

# MC-202

## Divisão e Conquista, MergeSort e Quicksort

Lehilton Pedrosa  
lehilton@ic.unicamp.br

Universidade Estadual de Campinas

Segundo semestre de 2024

Na unidade anterior...

Vimos três algoritmos de ordenação  $O(n^2)$ :

## Na unidade anterior...

Vimos três algoritmos de ordenação  $O(n^2)$ :

- `selectionsort`

## Na unidade anterior...

Vimos três algoritmos de ordenação  $O(n^2)$ :

- `selectionsort`
- `bubblesort`

## Na unidade anterior...

Vimos três algoritmos de ordenação  $O(n^2)$ :

- `selectionsort`
- `bubblesort`
- `insertionsort`

## Na unidade anterior...

Vimos três algoritmos de ordenação  $O(n^2)$ :

- `selectionsort`
- `bubblesort`
- `insertionsort`

E um algoritmo de ordenação  $O(n \lg n)$

## Na unidade anterior...

Vimos três algoritmos de ordenação  $O(n^2)$ :

- `selectionsort`
- `bubblesort`
- `insertionsort`

E um algoritmo de ordenação  $O(n \lg n)$

- `heapsort`

## Na unidade anterior...

Vimos três algoritmos de ordenação  $O(n^2)$ :

- `selectionsort`
- `bubblesort`
- `insertionsort`

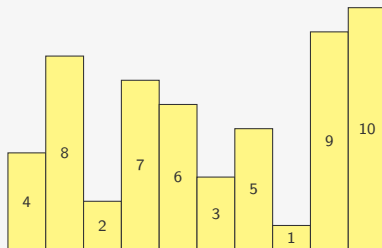
E um algoritmo de ordenação  $O(n \lg n)$

- `heapsort`

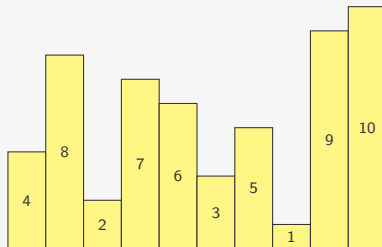
Nessa unidade veremos mais dois algoritmos de ordenação



## Estratégia: recursão

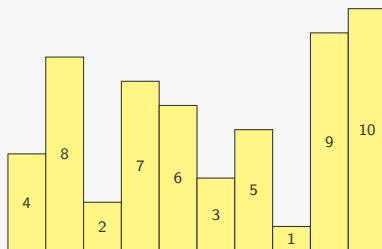


## Estratégia: recursão



Como ordenar a primeira metade do vetor?

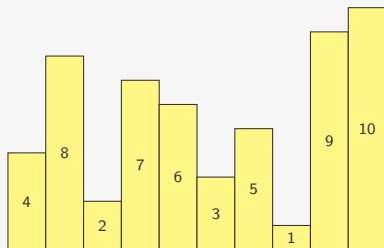
## Estratégia: recursão



Como ordenar a primeira metade do vetor?

- usamos uma função `ordenar(int *v, int l, int r)`

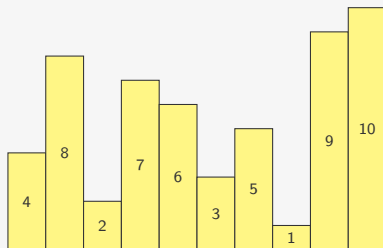
## Estratégia: recursão



Como ordenar a primeira metade do vetor?

- usamos uma função `ordenar(int *v, int l, int r)`
  - ordena o vetor das posições `l` a `r` (inclusive)

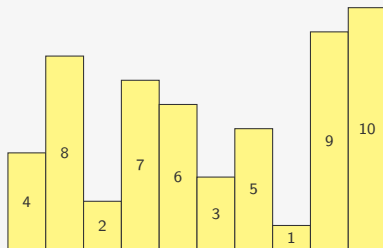
## Estratégia: recursão



Como ordenar a primeira metade do vetor?

- usamos uma função `ordenar(int *v, int l, int r)`
  - ordena o vetor das posições `l` a `r` (inclusive)
  - poderia ser um dos algoritmos vistos anteriormente

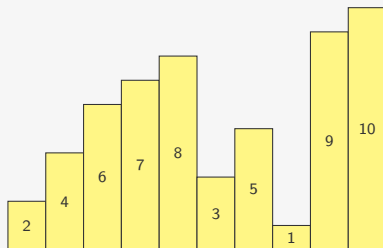
## Estratégia: recursão



Como ordenar a primeira metade do vetor?

- usamos uma função `ordenar(int *v, int l, int r)`
  - ordena o vetor das posições `l` a `r` (inclusive)
  - poderia ser um dos algoritmos vistos anteriormente
  - mas faremos algo mais simples e melhor

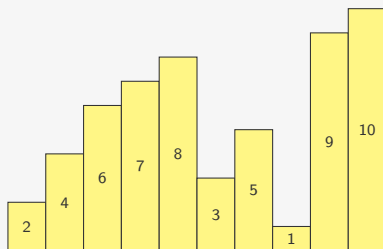
## Estratégia: recursão



Como ordenar a primeira metade do vetor?

- usamos uma função `ordenar(int *v, int l, int r)`
  - ordena o vetor das posições `l` a `r` (inclusive)
  - poderia ser um dos algoritmos vistos anteriormente
  - mas faremos algo mais simples e melhor
- executamos `ordenar(v, 0, 4);`

## Estratégia: recursão



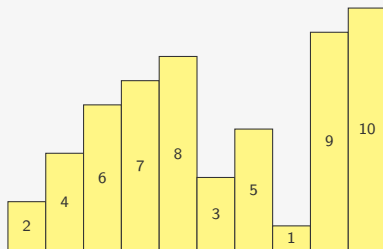
Como ordenar a primeira metade do vetor?

- usamos uma função `ordenar(int *v, int l, int r)`
  - ordena o vetor das posições `l` a `r` (inclusive)
  - poderia ser um dos algoritmos vistos anteriormente
  - mas faremos algo mais simples e melhor
- executamos `ordenar(v, 0, 4);`

E se quiséssemos ordenar a segunda parte?



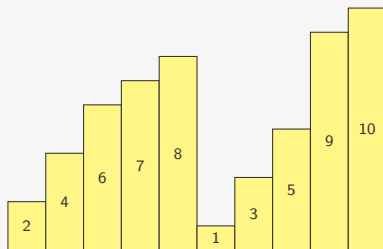
## Ordenando a segunda parte



Para ordenar a segunda metade:

- executamos `ordenar(v, 5, 9);`

## Ordenando a segunda parte



Para ordenar a segunda metade:

- executamos `ordenar(v, 5, 9);`

## Ordenando todo o vetor

Se temos um vetor com as suas duas metades já ordenadas

## Ordenando todo o vetor

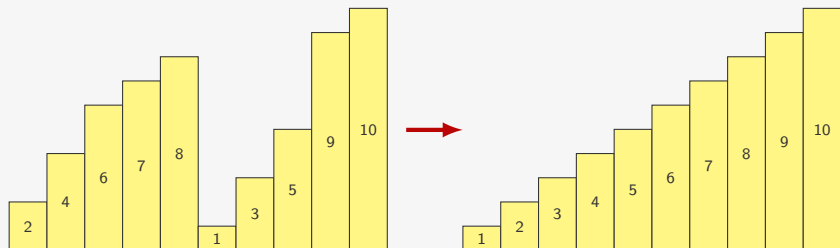
Se temos um vetor com as suas duas metades já ordenadas

- Como ordenar todo o vetor?

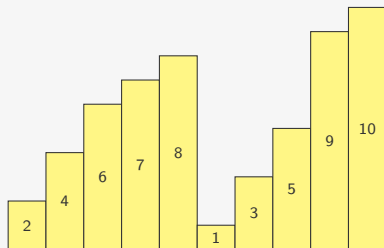
## Ordenando todo o vetor

Se temos um vetor com as suas duas metades já ordenadas

- Como ordenar todo o vetor?

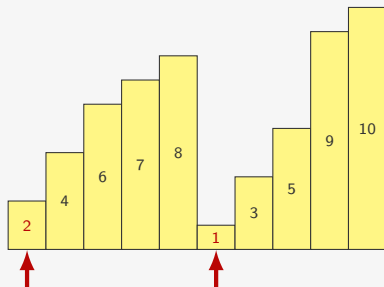


# Intercalando



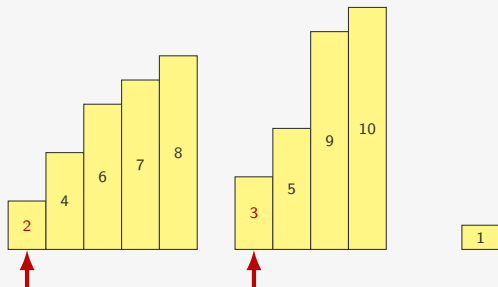
- Percorremos os dois subvetores

# Intercalando



- Percorremos os dois subvetores
- Pegamos o **mínimo** e inserimos em um vetor auxiliar

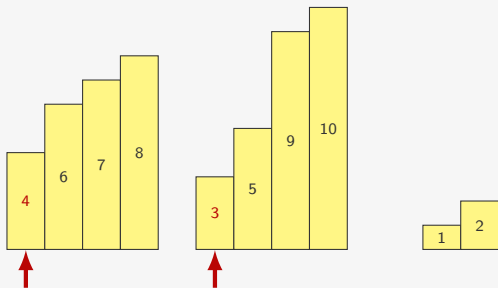
# Intercalando



- Percorremos os dois subvetores
- Pegamos o **mínimo** e inserimos em um vetor auxiliar

