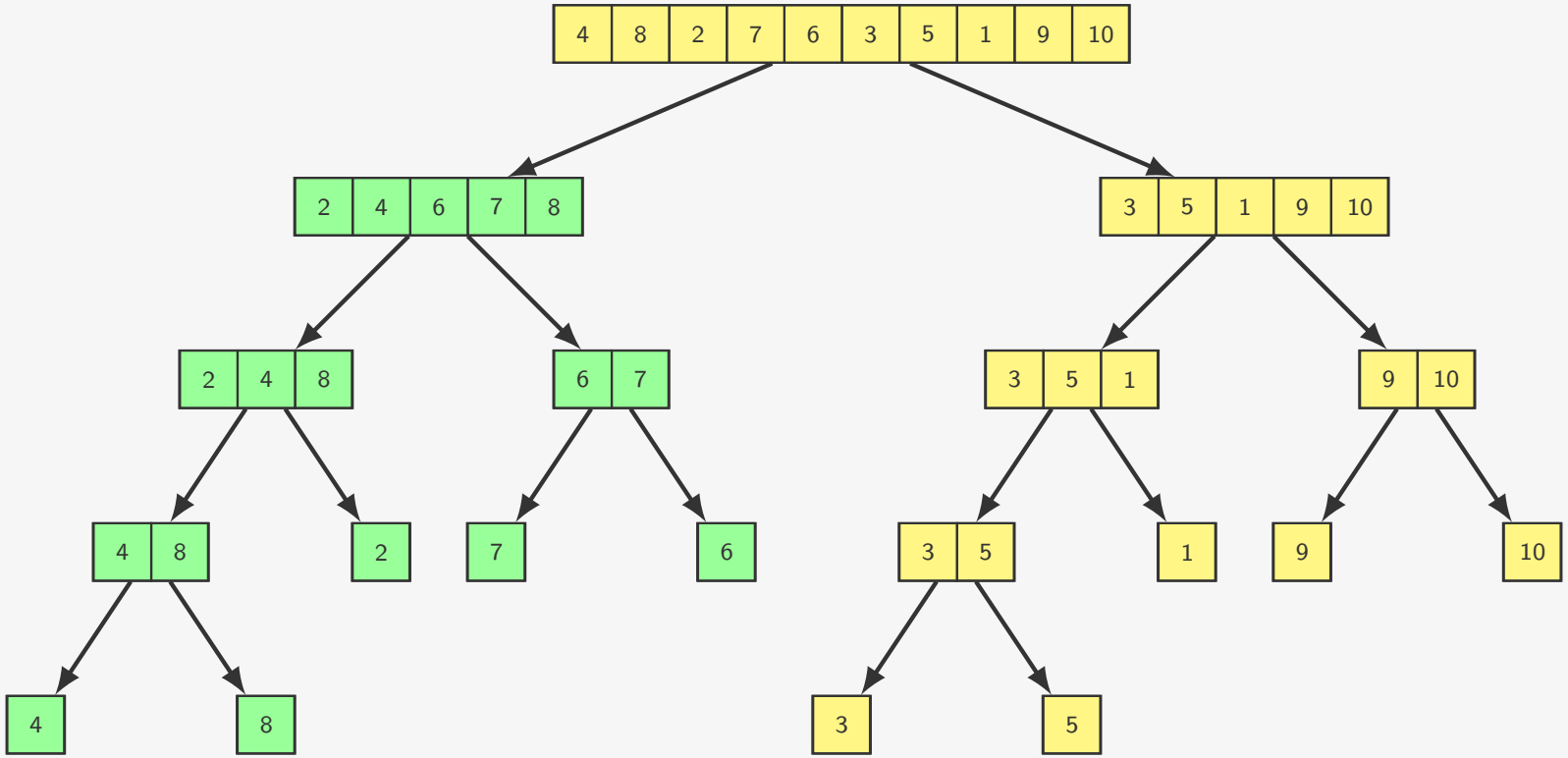
Simulação



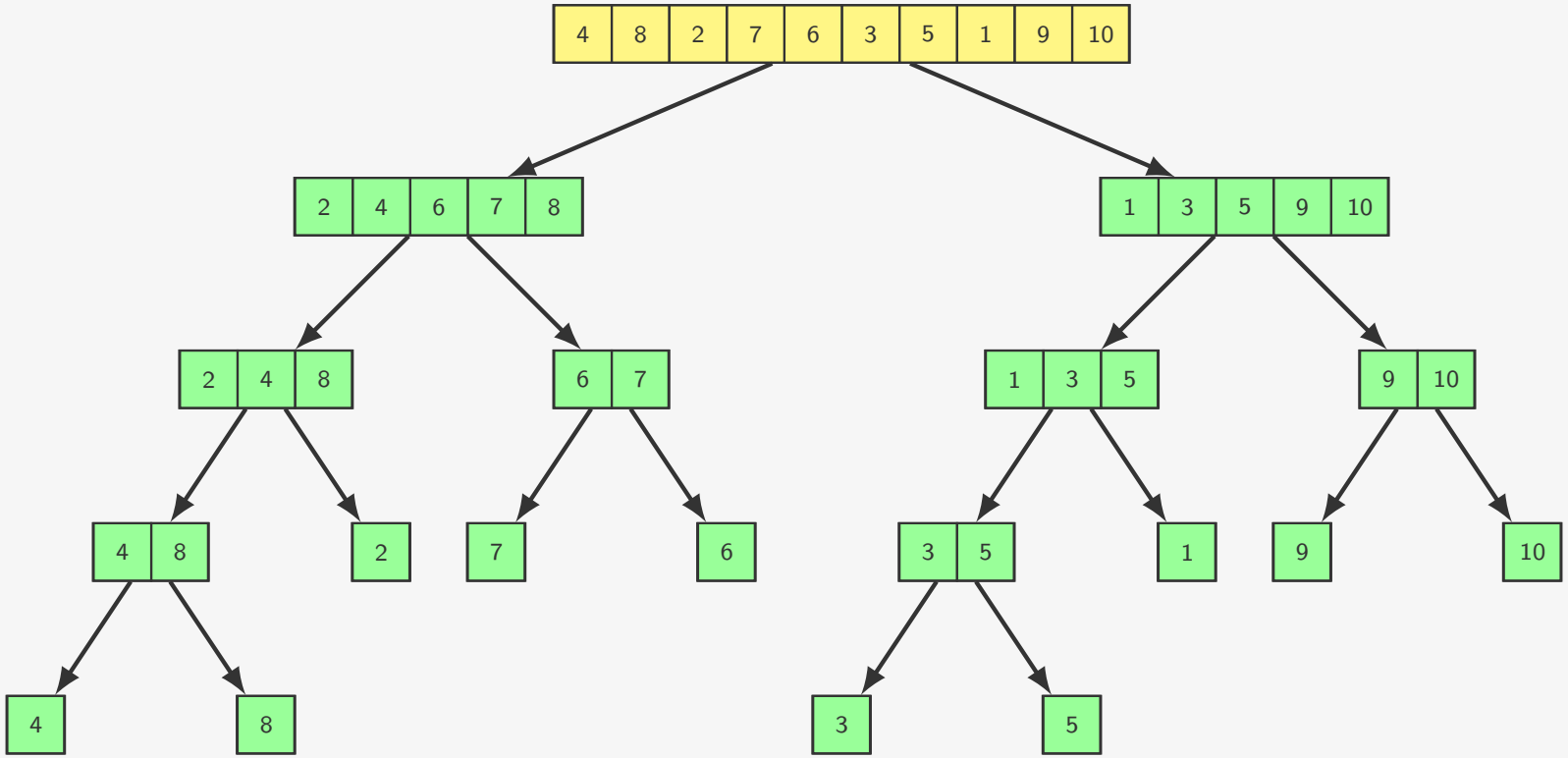
Simulação



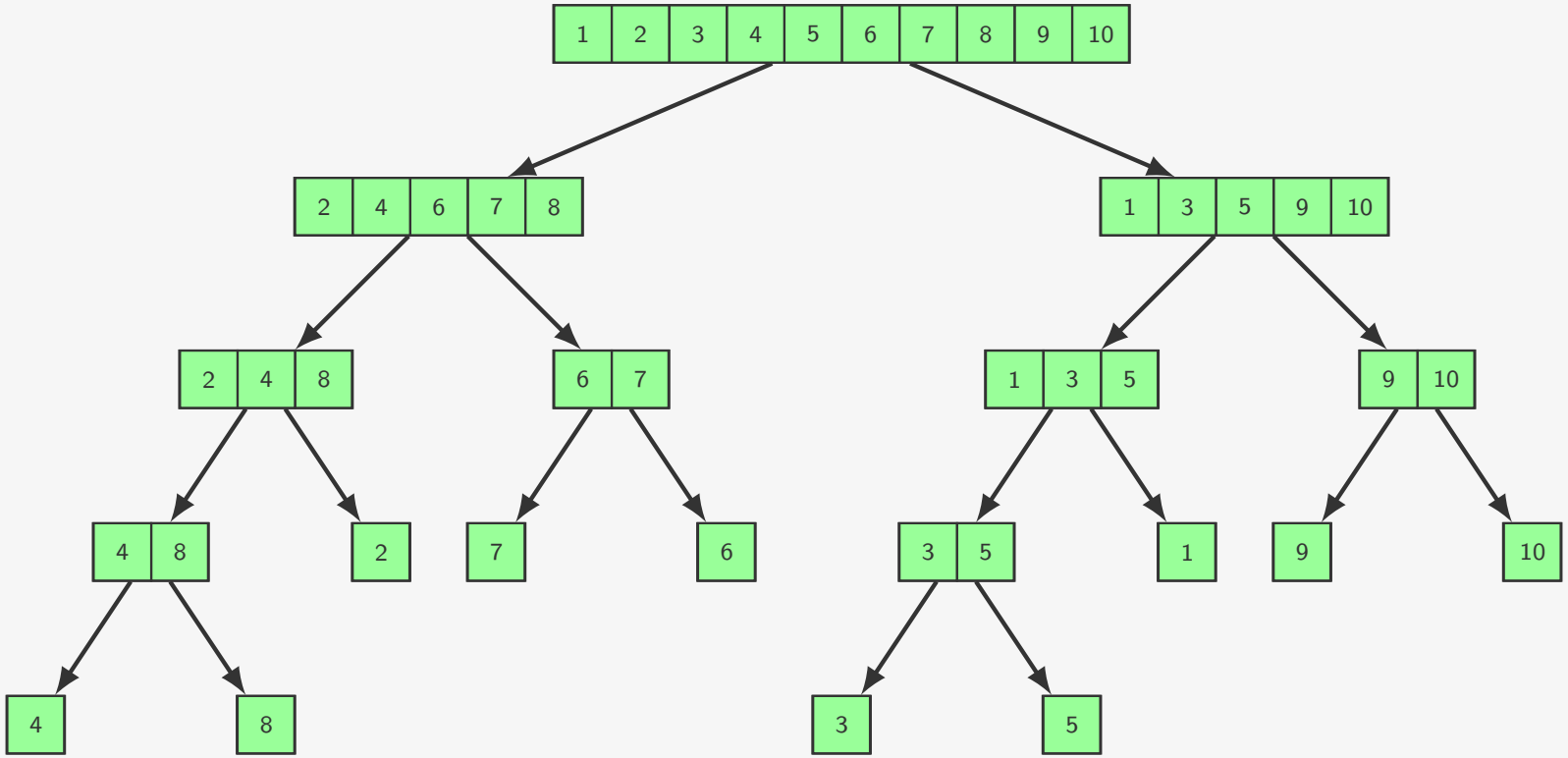
Simulação



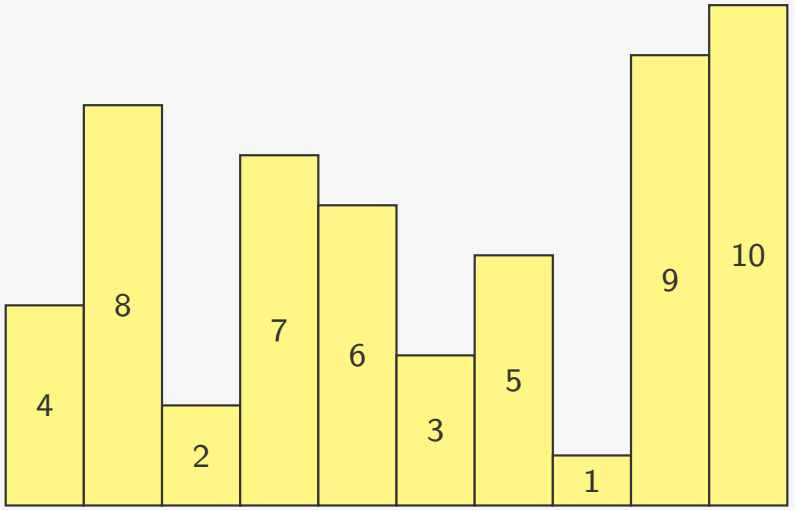
Simulação



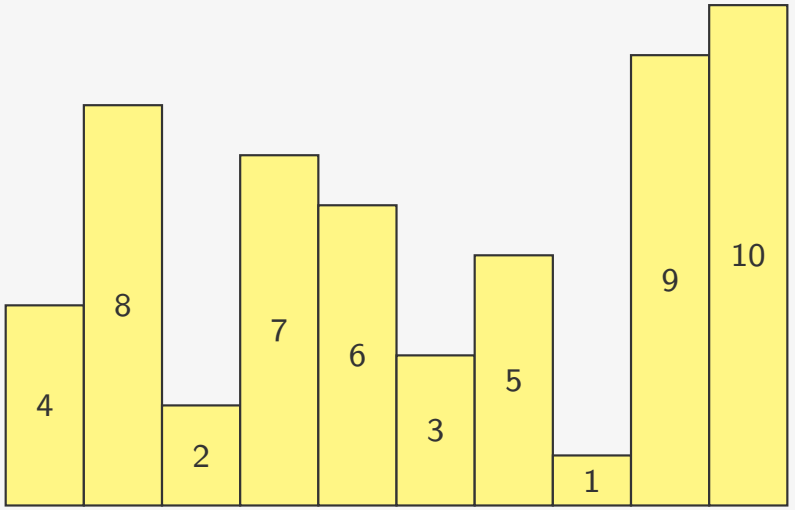
Simulação



Quicksort — Ideia

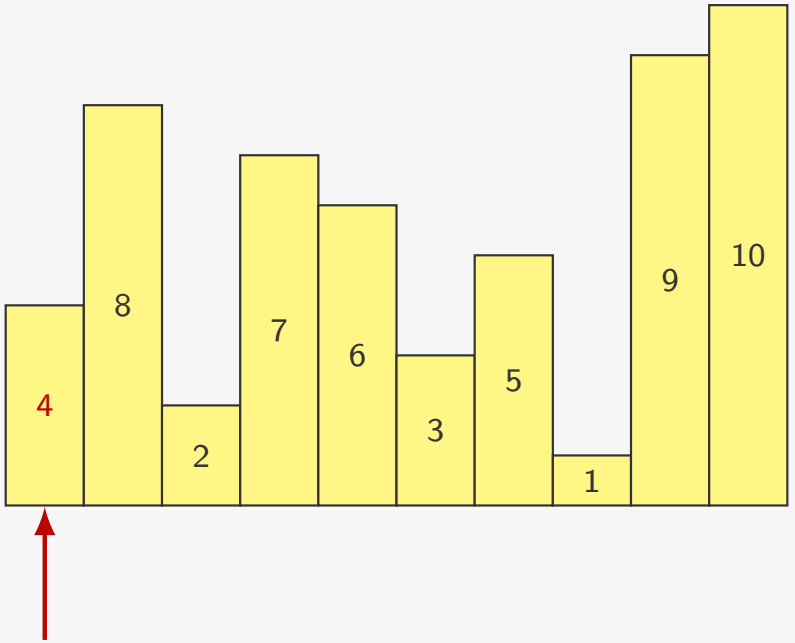


Quicksort — Ideia



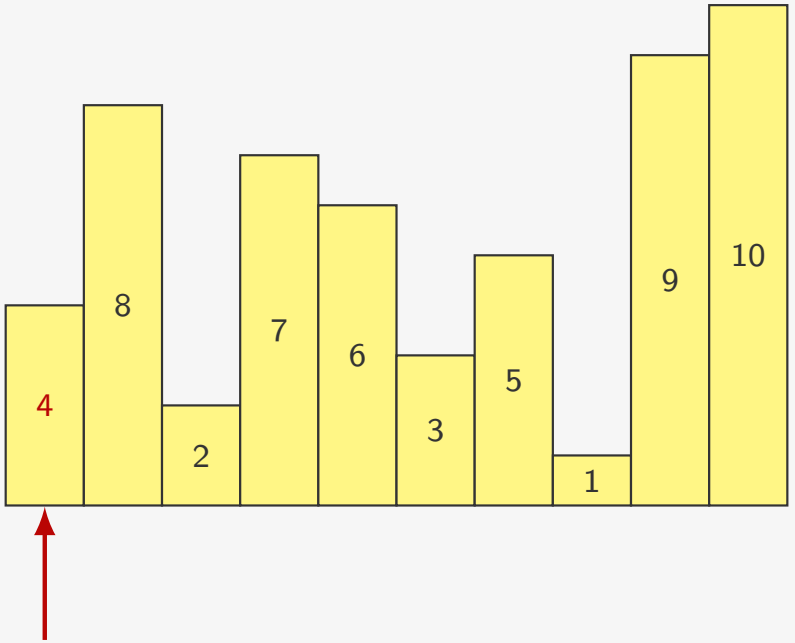
- Escolhemos um pivô (ex: 4)

Quicksort — Ideia



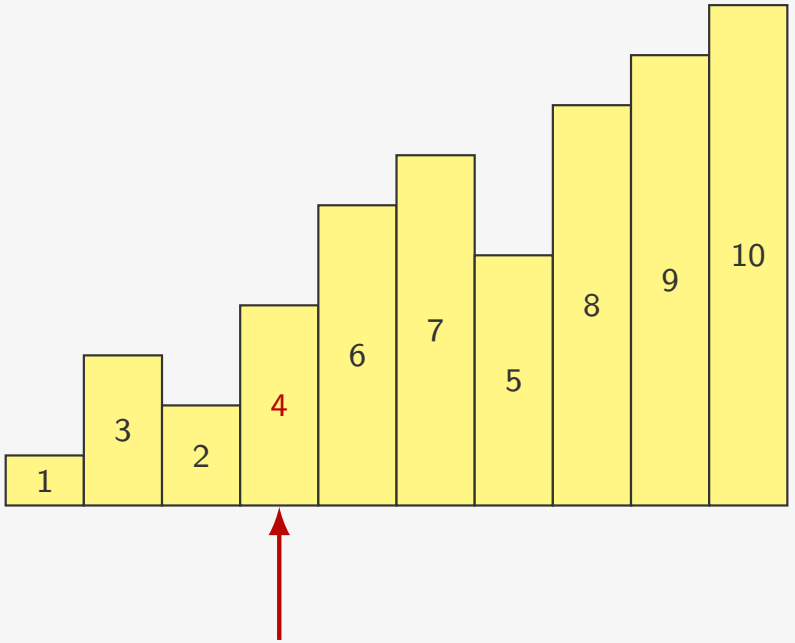
- Escolhemos um pivô (ex: 4)

Quicksort — Ideia



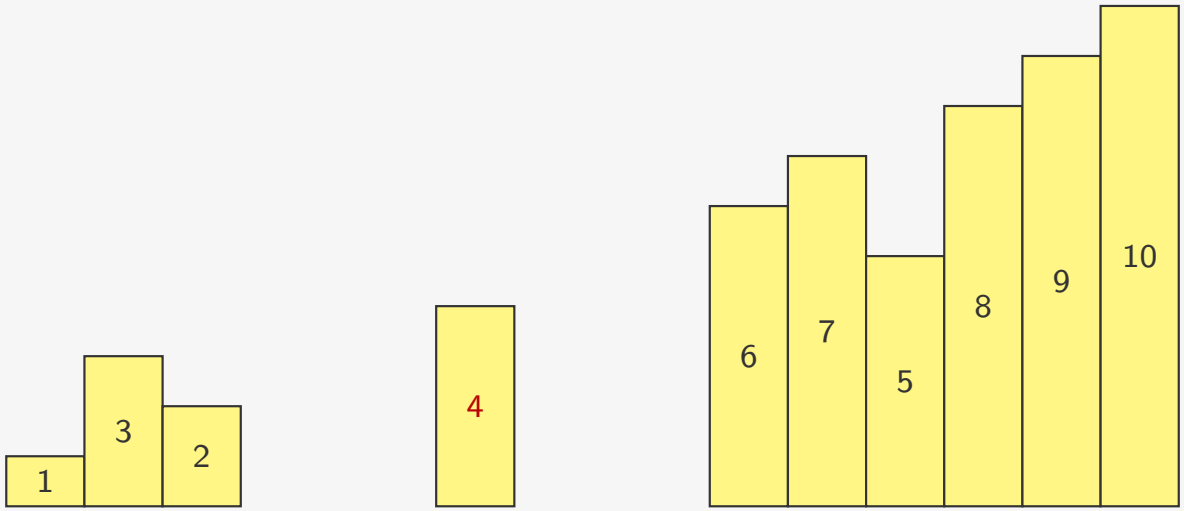
- Escolhemos um pivô (ex: 4)
- Colocamos
  - os elementos menores que o pivô na esquerda
  - os elementos maiores que o pivô na direita

Quicksort — Ideia



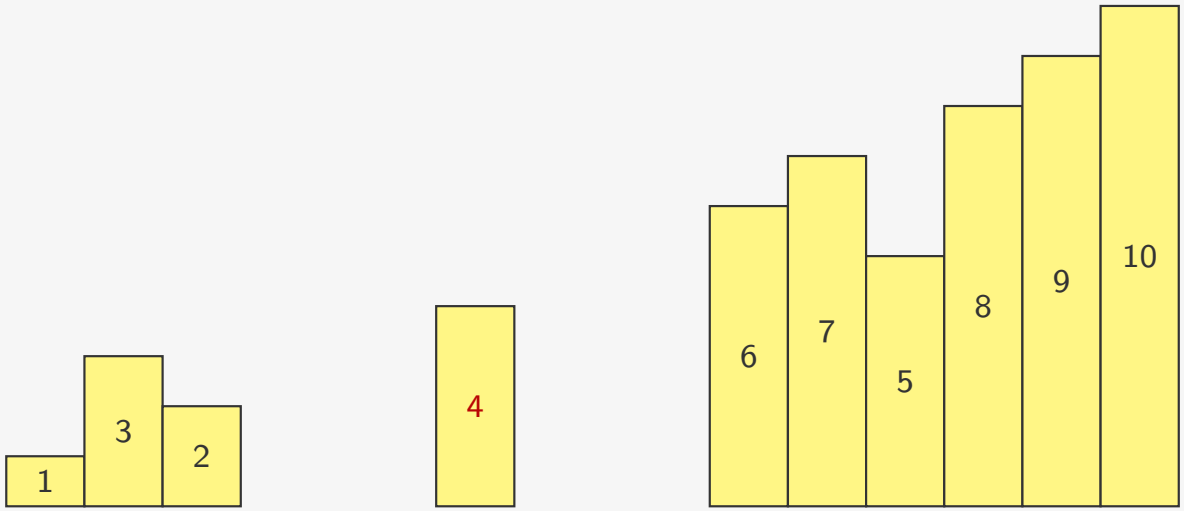
- Escolhemos um pivô (ex: 4)
- Colocamos
  - os elementos menores que o pivô na esquerda
  - os elementos maiores que o pivô na direita

Quicksort — Ideia



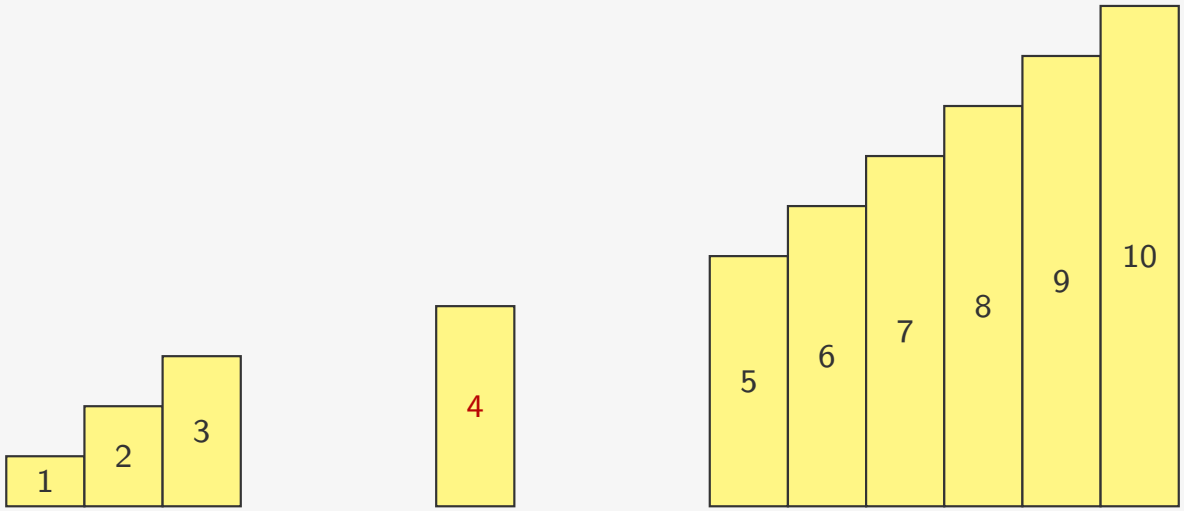
- Escolhemos um pivô (ex: 4)
- Colocamos
  - os elementos menores que o pivô na esquerda
  - os elementos maiores que o pivô na direita
- O pivô está na posição correta

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
- e 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 `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?

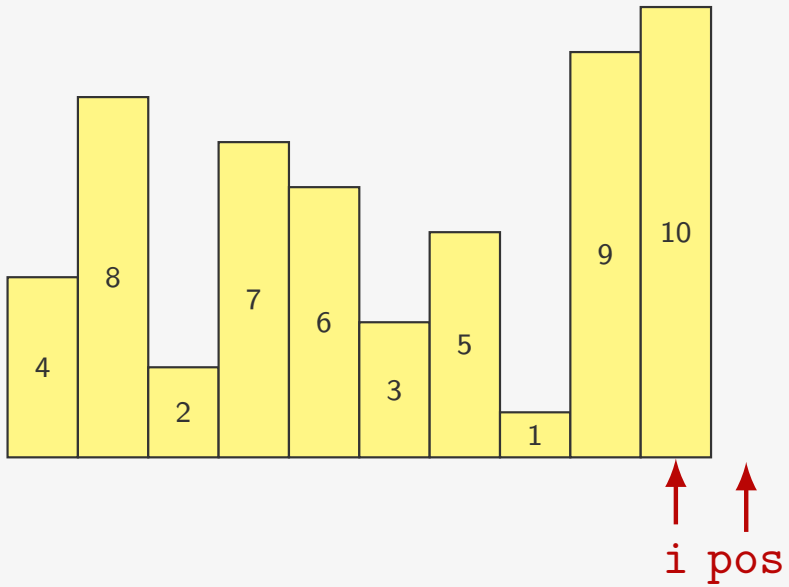
- Andamos da direita para a esquerda com um índice `i`
- De `i` até `pos - 1` ficam os menores do que o pivô
- De `pos` até `d` ficam os maiores ou iguais ao pivô
- Se o elemento em `i` for maior ou igual ao pivô
  - diminuímos `pos` e realizamos uma troca de `i` com `pos`
- No final, o pivô está em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