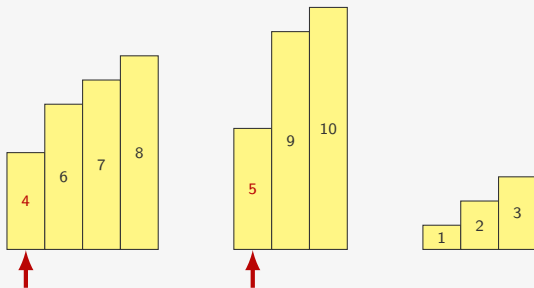


# Intercalando



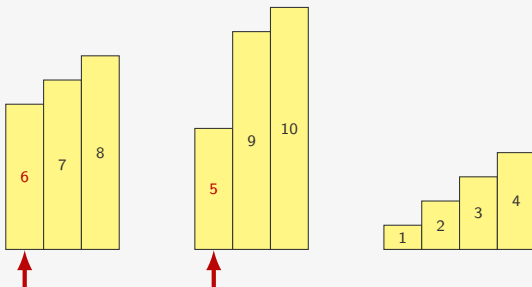
- Percorremos os dois subvetores
- Pegamos o **mínimo** e inserimos em um vetor auxiliar

# Intercalando



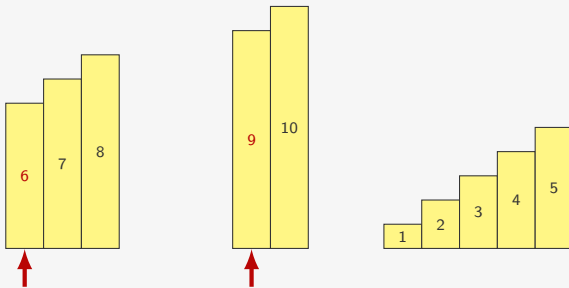
- Percorremos os dois subvetores
- Pegamos o **mínimo** e inserimos em um vetor auxiliar

# Intercalando



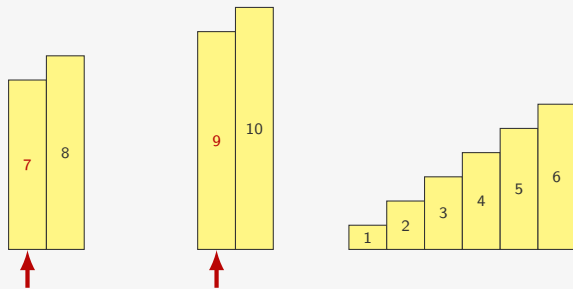
- Percorremos os dois subvetores
- Pegamos o **mínimo** e inserimos em um vetor auxiliar

# Intercalando



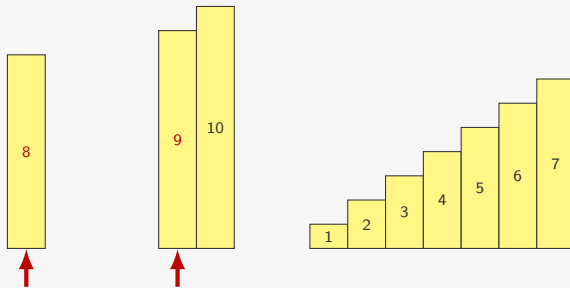
- Percorremos os dois subvetores
- Pegamos o **mínimo** e inserimos em um vetor auxiliar

# Intercalando



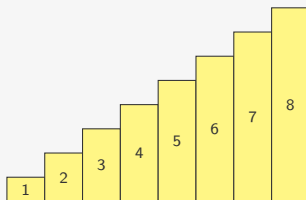
- Percorremos os dois subvetores
- Pegamos o **mínimo** e inserimos em um vetor auxiliar

# Intercalando



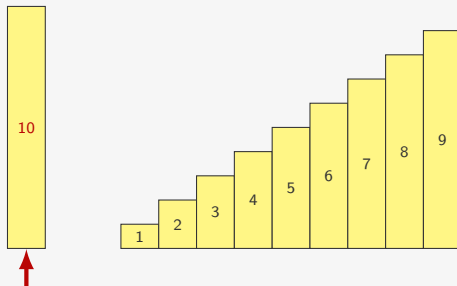
- Percorremos os dois subvetores
- Pegamos o **mínimo** e inserimos em um vetor auxiliar

# Intercalando



- Percorremos os dois subvetores
- Pegamos o **mínimo** e inserimos em um vetor auxiliar
- Depois copiamos o restante

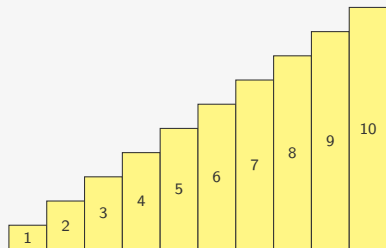
# Intercalando



- Percorremos os dois subvetores
- Pegamos o **mínimo** e inserimos em um vetor auxiliar
- Depois copiamos o restante

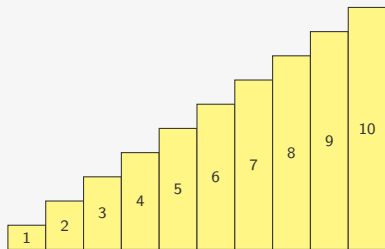


# Intercalando



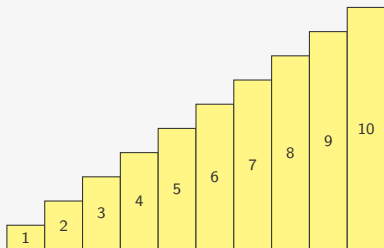
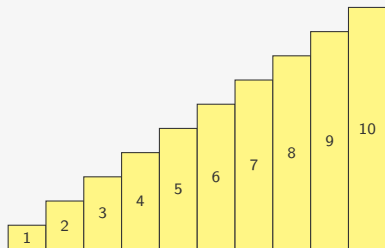
- Percorremos os dois subvetores
- Pegamos o **mínimo** e inserimos em um vetor auxiliar
- Depois copiamos o restante

# Intercalando



- Percorremos os dois subvetores
- Pegamos o **mínimo** e inserimos em um vetor auxiliar
- Depois copiamos o restante
- No final, copiamos do vetor auxiliar para o original

# Intercalando



- Percorremos os dois subvetores
- Pegamos o **mínimo** e inserimos em um vetor auxiliar
- Depois copiamos o restante
- No final, copiamos do vetor auxiliar para o 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 um vetor a ser ordenado em dois

# 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 um vetor a ser ordenado em dois
- **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 um vetor a ser ordenado em dois
- **Conquista:** Combinamos a solução dos problemas menores
  - ex: intercalamos os dois vetores ordenados

# Ordenação por intercalação (*MergeSort*)

Intercalação:

# Ordenação por intercalação (*MergeSort*)

Intercalação:

- Os dois subvetores estão armazenados em  $v$ :

# Ordenação por intercalação (*MergeSort*)

Intercalação:

- Os dois subvetores estão armazenados em  $v$ :
  - O primeiro nas posições de  $l$  até  $m$

# Ordenação por intercalação (*MergeSort*)

Intercalação:

- Os dois subvetores estão armazenados em  $v$ :
  - O primeiro nas posições de  $l$  até  $m$
  - O segundo nas posições de  $m + 1$  até  $r$

# Ordenação por intercalação (*MergeSort*)

Intercalação:

- Os dois subvetores estão armazenados em  $v$ :
  - O primeiro nas posições de  $l$  até  $m$
  - O segundo nas posições de  $m + 1$  até  $r$
- Precisamos de um vetor auxiliar do tamanho do vetor



# Ordenação por intercalação (*MergeSort*)

Intercalação:

- Os dois subvetores estão armazenados em  $v$ :
  - O primeiro nas posições de  $l$  até  $m$
  - O segundo nas posições de  $m + 1$  até  $r$
- Precisamos de um vetor auxiliar do tamanho do vetor
- Vamos considerar que o maior vetor tem tamanho **MAX**

# Ordenação por intercalação (*MergeSort*)

Intercalação:

- Os dois subvetores estão armazenados em **v**:
  - O primeiro nas posições de **l** até **m**
  - O segundo nas posições de **m + 1** até **r**
- Precisamos de um vetor auxiliar do tamanho do vetor
- Vamos considerar que o maior vetor tem tamanho **MAX**
  - Exemplo `#define MAX 100`

# Ordenação por intercalação (*MergeSort*)

```
1 void merge(int *v, int l, int m, int r) {
```

## Ordenação por intercalação (*MergeSort*)

```
1 void merge(int *v, int l, int m, int r) {  
2     int aux[MAX];  
3     int i = l, j = m + 1, k = 0;
```

## Ordenação por intercalação (*MergeSort*)

```
1 void merge(int *v, int l, int m, int r) {
2     int aux[MAX];
3     int i = l, j = m + 1, k = 0;
4     /*intercala*/
5     while (i <= m && j <= r)
6         if (v[i] <= v[j])
```

## Ordenação por intercalação (*MergeSort*)

```
1 void merge(int *v, int l, int m, int r) {
2     int aux[MAX];
3     int i = l, j = m + 1, k = 0;
4     /*intercala*/
5     while (i <= m && j <= r)
6         if (v[i] <= v[j])
7             aux[k++] = v[i++];
```

## Ordenação por intercalação (*MergeSort*)

```
1 void merge(int *v, int l, int m, int r) {
2     int aux[MAX];
3     int i = l, j = m + 1, k = 0;
4     /*intercala*/
5     while (i <= m && j <= r)
6         if (v[i] <= v[j])
7             aux[k++] = v[i++];
8         else
9             aux[k++] = v[j++];
```

## Ordenação por intercalação (*MergeSort*)

```
1 void merge(int *v, int l, int m, int r) {
2     int aux[MAX];
3     int i = l, j = m + 1, k = 0;
4     /*intercala*/
5     while (i <= m && j <= r)
6         if (v[i] <= v[j])
7             aux[k++] = v[i++];
8         else
9             aux[k++] = v[j++];
10    /*copia o resto do subvetor que não terminou*/
11    while (i <= m)
12        aux[k++] = v[i++];
```



## Ordenação por intercalação (*MergeSort*)

```
1 void merge(int *v, int l, int m, int r) {
2     int aux[MAX];
3     int i = l, j = m + 1, k = 0;
4     /*intercala*/
5     while (i <= m && j <= r)
6         if (v[i] <= v[j])
7             aux[k++] = v[i++];
8         else
9             aux[k++] = v[j++];
10    /*copia o resto do subvetor que não terminou*/
11    while (i <= m)
12        aux[k++] = v[i++];
13    while (j <= r)
14        aux[k++] = v[j++];
```

## Ordenação por intercalação (*MergeSort*)

```
1 void merge(int *v, int l, int m, int r) {
2     int aux[MAX];
3     int i = l, j = m + 1, k = 0;
4     /*intercala*/
5     while (i <= m && j <= r)
6         if (v[i] <= v[j])
7             aux[k++] = v[i++];
8         else
9             aux[k++] = v[j++];
10    /*copia o resto do subvetor que não terminou*/
11    while (i <= m)
12        aux[k++] = v[i++];
13    while (j <= r)
14        aux[k++] = v[j++];
15    /*copia de volta para v*/
16    for (i = l, k = 0; i <= r; i++, k++)
17        v[i] = aux[k];
18 }
```

## Ordenação por intercalação (*MergeSort*)

```
1 void merge(int *v, int l, int m, int r) {
2     int aux[MAX];
3     int i = l, j = m + 1, k = 0;
4     /*intercala*/
5     while (i <= m && j <= r)
6         if (v[i] <= v[j])
7             aux[k++] = v[i++];
8         else
9             aux[k++] = v[j++];
10    /*copia o resto do subvetor que não terminou*/
11    while (i <= m)
12        aux[k++] = v[i++];
13    while (j <= r)
14        aux[k++] = v[j++];
15    /*copia de volta para v*/
16    for (i = l, k = 0; i <= r; i++, k++)
17        v[i] = aux[k];
18 }
```

Quantas comparações são feitas?