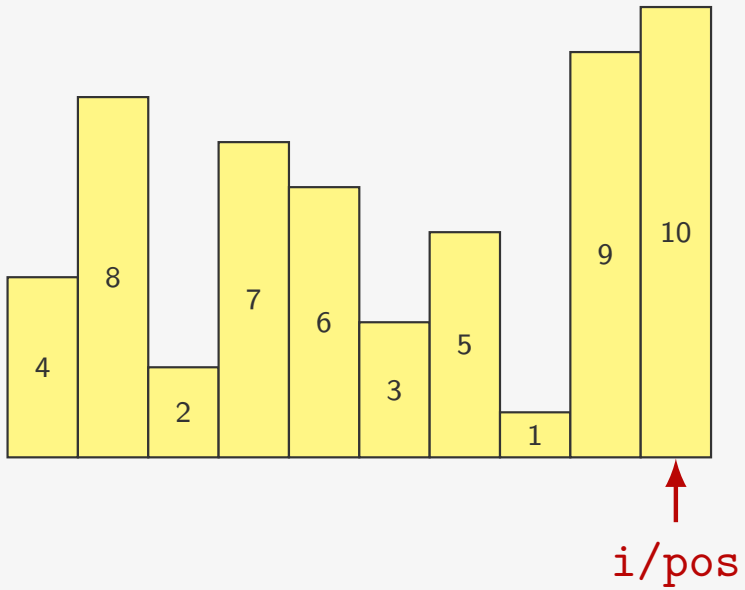
```
1 def partition(l, e, d):
2     pivo = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivo:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
```



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`

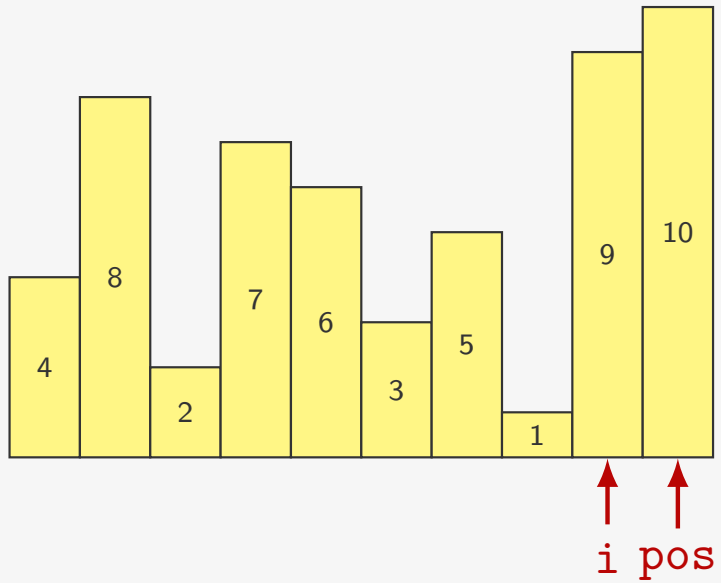
```
1 def partition(l, e, d):
2     pivo = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivo:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
```



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`

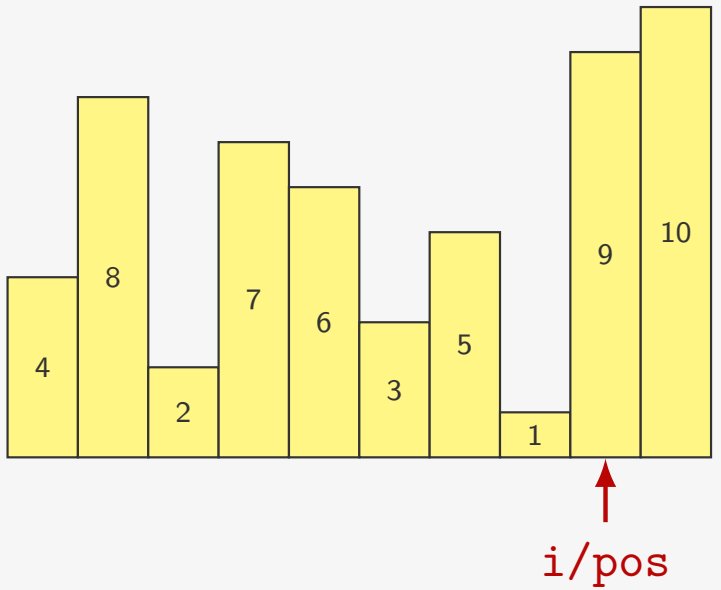
```
1 def partition(l, e, d):
2     pivo = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivo:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
```



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`

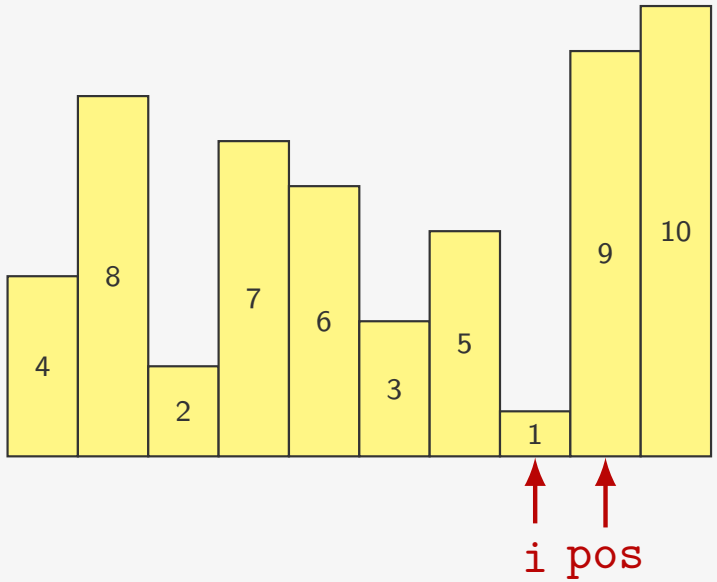
```
1 def partition(l, e, d):
2     pivo = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivo:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
```



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`

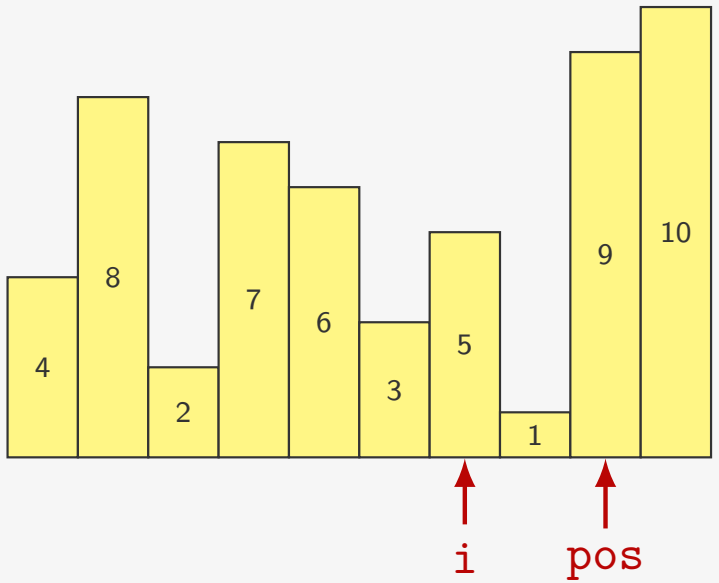
```
1 def partition(l, e, d):
2     pivo = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivo:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
```



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`

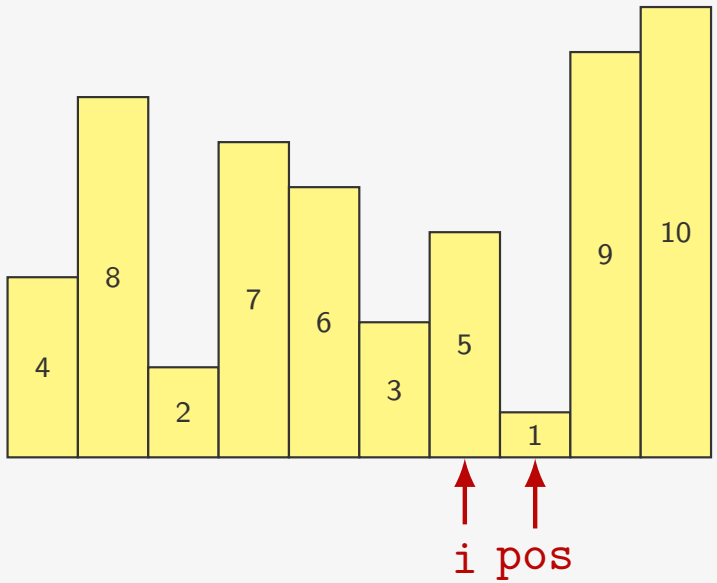
```
1 def partition(l, e, d):
2     pivo = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivo:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
```



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`

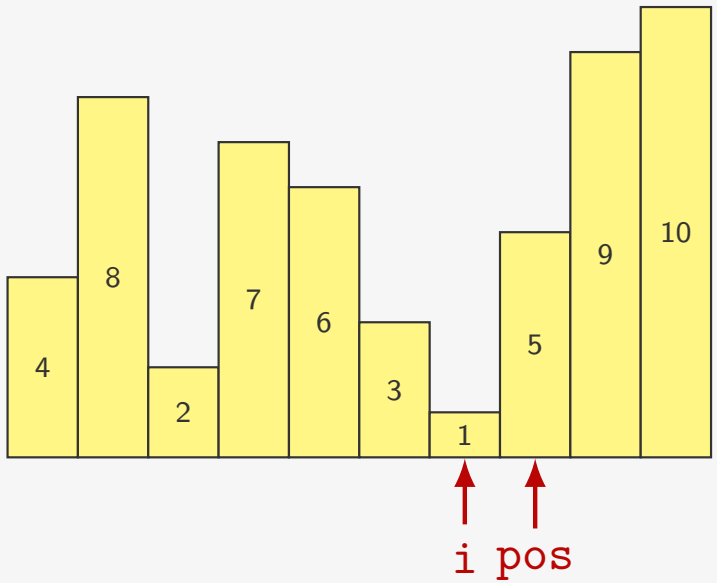
```
1 def partition(l, e, d):
2     pivo = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivo:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
```



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`

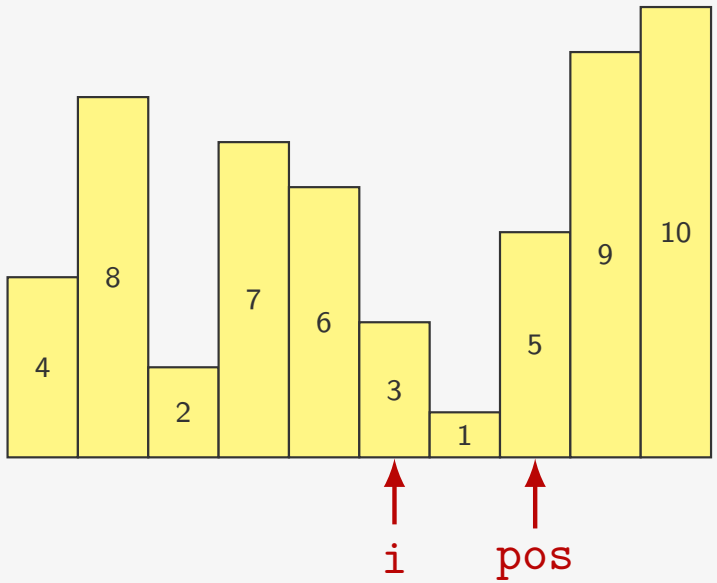
```
1 def partition(l, e, d):
2     pivo = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivo:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
```



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`

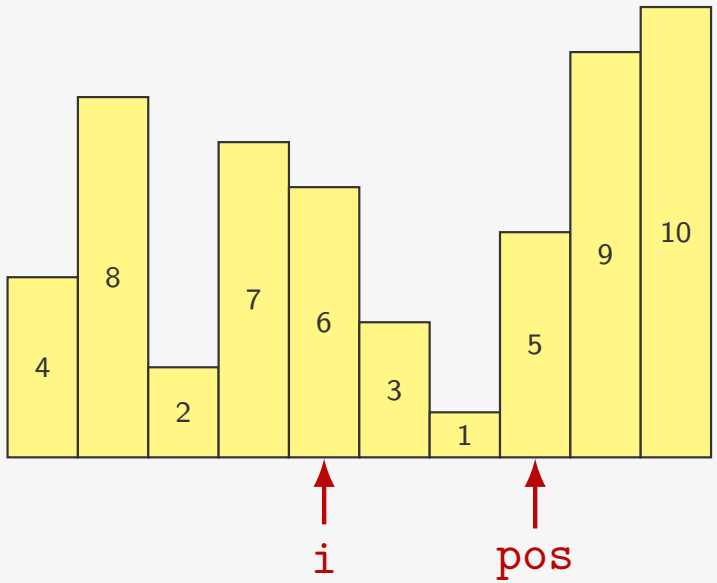
```
1 def partition(l, e, d):
2     pivo = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivo:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
```



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`

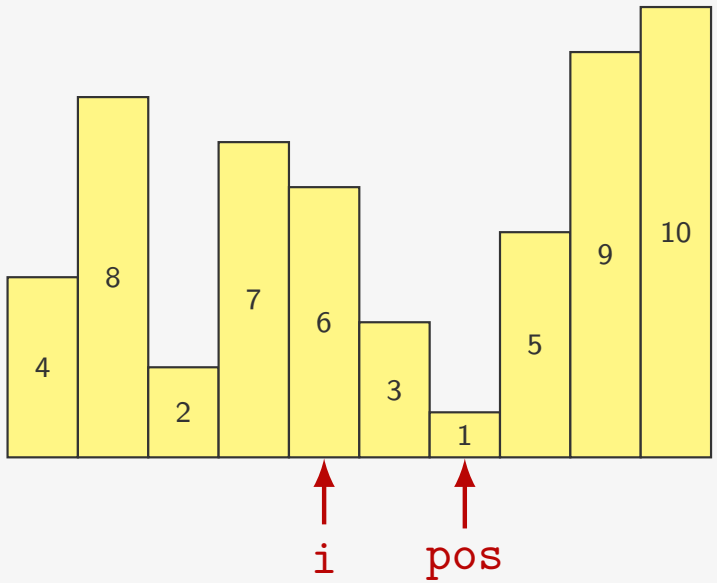
```
1 def partition(l, e, d):
2     pivo = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivo:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
```