## Ordenação por intercalação (*MergeSort*)

```
1 void merge(int *v, int l, int m, int r) {
2     int aux[MAX];
3     int i = l, j = m + 1, k = 0;
4     /*intercala*/
5     while (i <= m && j <= r)
6         if (v[i] <= v[j])
7             aux[k++] = v[i++];
8         else
9             aux[k++] = v[j++];
10    /*copia o resto do subvetor que não terminou*/
11    while (i <= m)
12        aux[k++] = v[i++];
13    while (j <= r)
14        aux[k++] = v[j++];
15    /*copia de volta para v*/
16    for (i = l, k = 0; i <= r; i++, k++)
17        v[i] = aux[k];
18 }
```

Quantas comparações são feitas?

- a cada passo, aumentamos um em *i* ou em *j*

## Ordenação por intercalação (*MergeSort*)

```
1 void merge(int *v, int l, int m, int r) {
2     int aux[MAX];
3     int i = l, j = m + 1, k = 0;
4     /*intercala*/
5     while (i <= m && j <= r)
6         if (v[i] <= v[j])
7             aux[k++] = v[i++];
8         else
9             aux[k++] = v[j++];
10    /*copia o resto do subvetor que não terminou*/
11    while (i <= m)
12        aux[k++] = v[i++];
13    while (j <= r)
14        aux[k++] = v[j++];
15    /*copia de volta para v*/
16    for (i = l, k = 0; i <= r; i++, k++)
17        v[i] = aux[k];
18 }
```

Quantas comparações são feitas?

- a cada passo, aumentamos um em  $i$  ou em  $j$
- no máximo  $n := r - l + 1$

# Ordenação por intercalação (*MergeSort*)

Ordenação:

# Ordenação por intercalação (*MergeSort*)

Ordenação:

- Recebemos um **vetor** de tamanho  $n$  com limites:

# Ordenação por intercalação (*MergeSort*)

Ordenação:

- Recebemos um **vetor** de tamanho  $n$  com limites:
  - O vetor começa na posição **vetor[1]**



# Ordenação por intercalação (*MergeSort*)

Ordenação:

- Recebemos um **vetor** de tamanho  $n$  com limites:
  - O vetor começa na posição **vetor[l]**
  - O vetor termina na posição **vetor[r]**

# Ordenação por intercalação (*MergeSort*)

Ordenação:

- Recebemos um **vetor** de tamanho  $n$  com limites:
  - O vetor começa na posição **vetor**[1]
  - O vetor termina na posição **vetor**[r]
- Dividimos o vetor em dois subvetores de tamanho  $n/2$

# Ordenação por intercalação (*MergeSort*)

Ordenação:

- Recebemos um **vetor** de tamanho  $n$  com limites:
  - O vetor começa na posição **vetor**[ $l$ ]
  - O vetor termina na posição **vetor**[ $r$ ]
- Dividimos o vetor em dois subvetores de tamanho  $n/2$
- O caso base é um vetor de tamanho  $0$  ou  $1$

# Ordenação por intercalação (*MergeSort*)

Ordenação:

- Recebemos um **vetor** de tamanho  $n$  com limites:
  - O vetor começa na posição **vetor**[ $l$ ]
  - O vetor termina na posição **vetor**[ $r$ ]
- Dividimos o vetor em dois subvetores de tamanho  $n/2$
- O caso base é um vetor de tamanho  $0$  ou  $1$

# Ordenação por intercalação (*MergeSort*)

Ordenação:

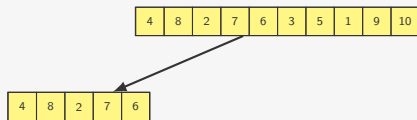
- Recebemos um **vetor** de tamanho  $n$  com limites:
  - O vetor começa na posição **vetor[1]**
  - O vetor termina na posição **vetor[r]**
- Dividimos o vetor em dois subvetores de tamanho  $n/2$
- O caso base é um vetor de tamanho **0** ou **1**

```
1 void mergesort(int *v, int l, int r) {
2     int m = (l + r) / 2;
3     if (l < r) {
4         /*divisão*/
5         mergesort(v, l, m);
6         mergesort(v, m + 1, r);
7         /*conquista*/
8         merge(v, l, m, r);
9     }
10 }
```