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`

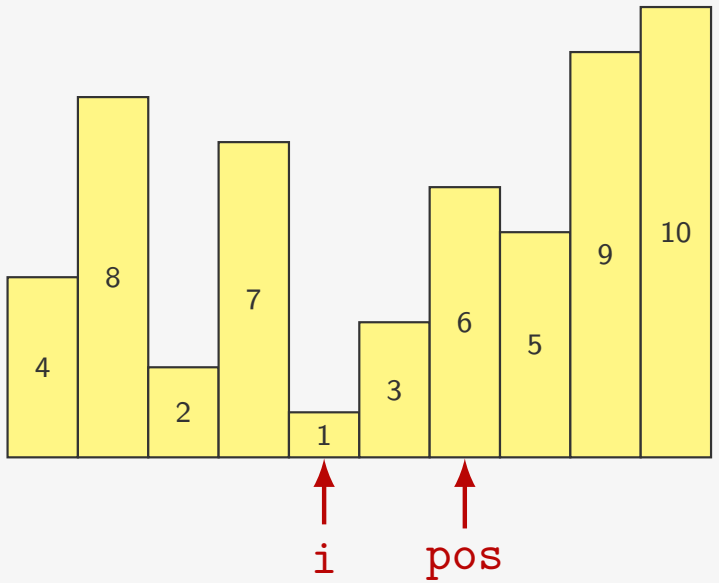
```
1 def partition(l, e, d):
2     pivo = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivo:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
```



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`

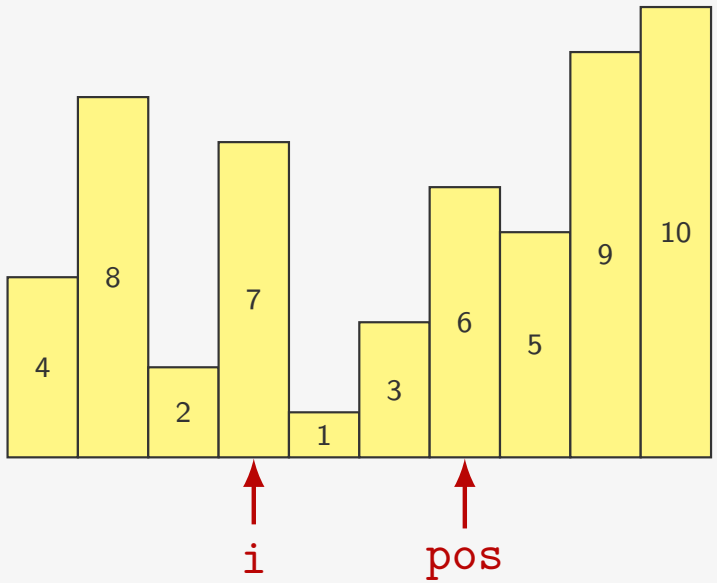
```
1 def partition(l, e, d):
2     pivo = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivo:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
```



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`

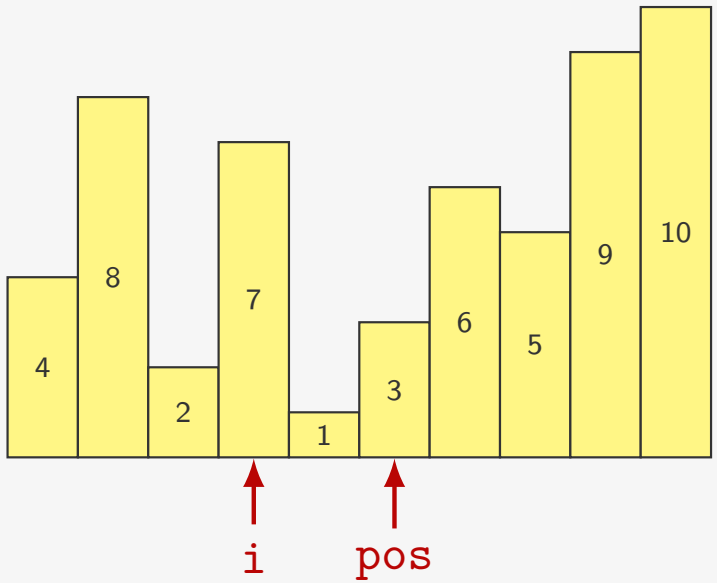
```
1 def partition(l, e, d):
2     pivo = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivo:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
```



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`

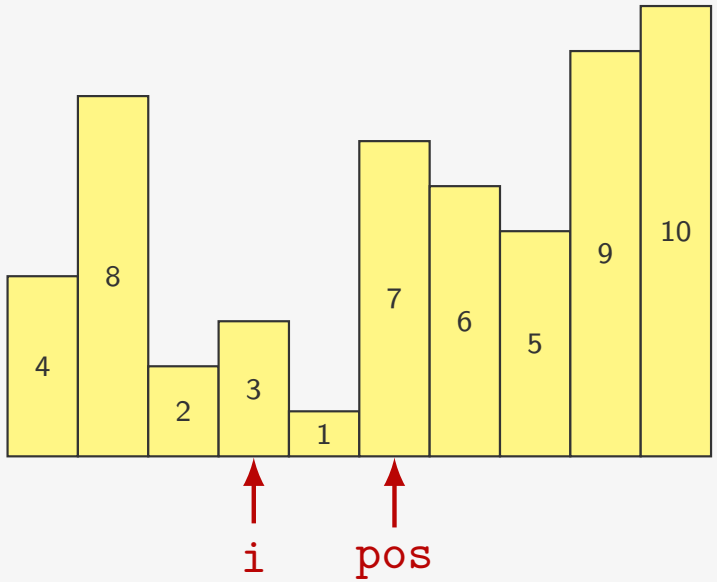
```
1 def partition(l, e, d):
2     pivo = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivo:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
```



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`

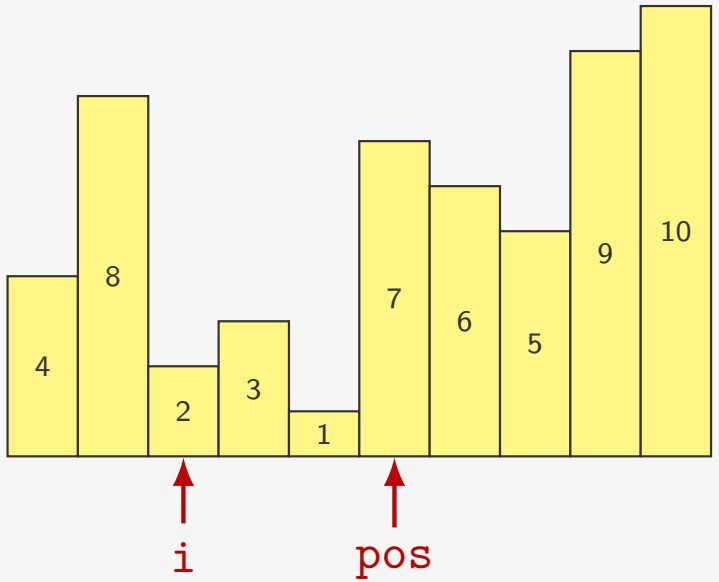
```
1 def partition(l, e, d):
2     pivo = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivo:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
```



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`

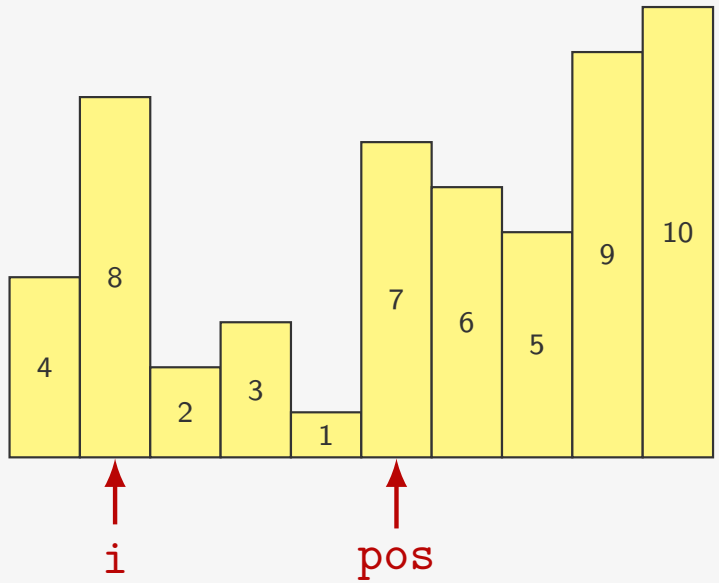
```
1 def partition(l, e, d):
2     pivo = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivo:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
```



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`

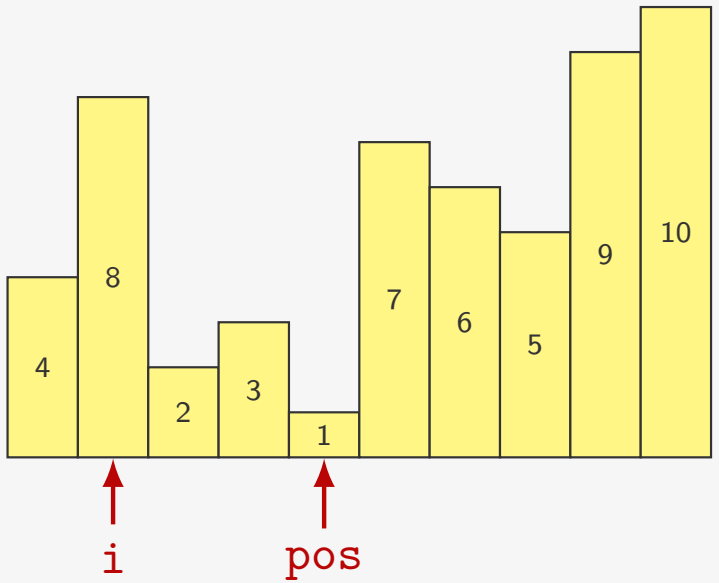
```
1 def partition(l, e, d):
2     pivo = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivo:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
```



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`

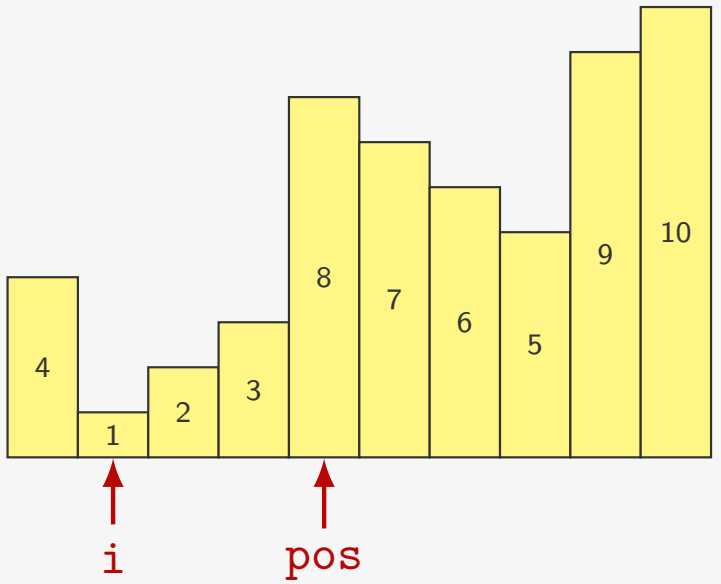
```
1 def partition(l, e, d):
2     pivo = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivo:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
```



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`

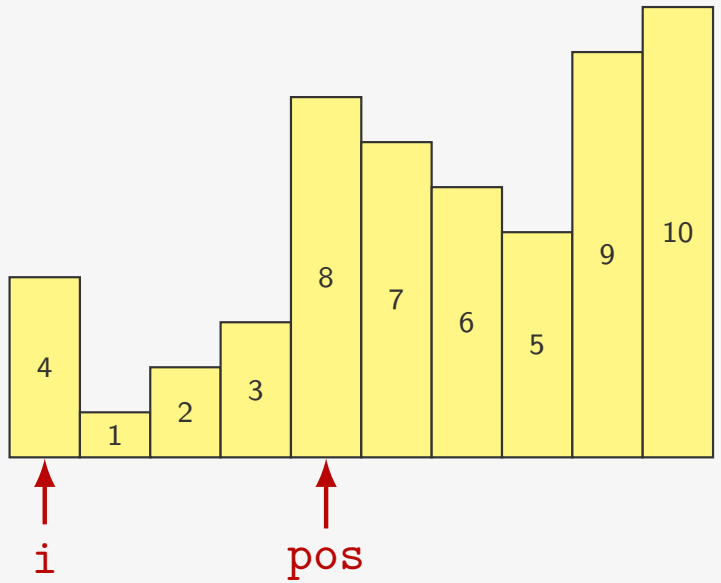
```
1 def partition(l, e, d):
2     pivo = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivo:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
```



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`

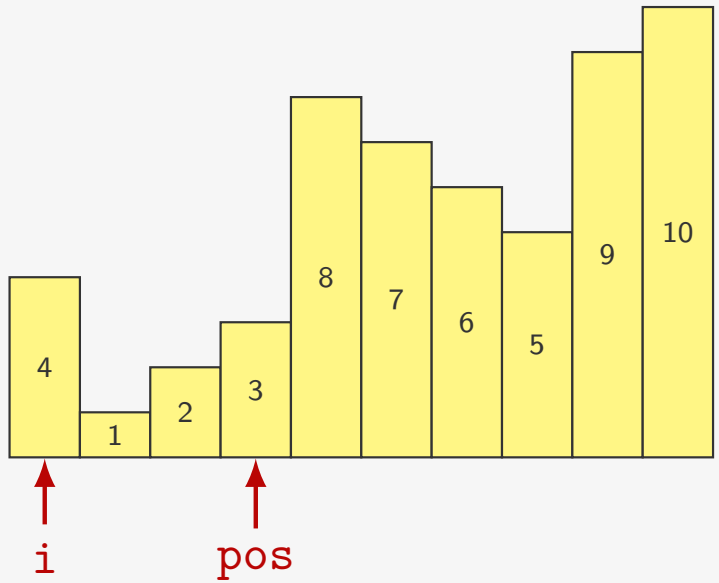
```
1 def partition(l, e, d):
2     pivo = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivo:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
```



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`

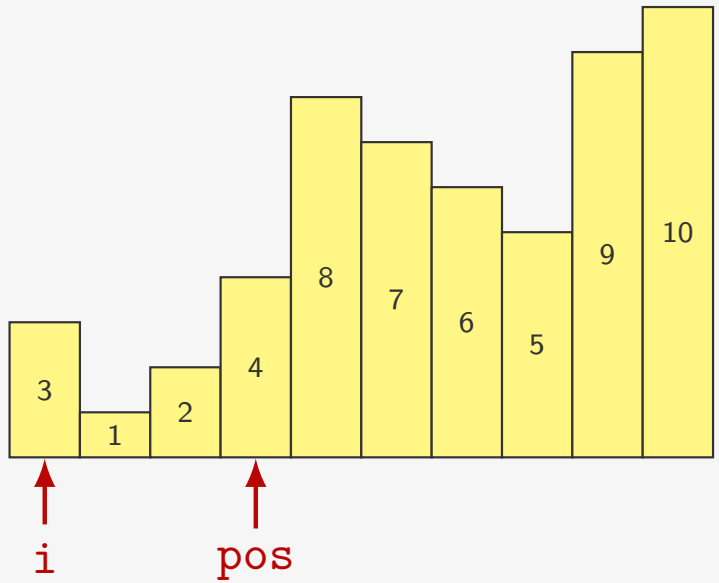
```
1 def partition(l, e, d):
2     pivo = l[e]
3     pos = d + 1
4     for i in range(d, e - 1, -1):
5         if l[i] >= pivo:
6             pos -= 1
7             l[i], l[pos] = l[pos], l[i]
8     return pos
```



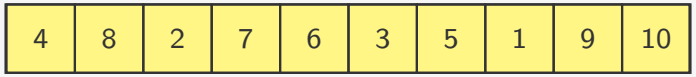
Como particionar uma lista?

- 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     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
```

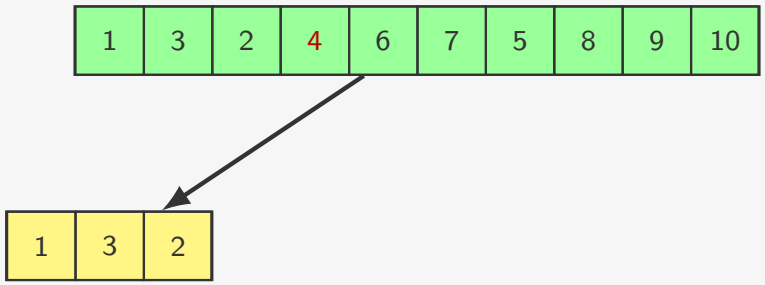


Simulação do Quicksort

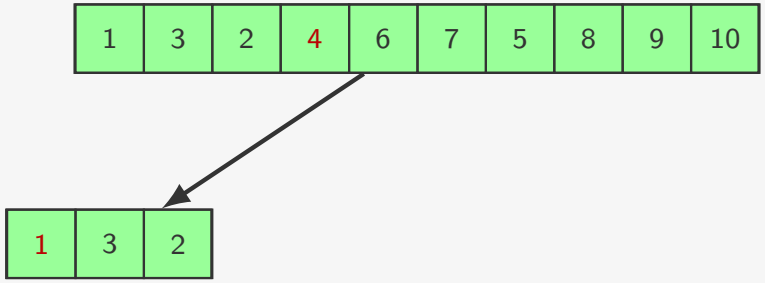


Simulação do Quicksort

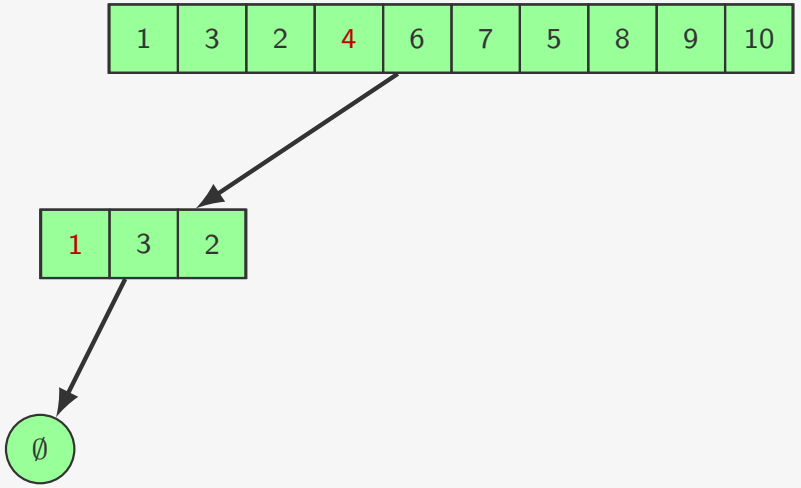
Simulação do Quicksort



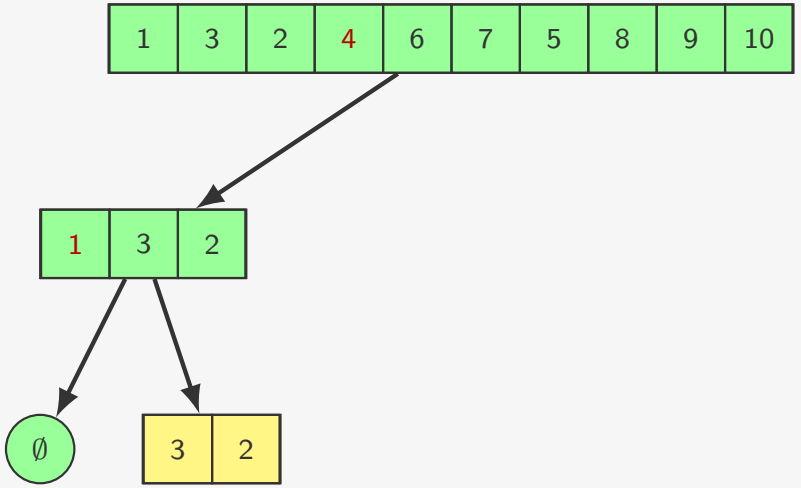
Simulação do Quicksort



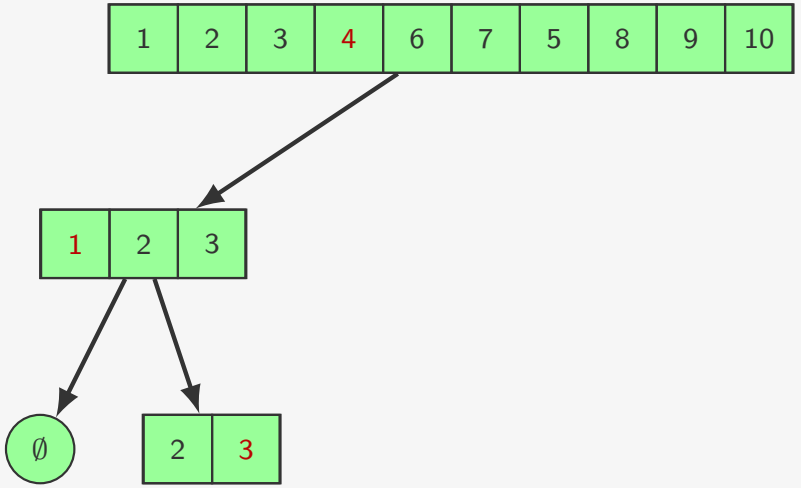
Simulação do Quicksort



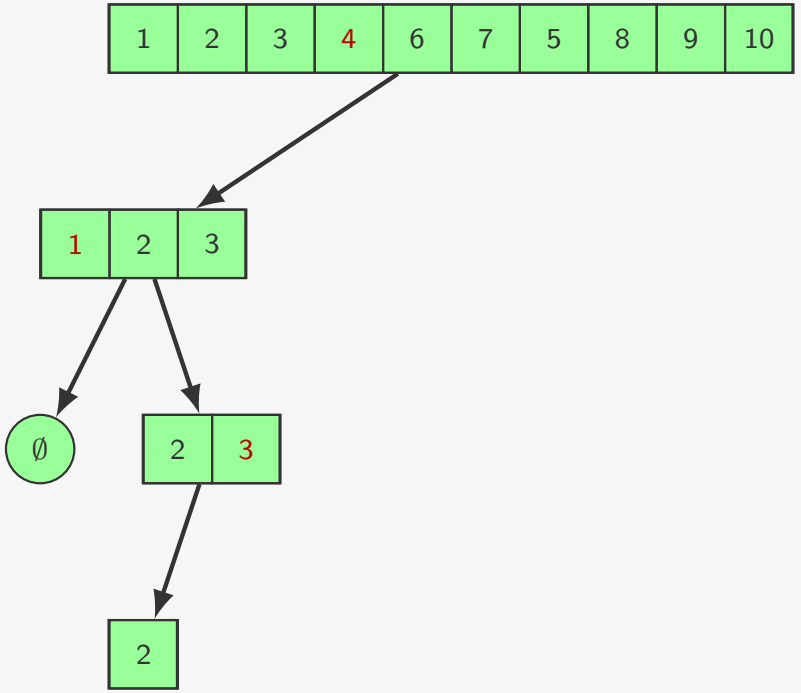
Simulação do Quicksort



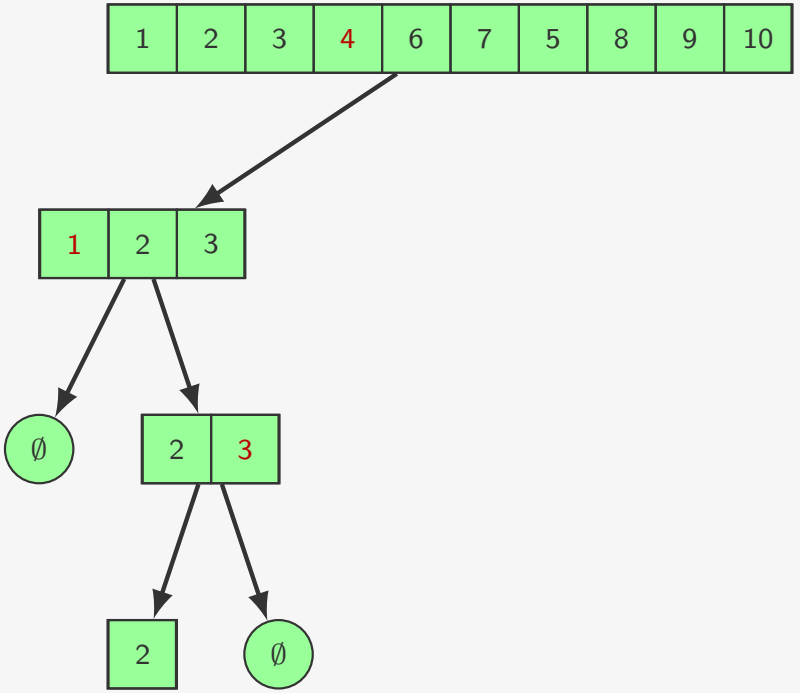