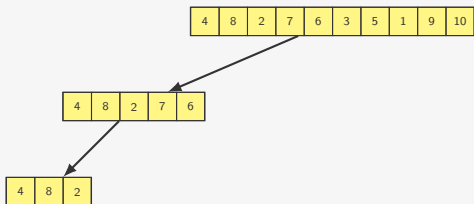
# Simulação

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

# Simulação

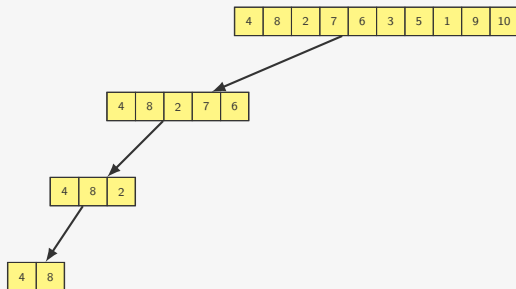


# Simulação

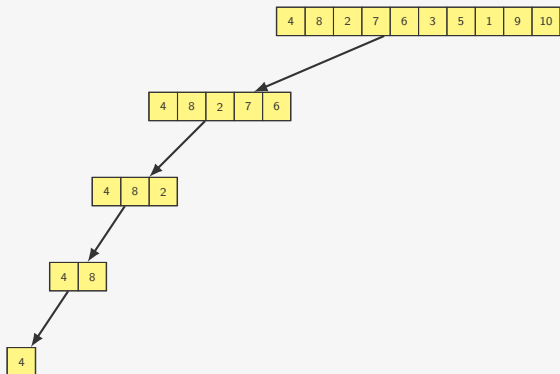




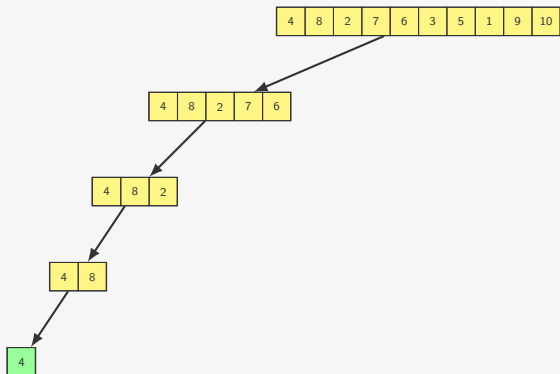
# Simulação



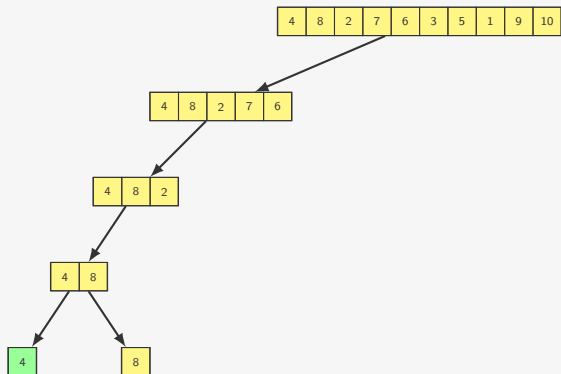
# Simulação



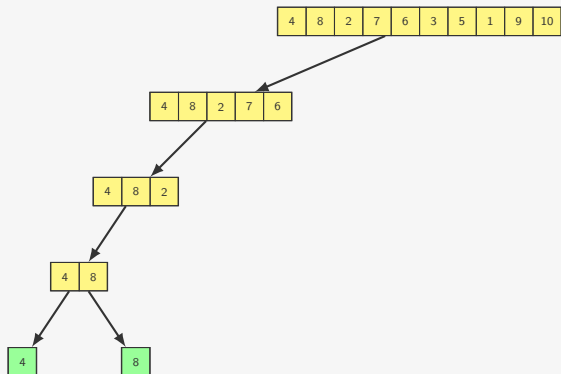
# Simulação



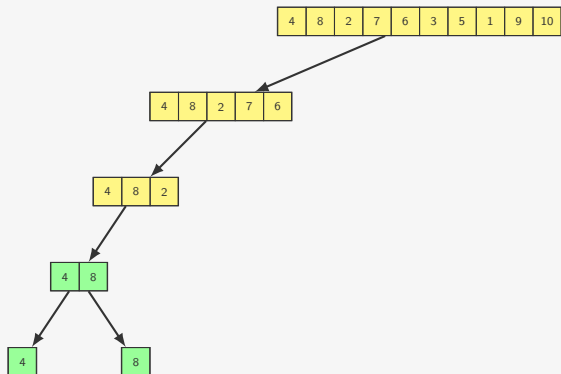
# Simulação



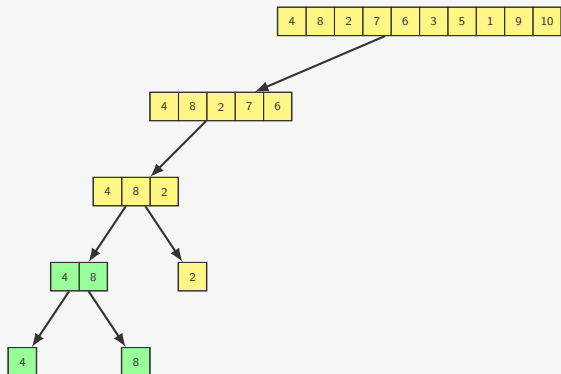
# Simulação



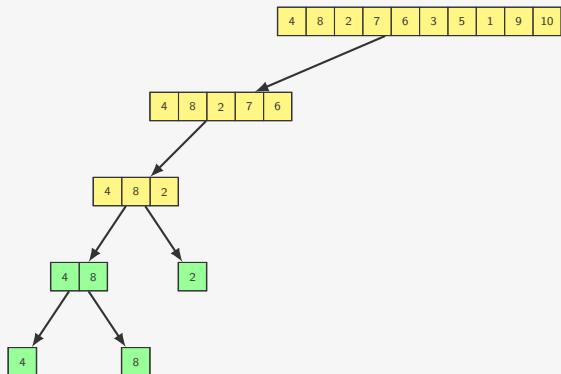
# Simulação



# Simulação

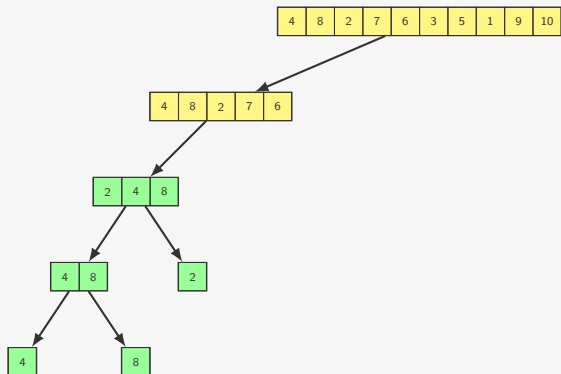


# Simulação

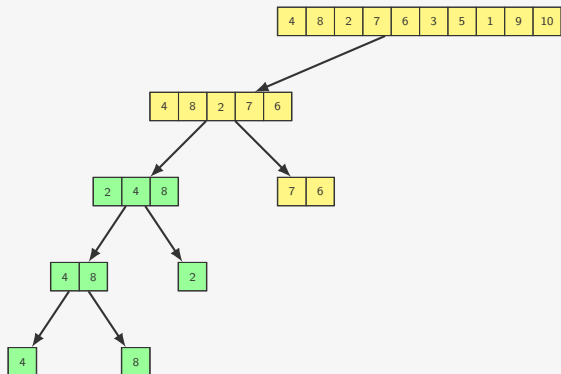




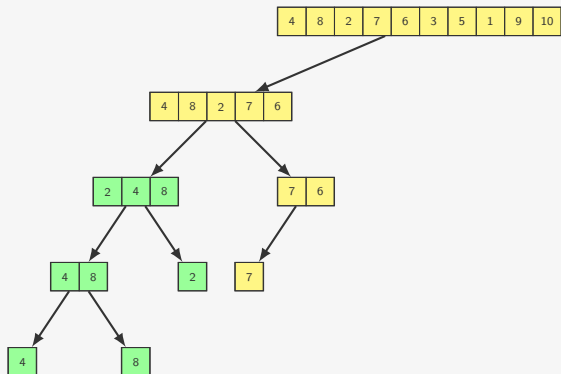
# Simulação



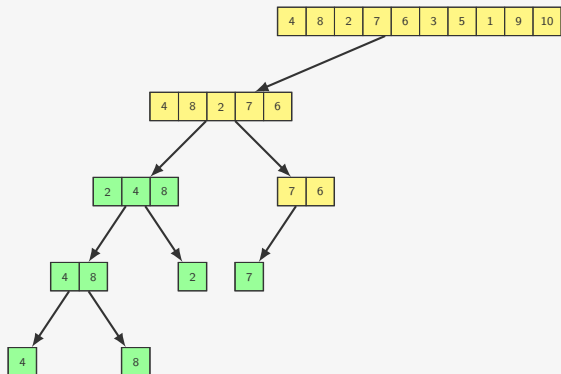
# Simulação



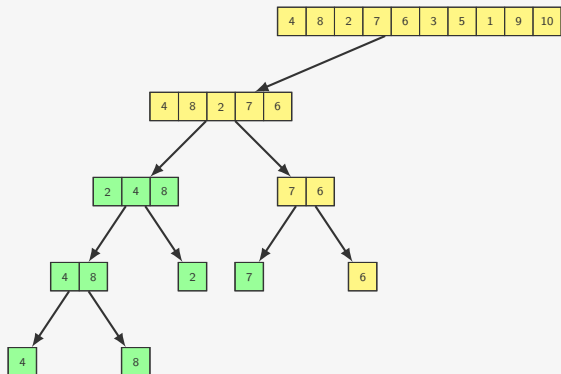
# Simulação



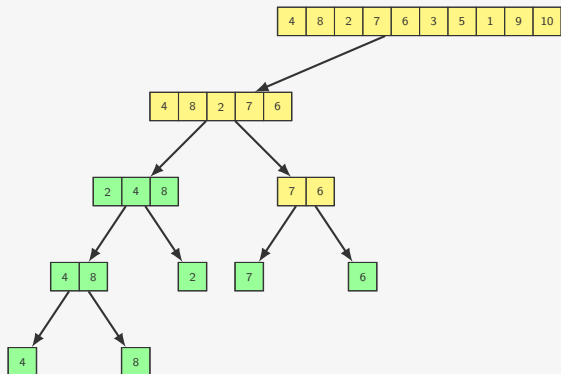
# Simulação



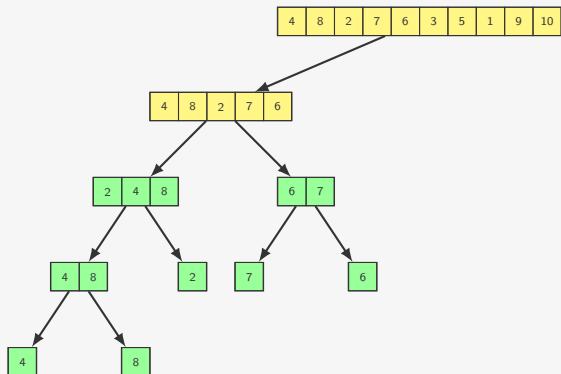
# Simulação



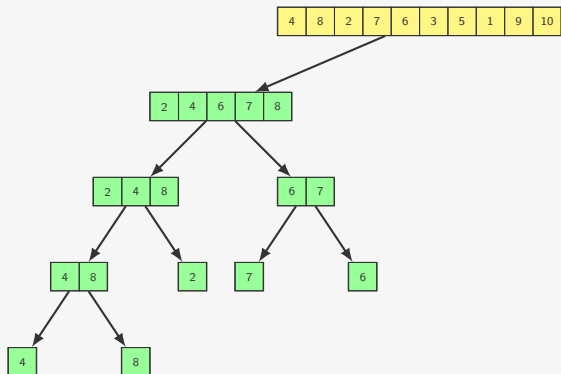
# Simulação



# Simulação

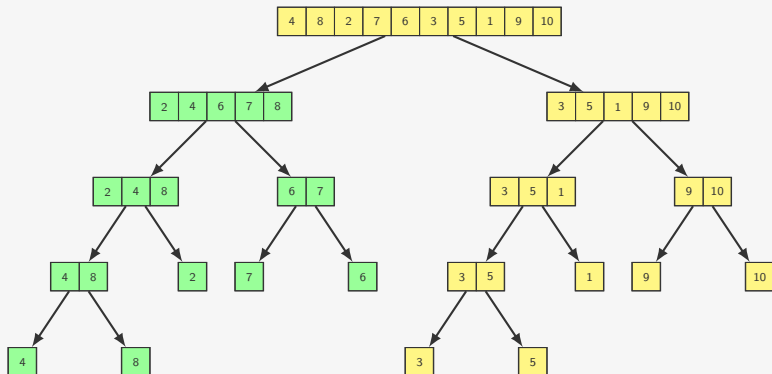


# Simulação

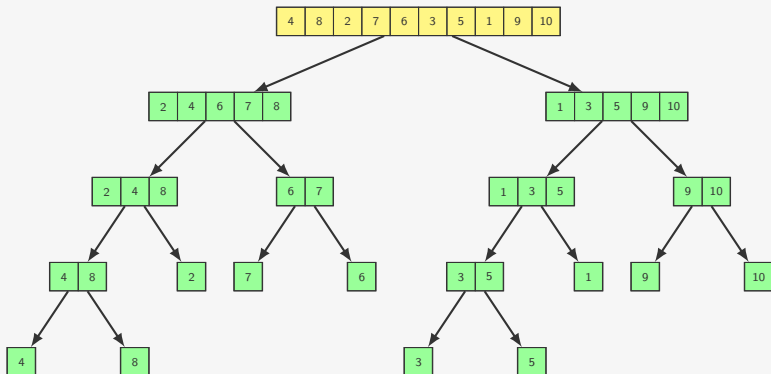




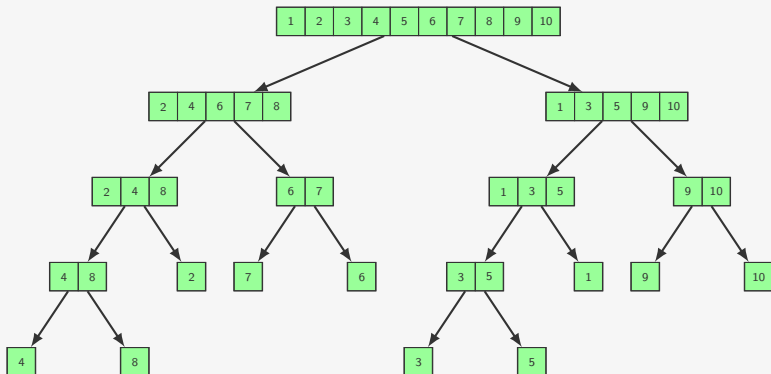
# Simulação



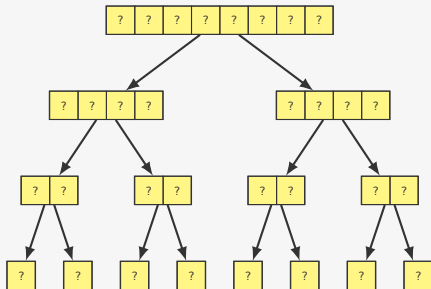
# Simulação



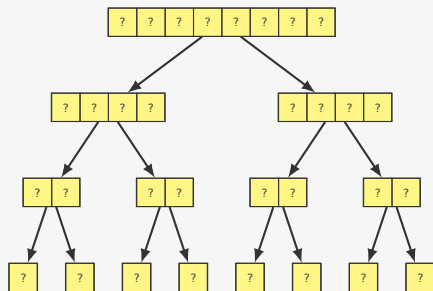
# Simulação



Tempo de execução para  $n = 2^l$



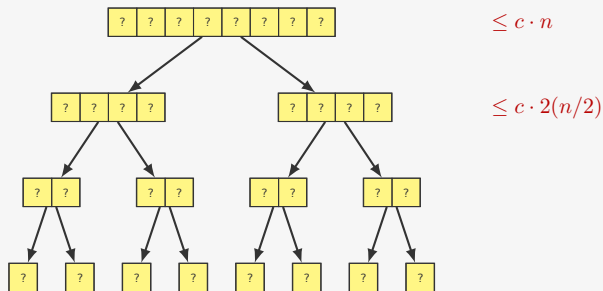
# Tempo de execução para $n = 2^l$



$$\leq c \cdot n$$

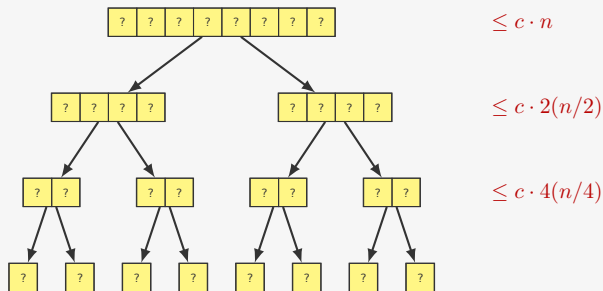
- No primeiro nível fazemos **um** merge com  $n$  elementos

## Tempo de execução para $n = 2^l$



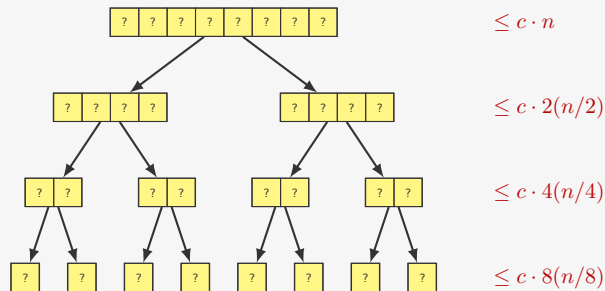
- No primeiro nível fazemos **um** merge com  $n$  elementos
- No segundo fazemos **dois** merge com  $n/2$  elementos