Simulação do Quicksort



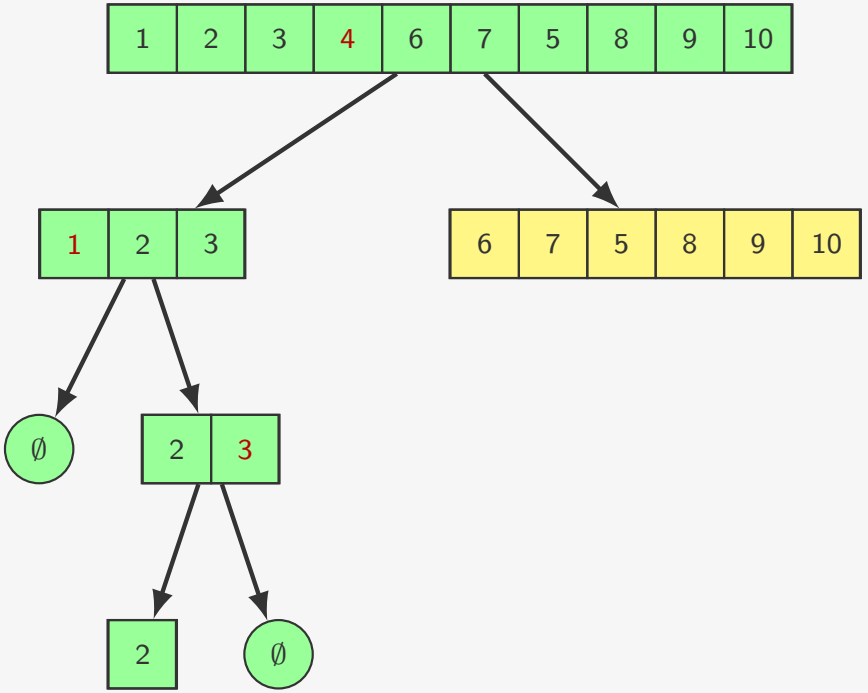
Simulação do Quicksort



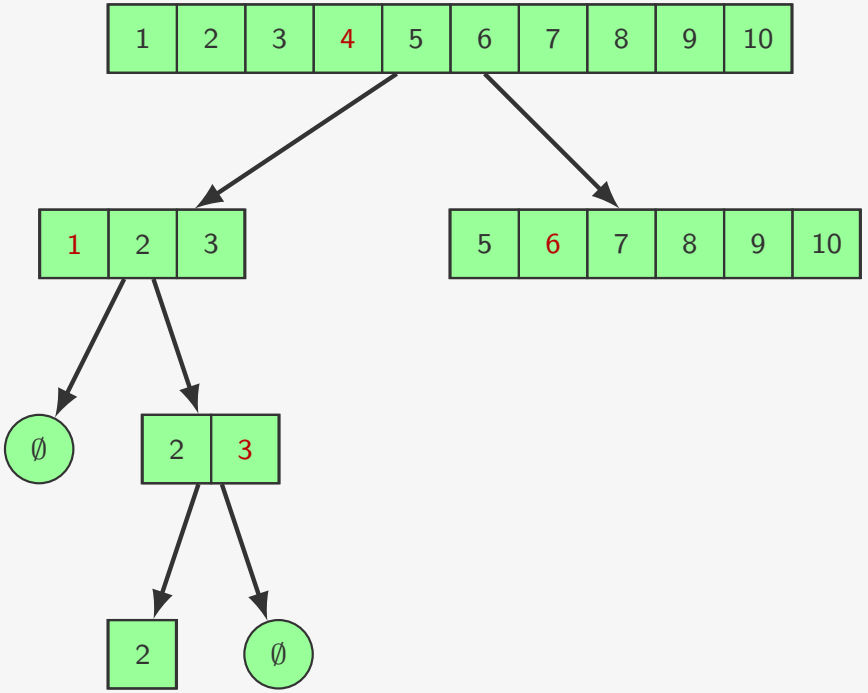
Simulação do Quicksort



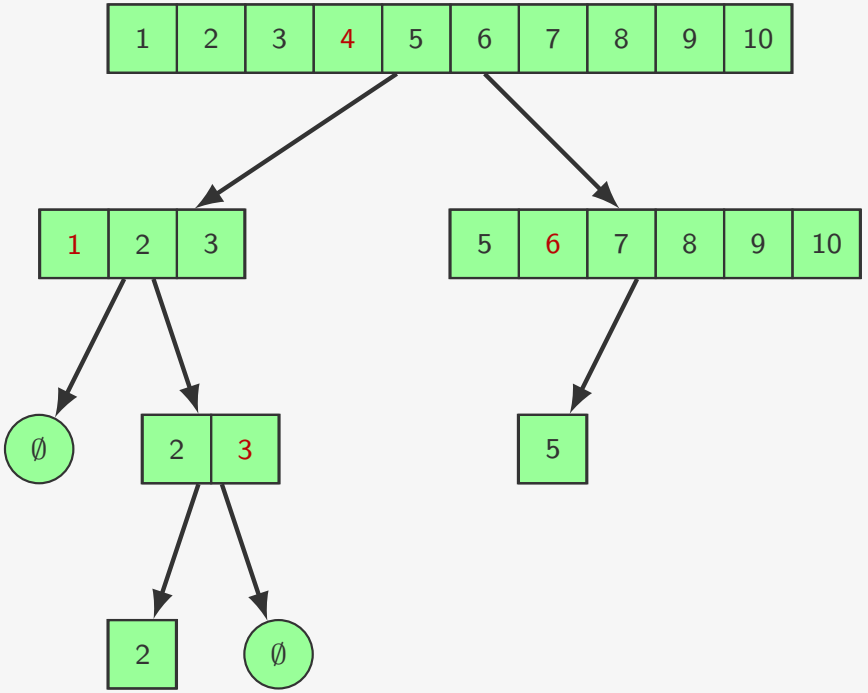
Simulação do Quicksort



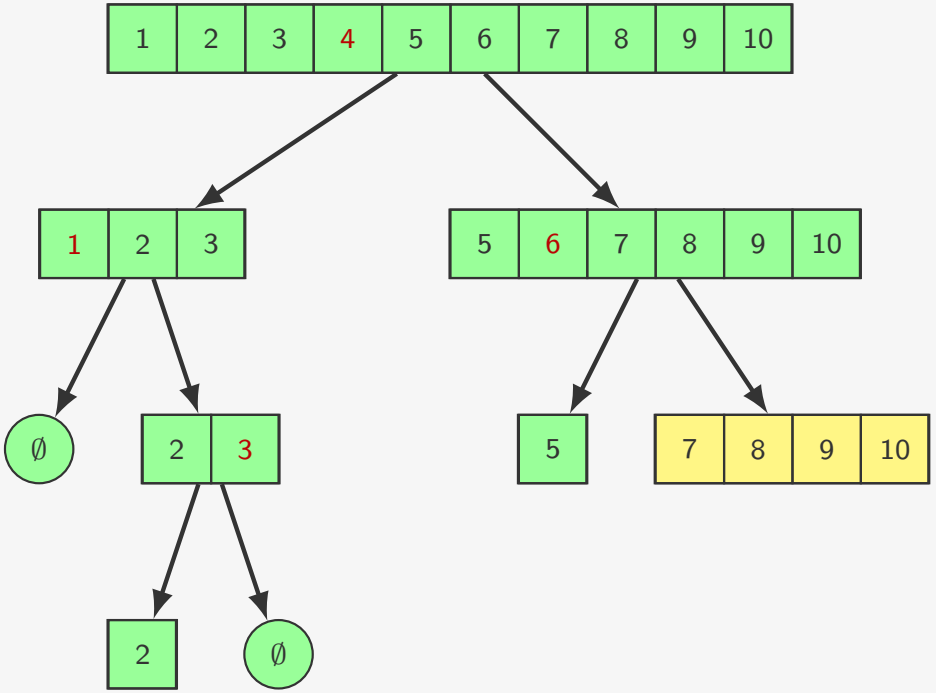
Simulação do Quicksort



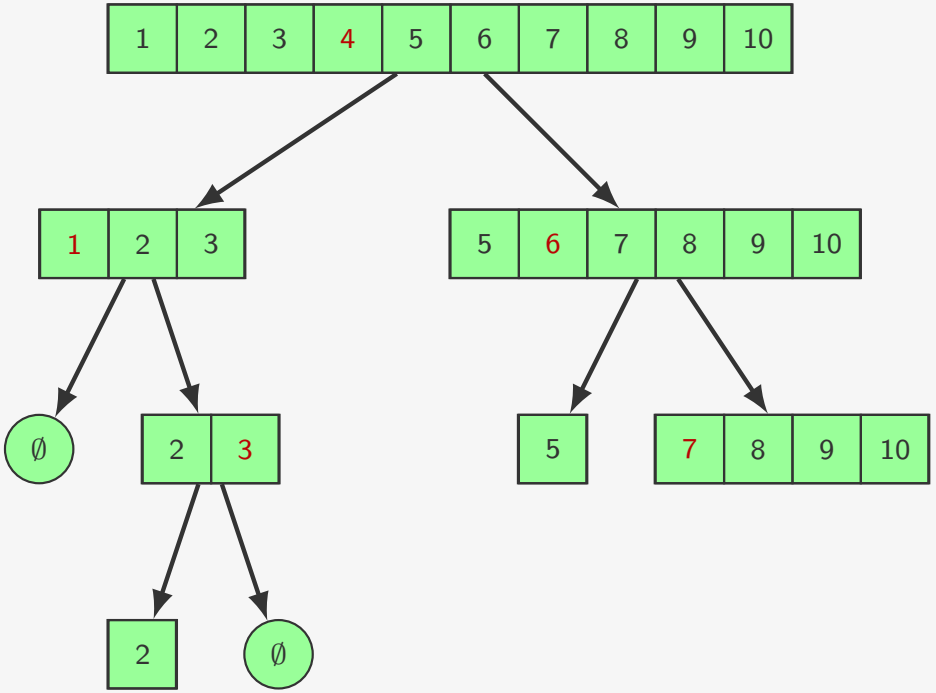
Simulação do Quicksort



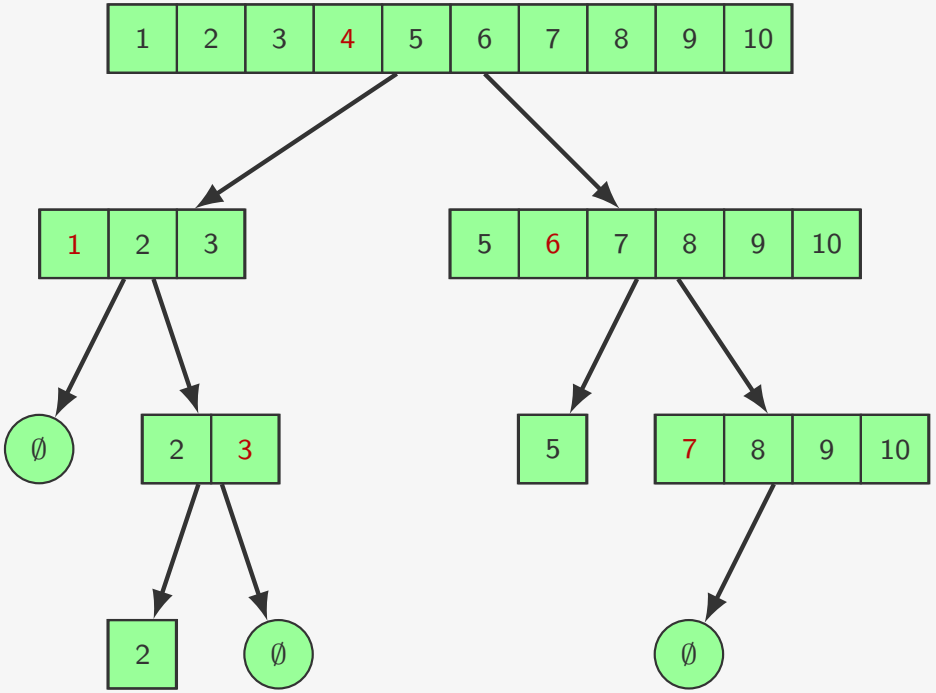
Simulação do Quicksort



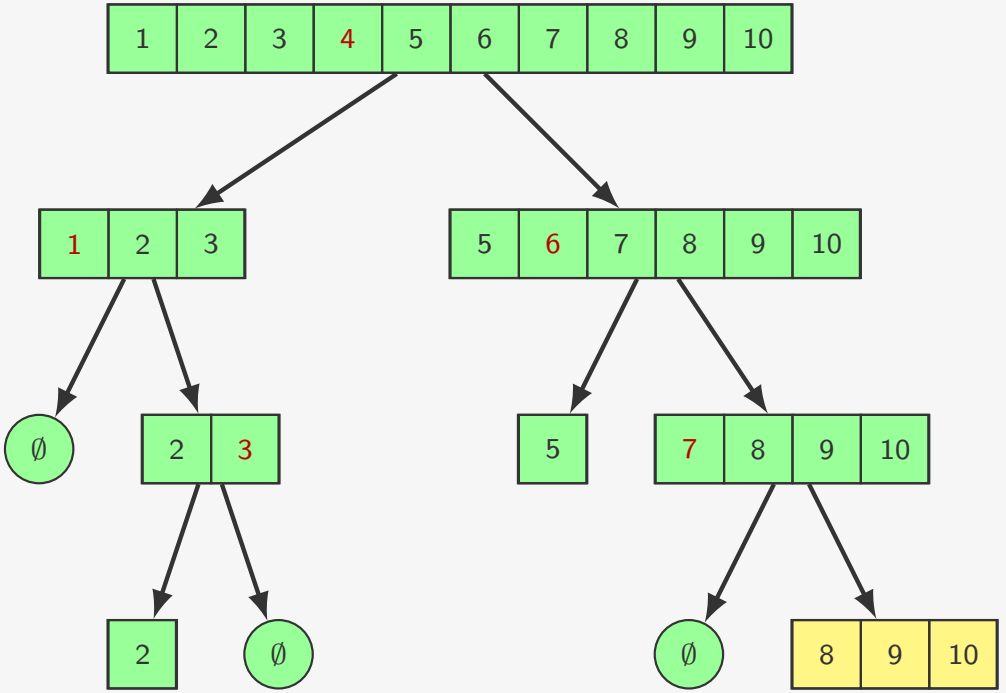
Simulação do Quicksort



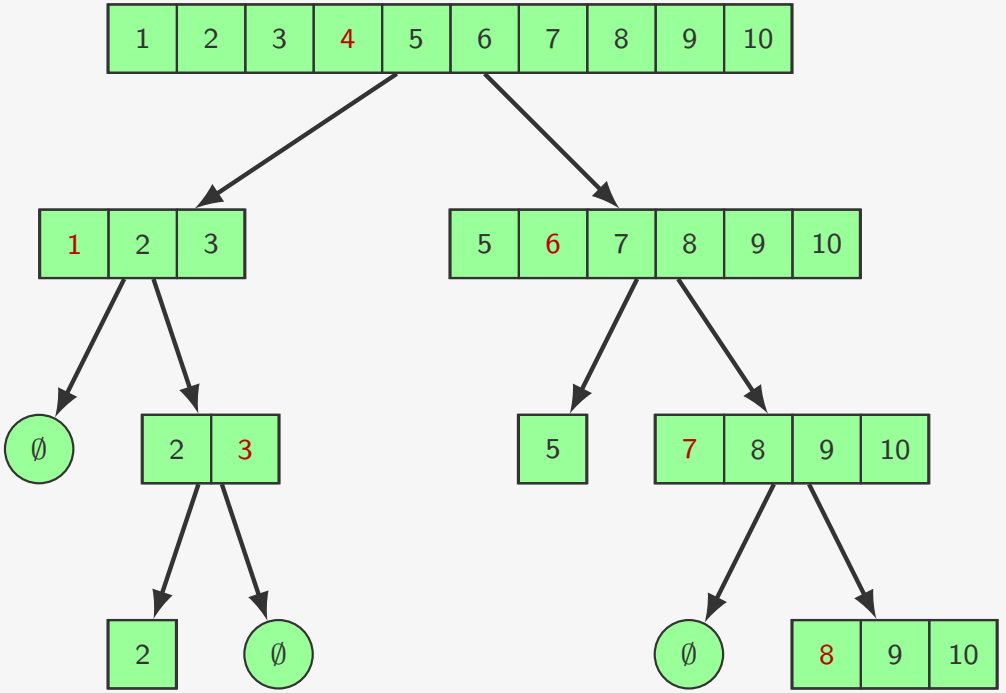
Simulação do Quicksort



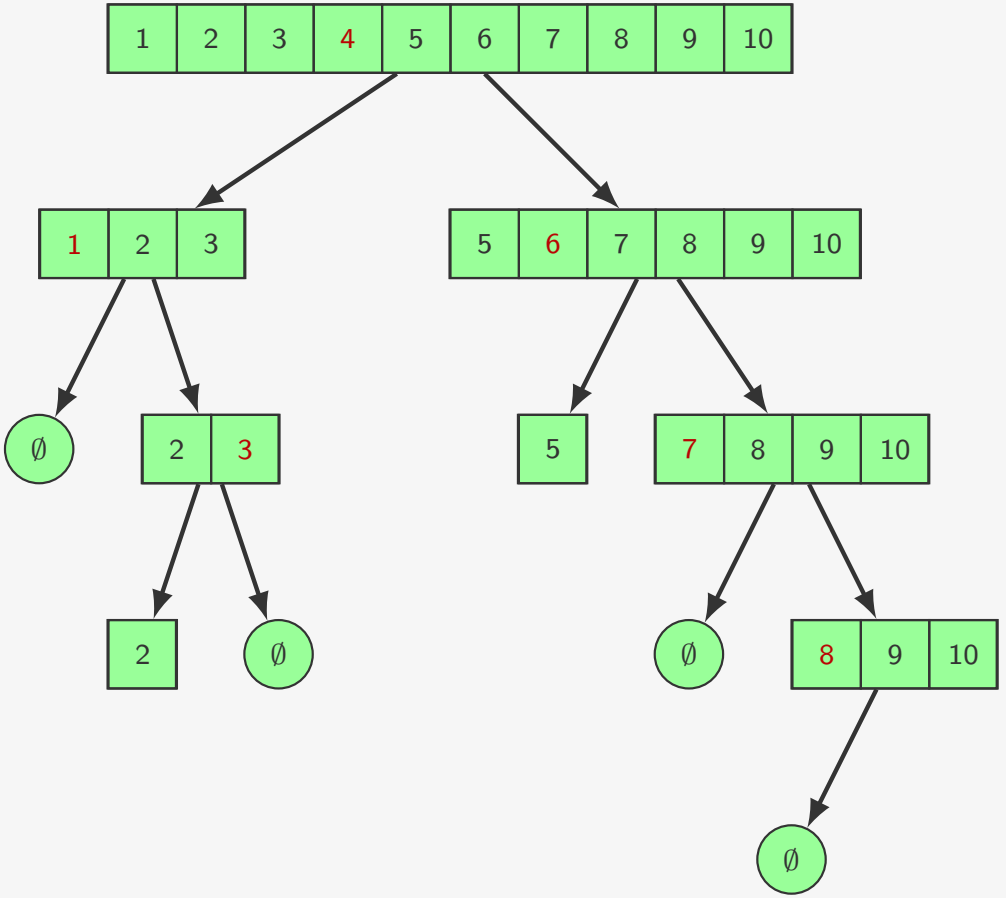
Simulação do Quicksort



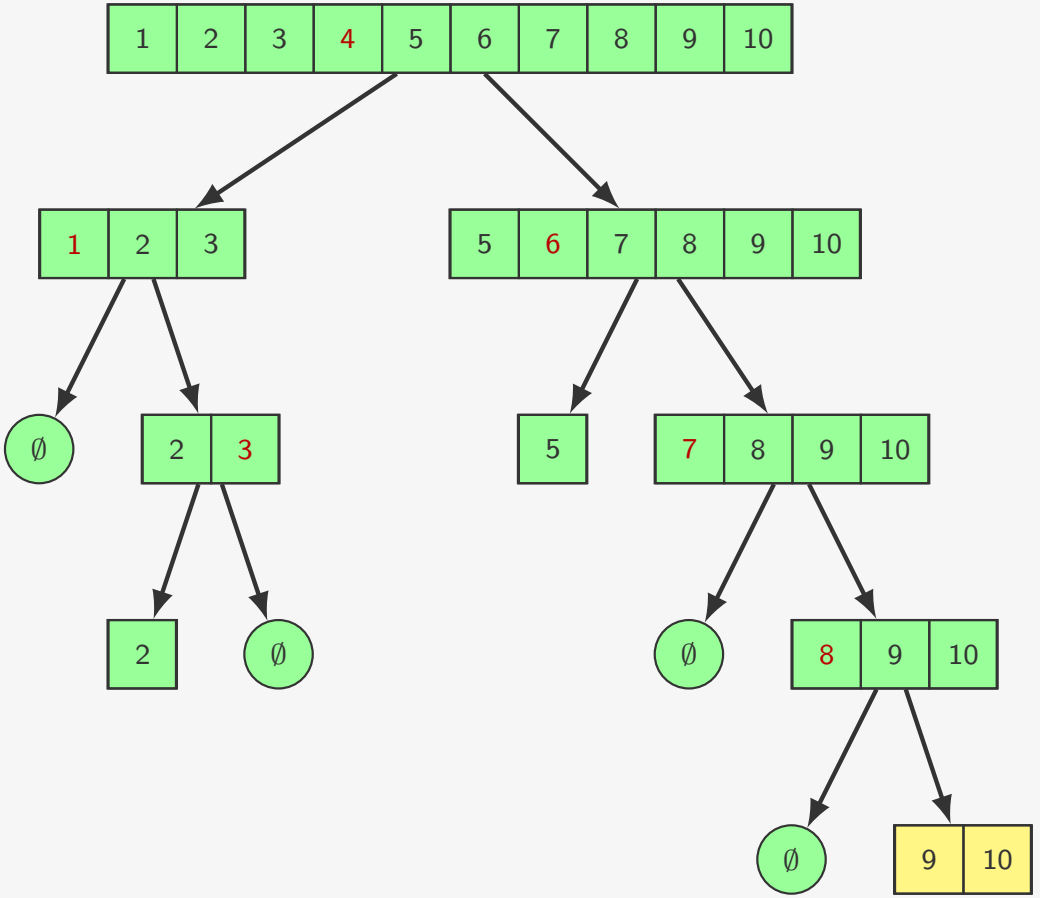
Simulação do Quicksort



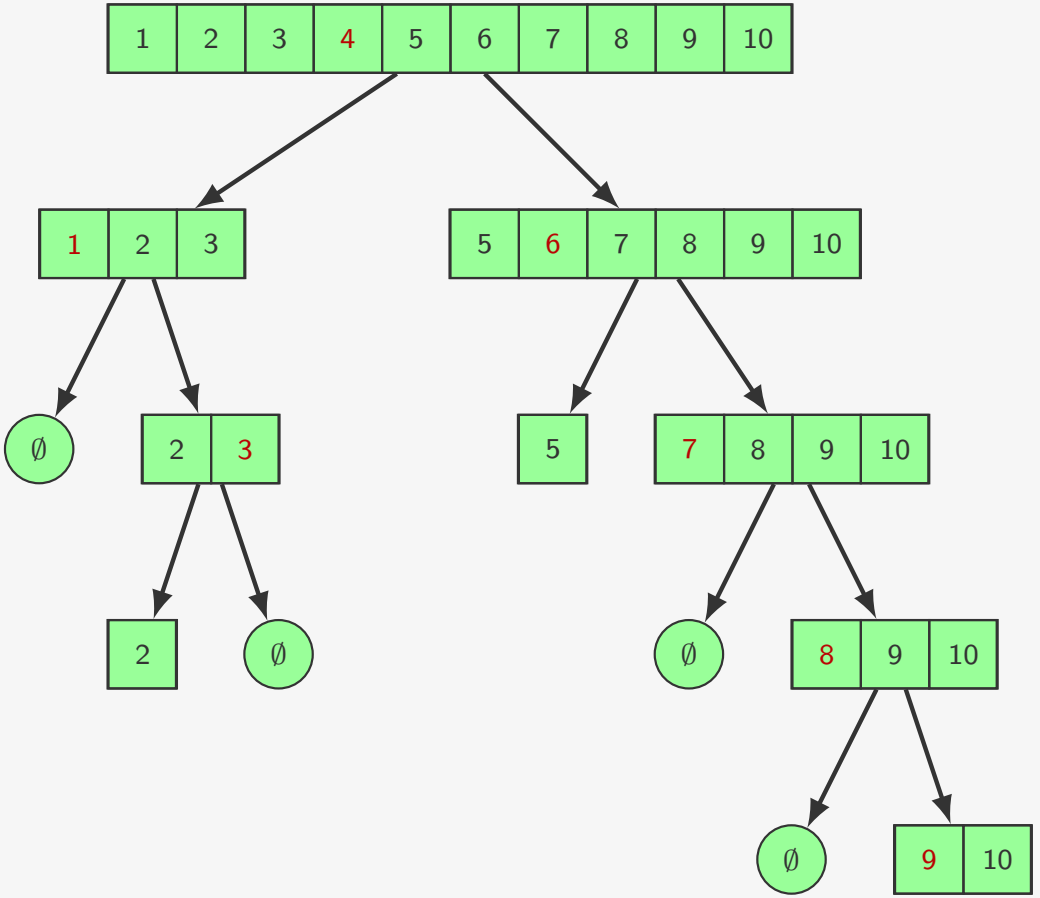
Simulação do Quicksort



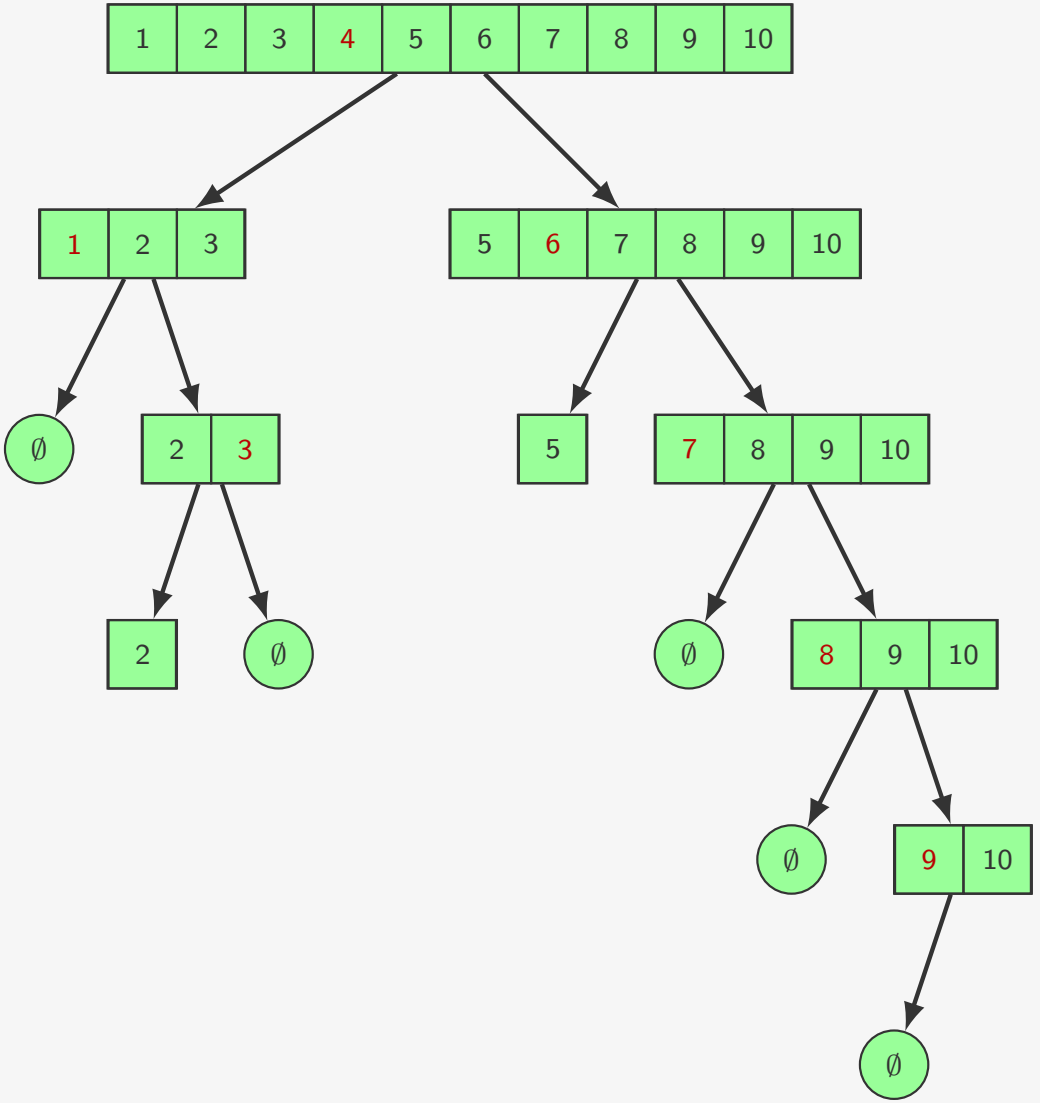
Simulação do Quicksort



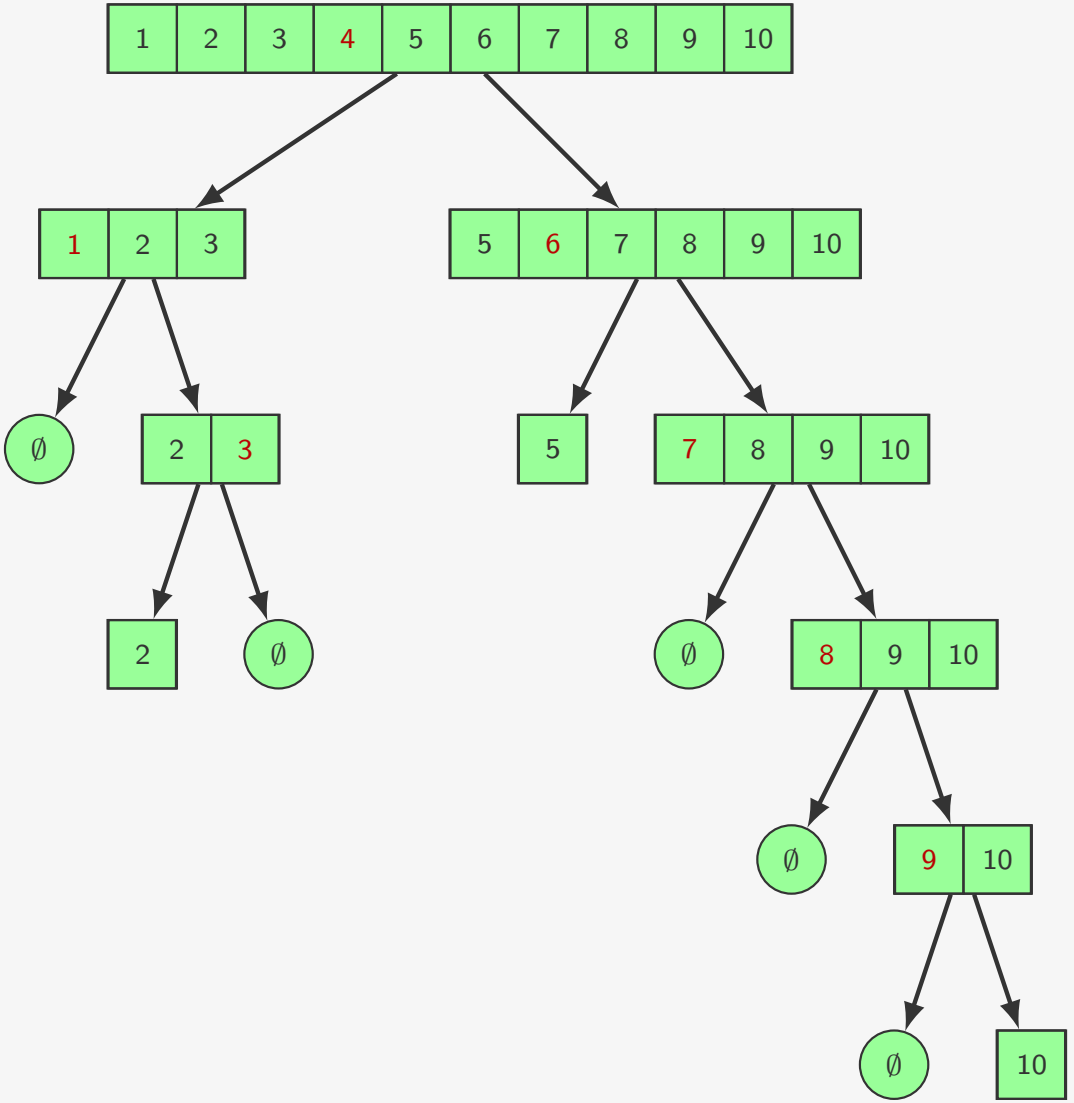
Simulação do Quicksort



Simulação do Quicksort

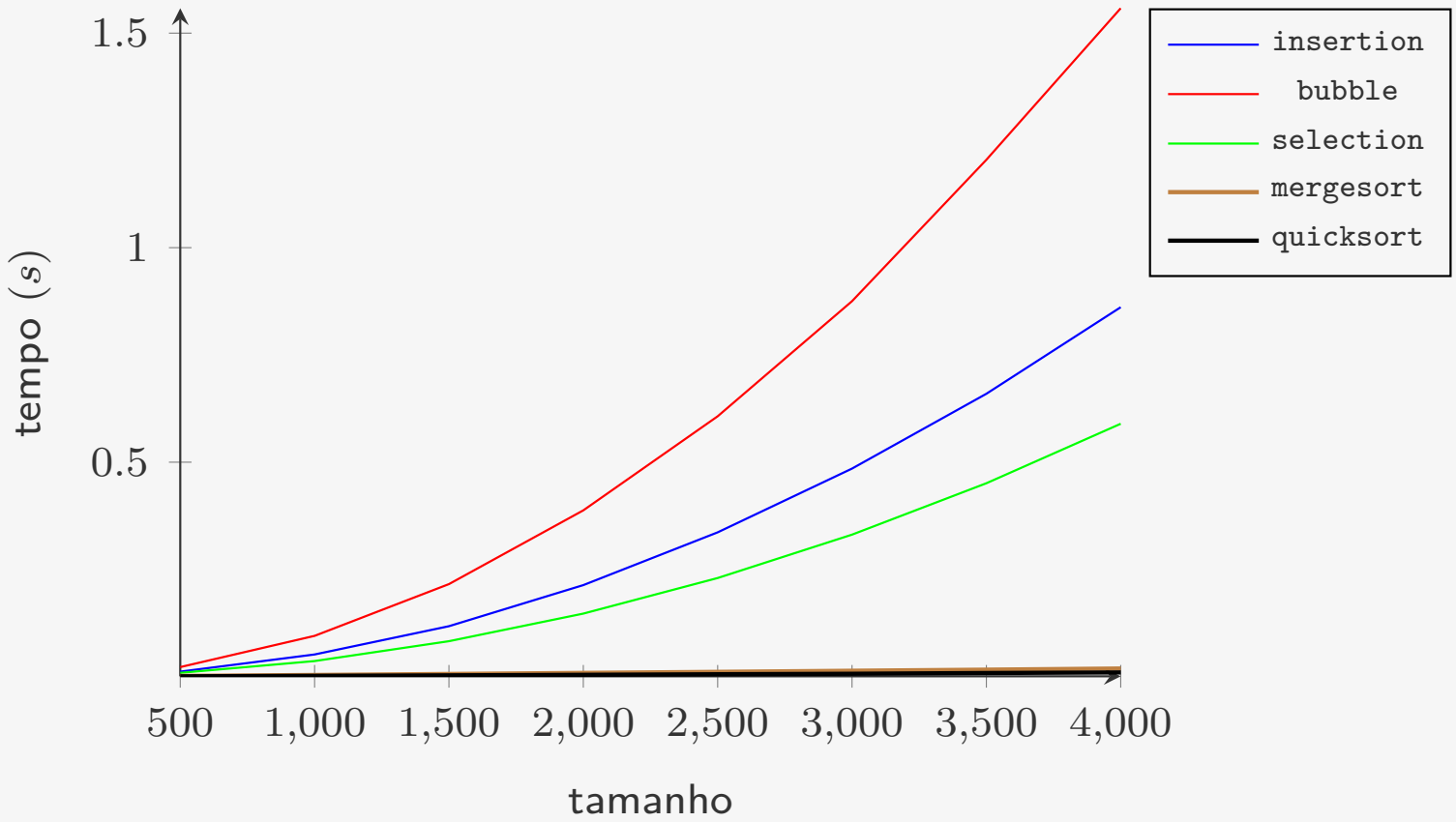


Simulação do Quicksort



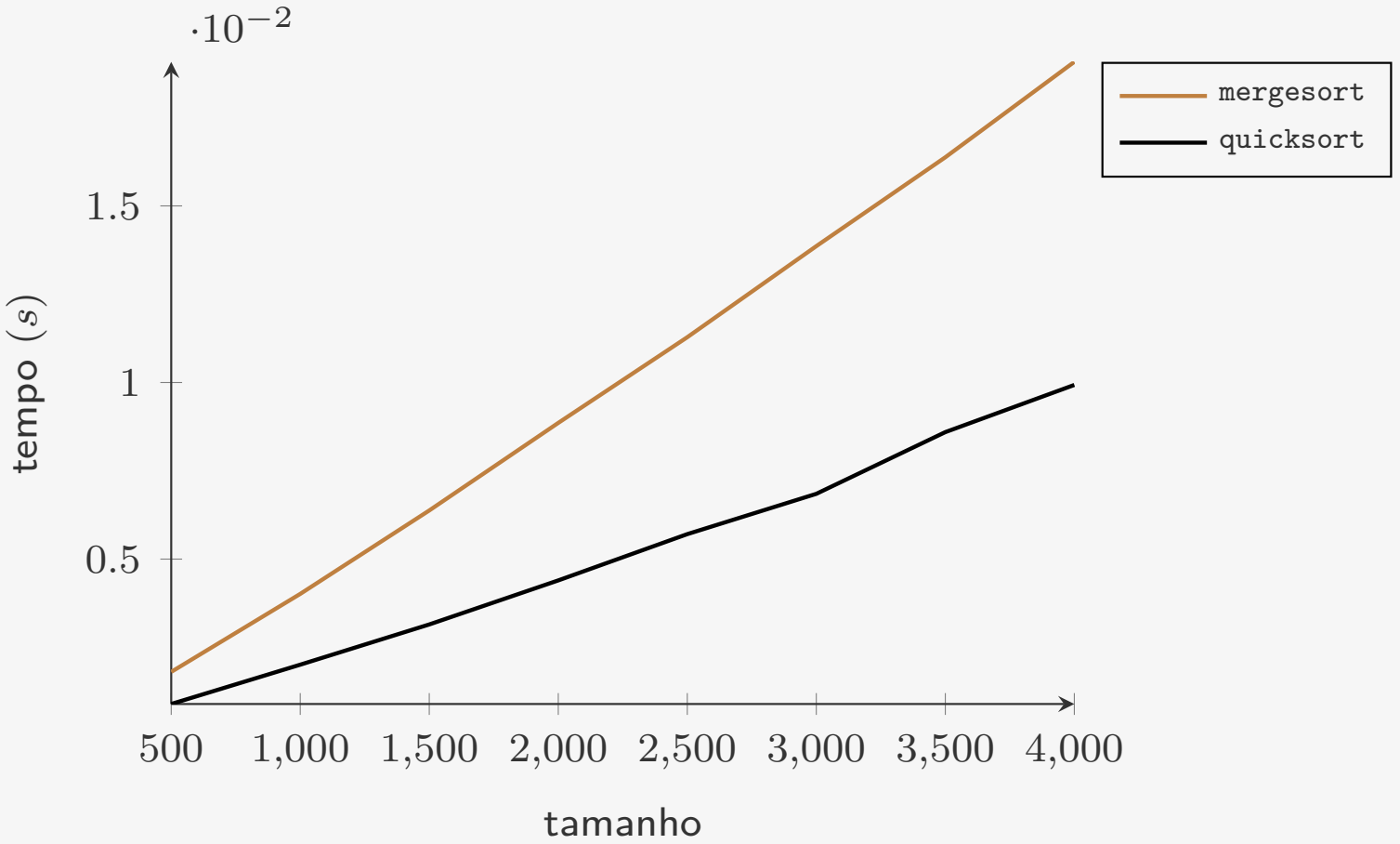
Experimento

- Listas de tamanho 500, 1000, ..., 4000
- Elemento da lista escolhido aleatoriamente entre 0 e 1