## Tempo de execução para $n = 2^l$



- No primeiro nível fazemos **um** merge com  $n$  elementos
- No segundo fazemos **dois** merge com  $n/2$  elementos
- No  $(k - 1)$ -ésimo fazemos  $2^k$  merge com  $n/2^k$  elementos

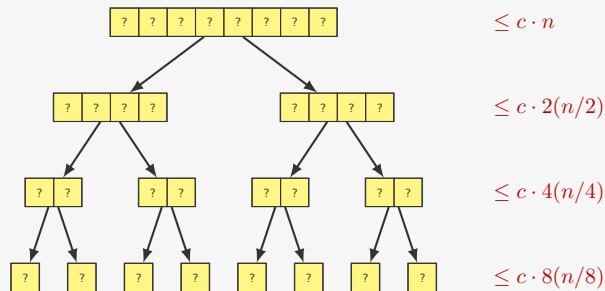
## Tempo de execução para $n = 2^l$



- No primeiro nível fazemos **um** merge com  $n$  elementos
- No segundo fazemos **dois** merge com  $n/2$  elementos
- No  $(k - 1)$ -ésimo fazemos  $2^k$  merge com  $n/2^k$  elementos
- No último gastamos tempo constante  $n$  vezes

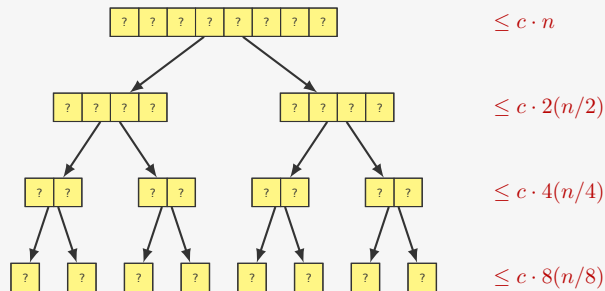


# Tempo de execução para $n = 2^l$



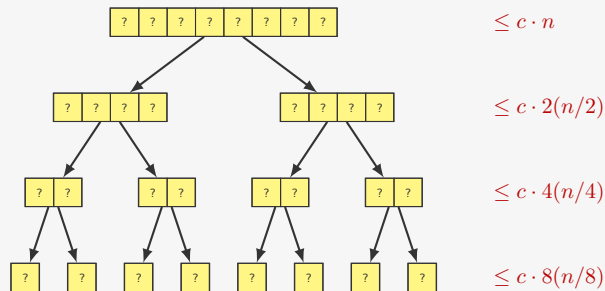
- No nível  $k$  gastamos tempo  $\leq c \cdot n$

# Tempo de execução para $n = 2^l$



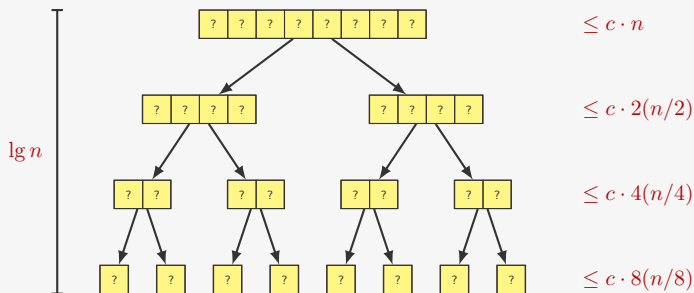
- No nível  $k$  gastamos tempo  $\leq c \cdot n$
- Quantos níveis temos?

## Tempo de execução para $n = 2^l$



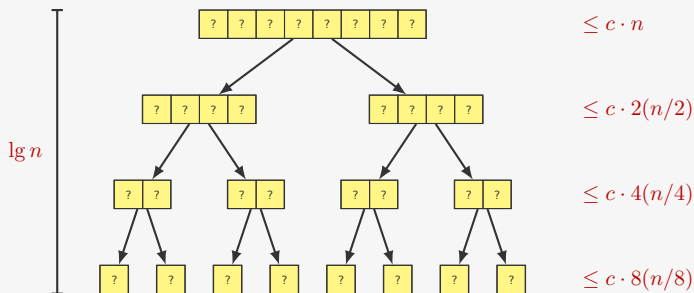
- No nível  $k$  gastamos tempo  $\leq c \cdot n$
- Quantos níveis temos?
  - Dividimos  $n$  por  $2$  até que fique menor ou igual a  $1$

# Tempo de execução para $n = 2^l$



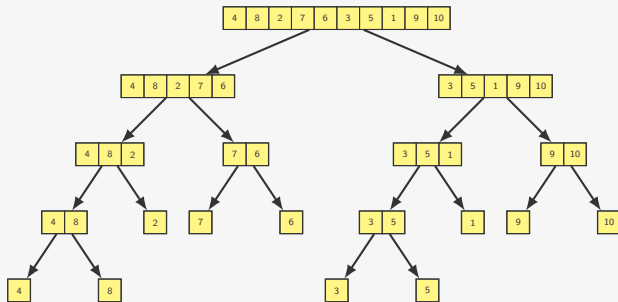
- No nível  $k$  gastamos tempo  $\leq c \cdot n$
- Quantos níveis temos?
  - Dividimos  $n$  por  $2$  até que fique menor ou igual a  $1$
  - Ou seja,  $l = \lg n$

## Tempo de execução para $n = 2^l$

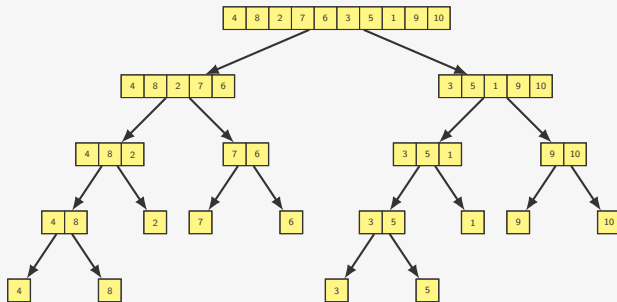


- No nível  $k$  gastamos tempo  $\leq c \cdot n$
- Quantos níveis temos?
  - Dividimos  $n$  por  $2$  até que fique menor ou igual a  $1$
  - Ou seja,  $l = \lg n$
- Tempo total:  $cn \lg n = O(n \lg n)$

# Tempo de execução para $n$ qualquer

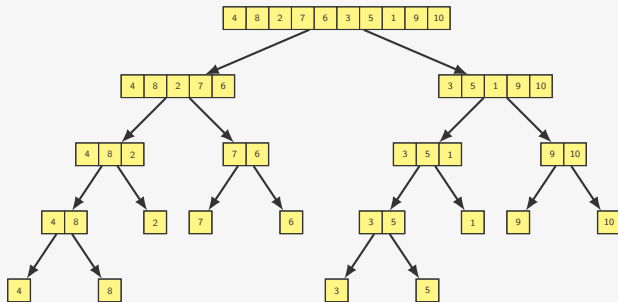


## Tempo de execução para $n$ qualquer



Qual o tempo de execução para  $n$  que não é potência de 2?