- Tiramos a média do tempo de 10 execuções



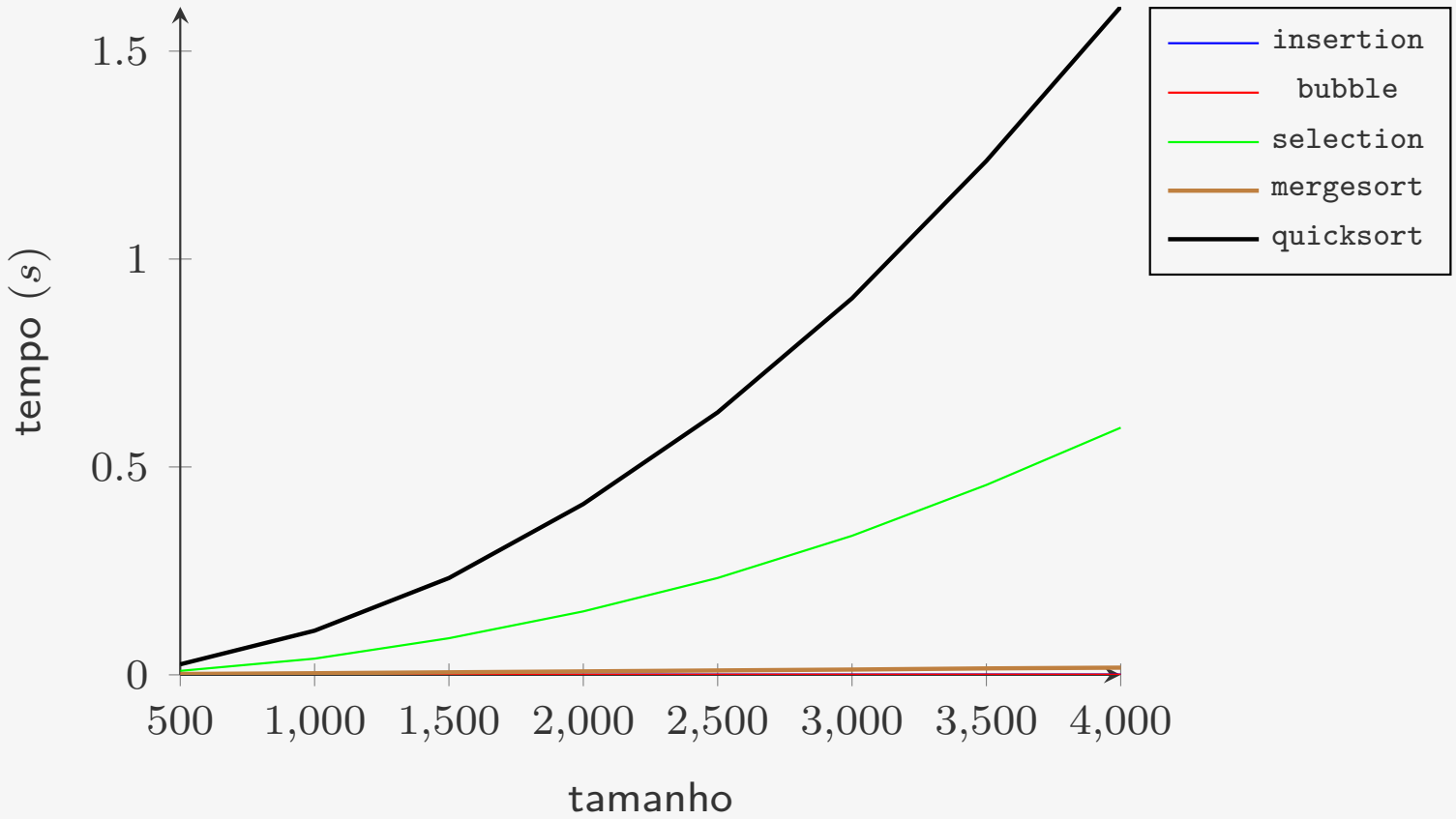
Experimento — Apenas quick e mergesort

- Listas de tamanho 500, 1000, ..., 4000
- Elemento da lista escolhido aleatoriamente entre 0 e 1
- Tiramos a média do tempo de 10 execuções



Experimento 2

- Listas de tamanho 500, 1000, ..., 4000
- As listas já estão ordenadas!
- Tiramos a média do tempo de 10 execuções



Experimento 2

- Listas de tamanho 500, 1000, ..., 4000
- As listas já estão ordenadas!
- Tiramos a média do tempo de 10 execuções