## Tempo de execução para $n$ qualquer

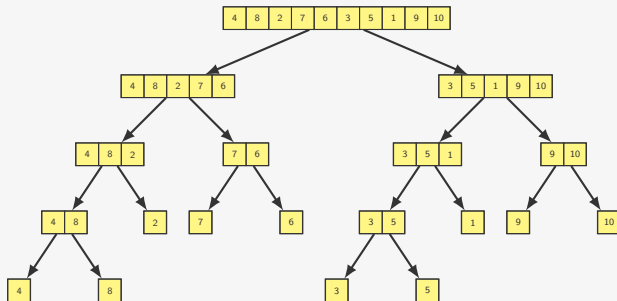


Qual o tempo de execução para  $n$  que não é potência de 2?

- Seja  $2^k$  a próxima potência de 2 depois de  $n$



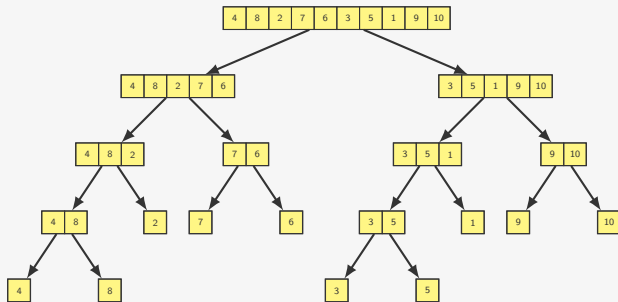
## Tempo de execução para $n$ qualquer



Qual o tempo de execução para  $n$  que não é potência de  $2$ ?

- Seja  $2^k$  a próxima potência de  $2$  depois de  $n$ 
  - Ex: Se  $n = 3000$ , a próxima potência é  $4096$

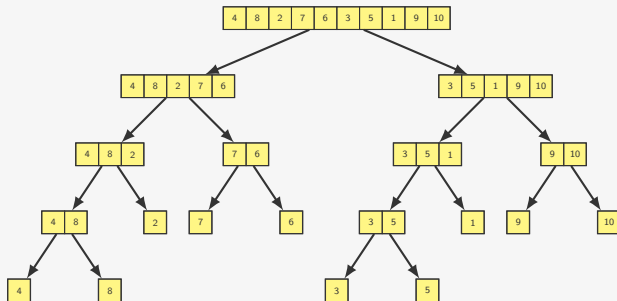
## Tempo de execução para $n$ qualquer



Qual o tempo de execução para  $n$  que não é potência de 2?

- Seja  $2^k$  a próxima potência de 2 depois de  $n$ 
  - Ex: Se  $n = 3000$ , a próxima potência é 4096
- Temos que  $2^{k-1} < n < 2^k$

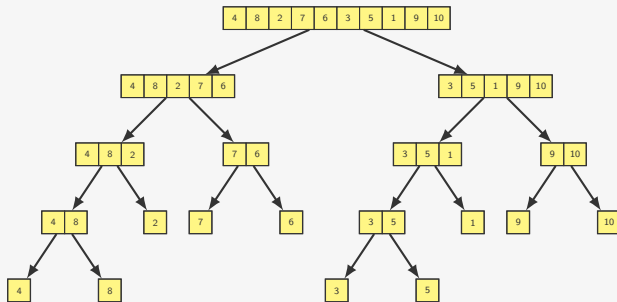
## Tempo de execução para $n$ qualquer



Qual o tempo de execução para  $n$  que não é potência de 2?

- Seja  $2^k$  a próxima potência de 2 depois de  $n$ 
  - Ex: Se  $n = 3000$ , a próxima potência é 4096
- Temos que  $2^{k-1} < n < 2^k$ 
  - Ou seja,  $2^k < 2n$

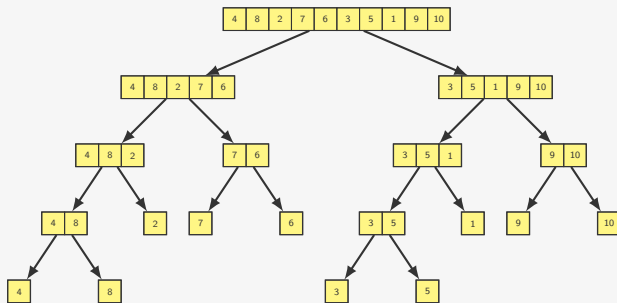
## Tempo de execução para $n$ qualquer



Qual o tempo de execução para  $n$  que não é potência de 2?

- Seja  $2^k$  a próxima potência de 2 depois de  $n$ 
  - Ex: Se  $n = 3000$ , a próxima potência é 4096
- Temos que  $2^{k-1} < n < 2^k$ 
  - Ou seja,  $2^k < 2n$
- O tempo de execução para  $n$  é menor do que

## Tempo de execução para $n$ qualquer

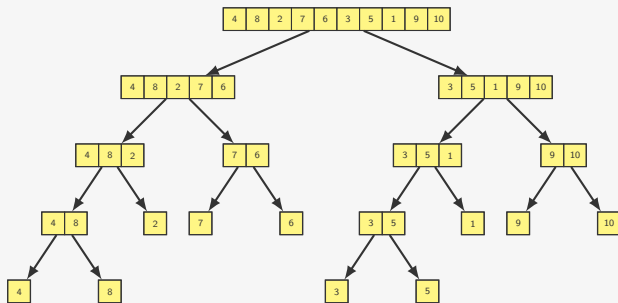


Qual o tempo de execução para  $n$  que não é potência de  $2$ ?

- Seja  $2^k$  a próxima potência de  $2$  depois de  $n$ 
  - Ex: Se  $n = 3000$ , a próxima potência é  $4096$
- Temos que  $2^{k-1} < n < 2^k$ 
  - Ou seja,  $2^k < 2n$
- O tempo de execução para  $n$  é menor do que

$$c 2^k \lg 2^k$$

## Tempo de execução para $n$ qualquer

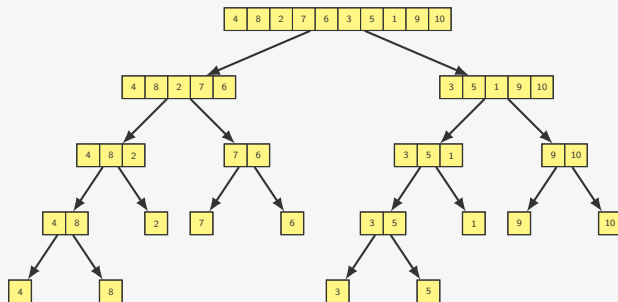


Qual o tempo de execução para  $n$  que não é potência de  $2$ ?

- Seja  $2^k$  a próxima potência de  $2$  depois de  $n$ 
  - Ex: Se  $n = 3000$ , a próxima potência é  $4096$
- Temos que  $2^{k-1} < n < 2^k$ 
  - Ou seja,  $2^k < 2n$
- O tempo de execução para  $n$  é menor do que

$$c 2^k \lg 2^k$$

## Tempo de execução para $n$ qualquer

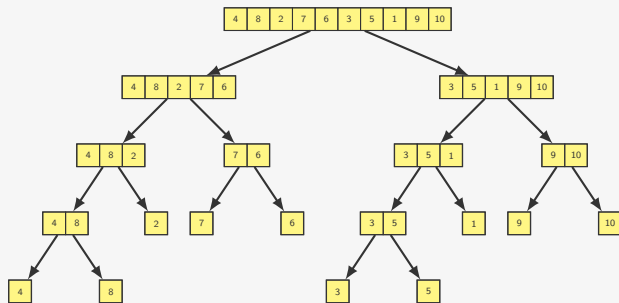


Qual o tempo de execução para  $n$  que não é potência de  $2$ ?

- Seja  $2^k$  a próxima potência de  $2$  depois de  $n$ 
  - Ex: Se  $n = 3000$ , a próxima potência é  $4096$
- Temos que  $2^{k-1} < n < 2^k$ 
  - Ou seja,  $2^k < 2n$
- O tempo de execução para  $n$  é menor do que

$$c2^k \lg 2^k \leq 2cn \lg(2n)$$

## Tempo de execução para $n$ qualquer



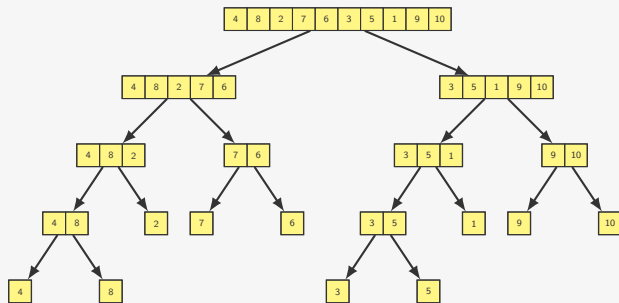
Qual o tempo de execução para  $n$  que não é potência de  $2$ ?

- Seja  $2^k$  a próxima potência de  $2$  depois de  $n$ 
  - Ex: Se  $n = 3000$ , a próxima potência é  $4096$
- Temos que  $2^{k-1} < n < 2^k$ 
  - Ou seja,  $2^k < 2n$
- O tempo de execução para  $n$  é menor do que

$$c2^k \lg 2^k \leq 2cn \lg(2n) = 2cn(\lg 2 + \lg n)$$



## Tempo de execução para $n$ qualquer

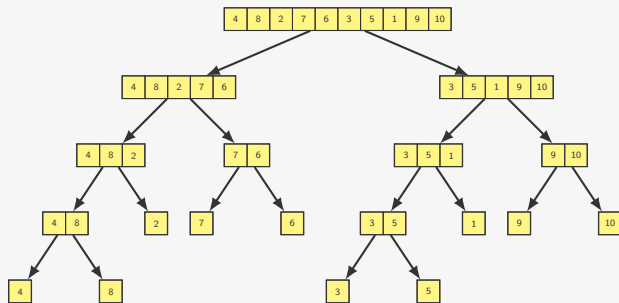


Qual o tempo de execução para  $n$  que não é potência de  $2$ ?

- Seja  $2^k$  a próxima potência de  $2$  depois de  $n$ 
  - Ex: Se  $n = 3000$ , a próxima potência é  $4096$
- Temos que  $2^{k-1} < n < 2^k$ 
  - Ou seja,  $2^k < 2n$
- O tempo de execução para  $n$  é menor do que

$$c2^k \lg 2^k \leq 2cn \lg(2n) = 2cn(\lg 2 + \lg n) = 2cn + 2cn \lg n$$

## Tempo de execução para $n$ qualquer

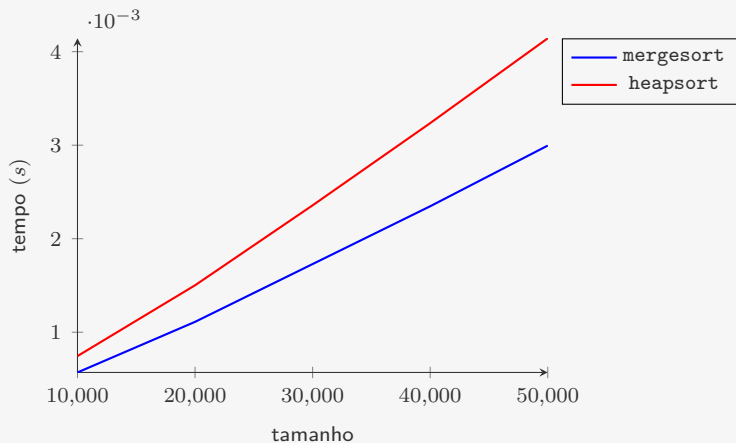


Qual o tempo de execução para  $n$  que não é potência de  $2$ ?

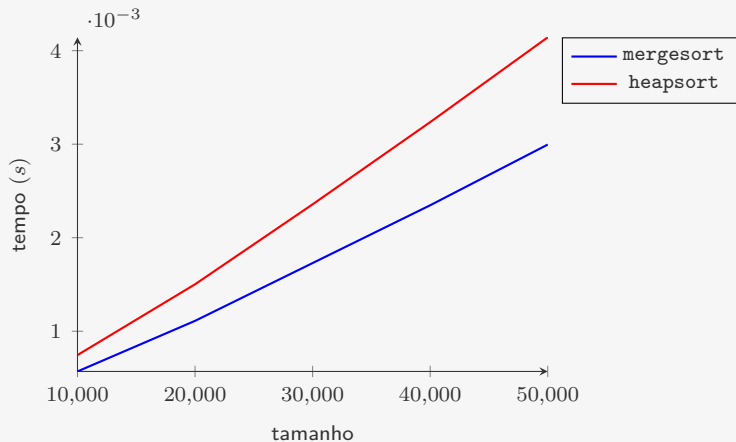
- Seja  $2^k$  a próxima potência de  $2$  depois de  $n$ 
  - Ex: Se  $n = 3000$ , a próxima potência é  $4096$
- Temos que  $2^{k-1} < n < 2^k$ 
  - Ou seja,  $2^k < 2n$
- O tempo de execução para  $n$  é menor do que

$$c 2^k \lg 2^k \leq 2cn \lg(2n) = 2cn(\lg 2 + \lg n) = 2cn + 2cn \lg n = O(n \lg n)$$

# Comparação entre mergesort e heapsort

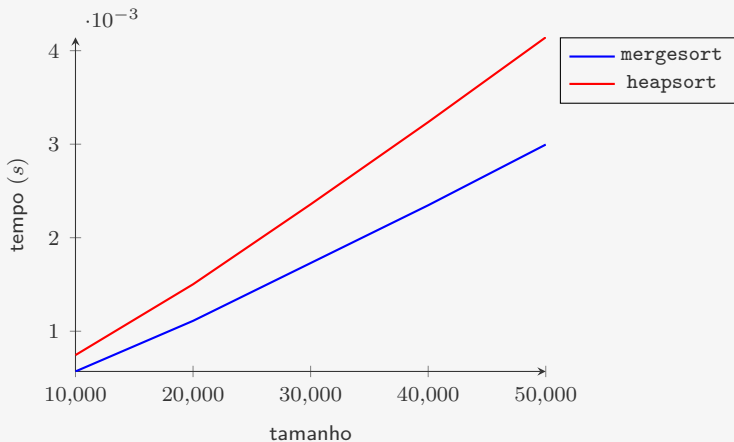


## Comparação entre mergesort e heapsort



**mergesort** é mais rápido do que o **heapsort**

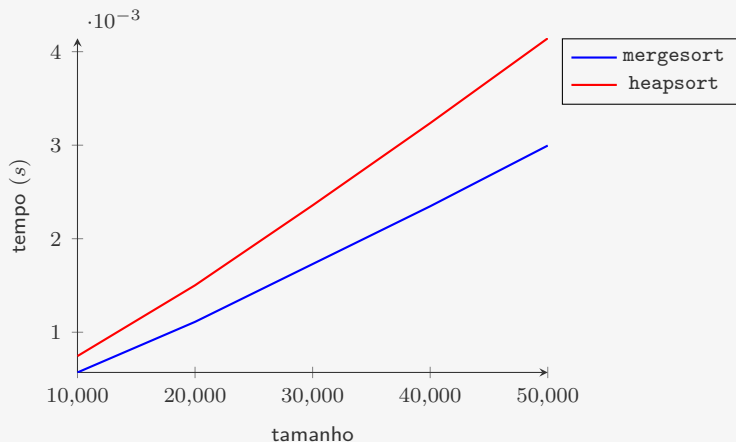
## Comparação entre mergesort e heapsort



**mergesort** é mais rápido do que o **heapsort**

- mas precisa de memória adicional

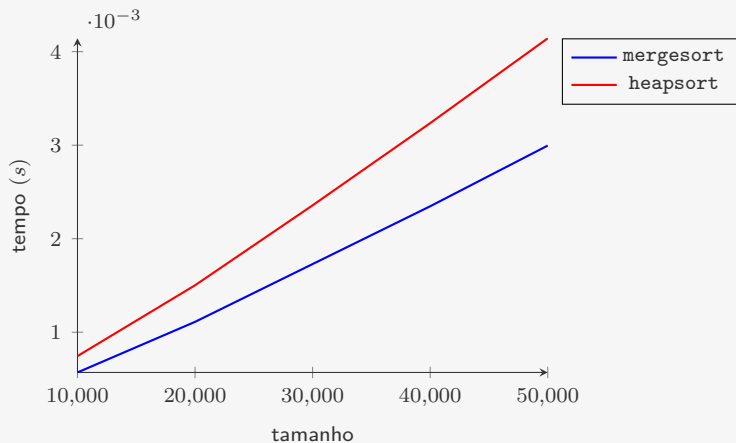
# Comparação entre mergesort e heapsort



**mergesort** é mais rápido do que o **heapsort**

- mas precisa de memória adicional
  - tanto para o vetor auxiliar -  $O(n)$

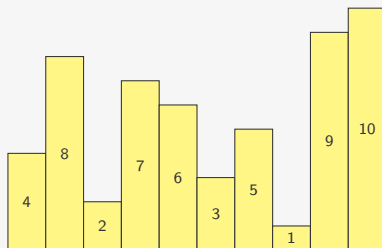
# Comparação entre mergesort e heapsort



**mergesort** é mais rápido do que o **heapsort**

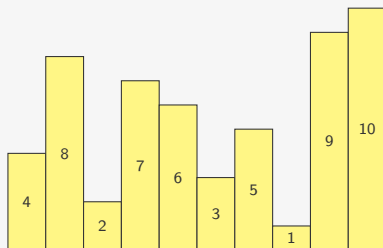
- mas precisa de memória adicional
  - tanto para o vetor auxiliar -  $O(n)$
  - quanto para a pilha de recursão -  $O(\lg n)$

# Quicksort - Ideia



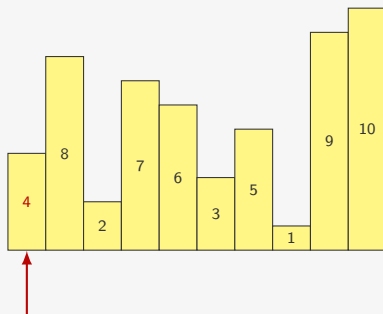


## Quicksort - Ideia



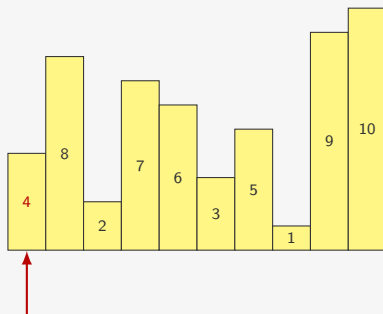
- Escolhemos um **pivô** (ex: 4)

## Quicksort - Ideia



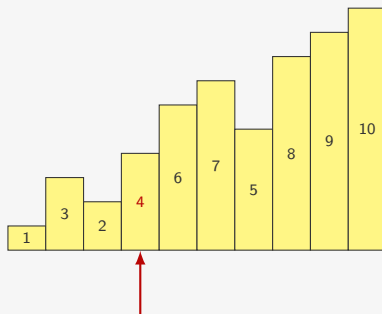
- Escolhemos um **pivô** (ex: 4)

## Quicksort - Ideia



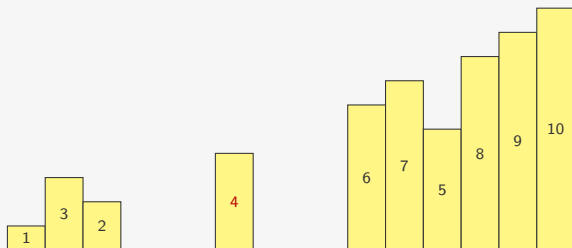
- Escolhemos um **pivô** (ex: 4)
- Colocamos
  - os elementos **menores** que o pivô **na esquerda**
  - os elementos **maiores** que o pivô **na direita**

# Quicksort - Ideia



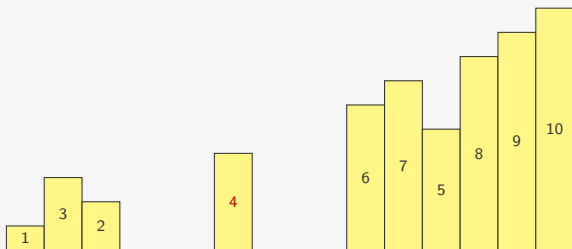
- Escolhemos um **pivô** (ex: 4)
- Colocamos
  - os elementos **menores** que o pivô **na esquerda**
  - os elementos **maiores** que o pivô **na direita**

# Quicksort - Ideia



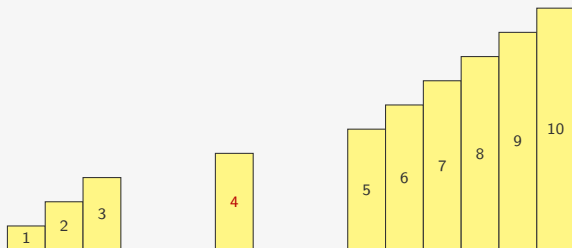
- Escolhemos um **pivô** (ex: 4)
- Colocamos
  - os elementos **menores** que o pivô **na esquerda**
  - os elementos **maiores** que o pivô **na direita**
- O **pivô** está na posição **correta**

# 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

```
1 int partition(int *v, int l, int r);
```



# Quicksort

```
1 int partition(int *v, int l, int r);
```

- escolhe um pivô

# Quicksort

```
1 int partition(int *v, int l, int r);
```

- escolhe um **pivô**
- coloca os elementos **menores à esquerda** do pivô

# Quicksort

```
1 int partition(int *v, int l, int r);
```

- escolhe um pivô
- coloca os elementos menores à esquerda do pivô
- coloca os elementos maiores à direita do pivô

# Quicksort

```
1 int partition(int *v, int l, int r);
```

- 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

```
1 int partition(int *v, int l, int r);
```

- 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 void quicksort(int *v, int l, int r) {  
2     int i;  
3     if (r <= l) return;  
4     i = partition(v, l, r);  
5     quicksort(v, l, i-1);  
6     quicksort(v, i+1, r);  
7 }
```

# Quicksort

```
1 int partition(int *v, int l, int r);
```

- 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 void quicksort(int *v, int l, int r) {  
2     int i;  
3     if (r <= l) return;  
4     i = partition(v, l, r);  
5     quicksort(v, l, i-1);  
6     quicksort(v, i+1, r);  
7 }
```

- Basta particionar o vetor em dois

# Quicksort

```
1 int partition(int *v, int l, int r);
```

- 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 void quicksort(int *v, int l, int r) {  
2     int i;  
3     if (r <= l) return;  
4     i = partition(v, l, r);  
5     quicksort(v, l, i-1);  
6     quicksort(v, i+1, r);  
7 }
```

- Basta particionar o vetor em dois
- e ordenar o lado esquerdo e o direito

## Como particionar um vetor?

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



## Como particionar um vetor?

- Andamos da direita para a esquerda com um índice  $i$
- De  $i$  até  $pos - 1$  ficam os menores do que o pivô

## Como particionar um vetor?

- 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  $r$  ficam os maiores ou iguais ao pivô

## Como particionar um vetor?

- 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  $r$  ficam os maiores ou iguais ao pivô
- Se o elemento em  $i$  for maior ou igual ao pivô

## Como particionar um vetor?

- 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  $r$  ficam os maiores ou iguais ao pivô
- Se o elemento em  $i$  for maior ou igual ao pivô
  - diminuimos  $pos$  e realizamos uma troca de  $i$  com  $pos$

## Como particionar um vetor?

- 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  $r$  ficam os maiores ou iguais ao pivô
- Se o elemento em  $i$  for maior ou igual ao pivô
  - diminuimos  $pos$  e realizamos uma troca de  $i$  com  $pos$
- No final, o pivô está em  $pos$

## Como particionar um vetor?

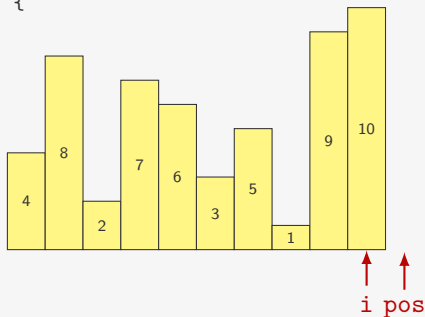
- Andamos da direita para a esquerda com um índice  $i$
- De  $i$  até  $pos - 1$  ficam os menores do que o pivô
- De  $pos$  até  $r$  ficam os maiores ou iguais ao pivô
- Se o elemento em  $i$  for maior ou igual ao pivô
  - diminuimos  $pos$  e realizamos uma troca de  $i$  com  $pos$
- No final,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

```
1 int partition(int *v, int l, int r) {
2     int i, pivo = v[l], pos = r + 1;
3     for (i = r; i >= l; i--) {
4         if (v[i] >= pivo) {
5             pos--;
6             troca(&v[i], &v[pos]);
7         }
8     }
9     return pos;
10 }
```

# Como particionar um vetor?

- 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  $r$  ficam os maiores ou iguais ao pivô
- Se o elemento em  $i$  for maior ou igual ao pivô
  - diminuimos  $pos$  e realizamos uma troca de  $i$  com  $pos$
- No final, o pivô está em  $pos$

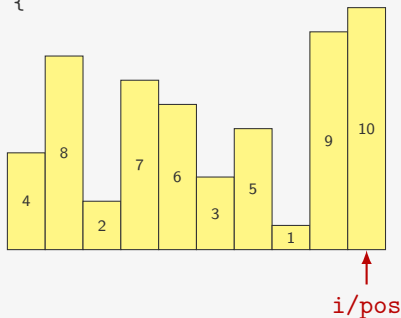
```
1 int partition(int *v, int l, int r) {
2     int i, pivo = v[l], pos = r + 1;
3     for (i = r; i >= l; i--) {
4         if (v[i] >= pivo) {
5             pos--;
6             troca(&v[i], &v[pos]);
7         }
8     }
9     return pos;
10 }
```



## Como particionar um vetor?

- 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  $r$  ficam os maiores ou iguais ao pivô
- Se o elemento em  $i$  for maior ou igual ao pivô
  - diminuimos  $pos$  e realizamos uma troca de  $i$  com  $pos$
- No final, o pivô está em  $pos$

```
1 int partition(int *v, int l, int r) {
2     int i, pivo = v[l], pos = r + 1;
3     for (i = r; i >= l; i--) {
4         if (v[i] >= pivo) {
5             pos--;
6             troca(&v[i], &v[pos]);
7         }
8     }
9     return pos;
10 }
```

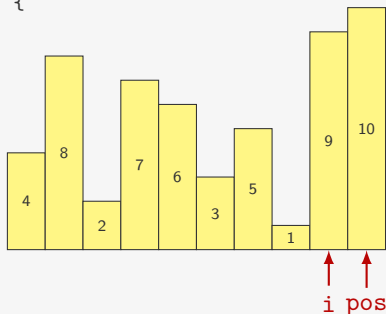




## Como particionar um vetor?

- 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  $r$  ficam os maiores ou iguais ao pivô
- Se o elemento em  $i$  for maior ou igual ao pivô
  - diminuimos  $pos$  e realizamos uma troca de  $i$  com  $pos$
- No final, o pivô está em  $pos$

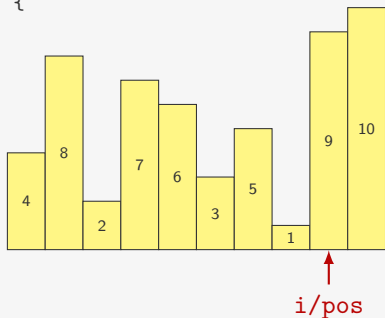
```
1 int partition(int *v, int l, int r) {
2   int i, pivo = v[l], pos = r + 1;
3   for (i = r; i >= l; i--) {
4     if (v[i] >= pivo) {
5       pos--;
6       troca(&v[i], &v[pos]);
7     }
8   }
9   return pos;
10 }
```



## Como particionar um vetor?

- 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  $r$  ficam os maiores ou iguais ao pivô
- Se o elemento em  $i$  for maior ou igual ao pivô
  - diminuimos  $pos$  e realizamos uma troca de  $i$  com  $pos$
- No final, o pivô está em  $pos$

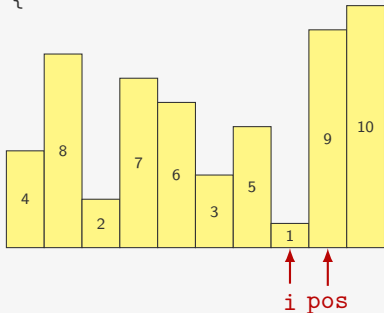
```
1 int partition(int *v, int l, int r) {
2     int i, pivo = v[l], pos = r + 1;
3     for (i = r; i >= l; i--) {
4         if (v[i] >= pivo) {
5             pos--;
6             troca(&v[i], &v[pos]);
7         }
8     }
9     return pos;
10 }
```



# Como particionar um vetor?

- 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  $r$  ficam os maiores ou iguais ao pivô
- Se o elemento em  $i$  for maior ou igual ao pivô
  - diminuimos  $pos$  e realizamos uma troca de  $i$  com  $pos$
- No final, o pivô está em  $pos$

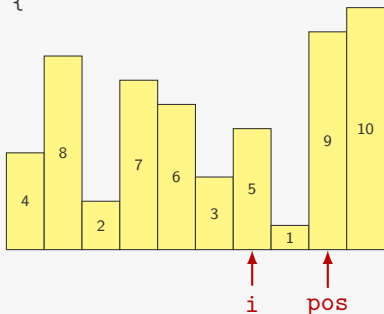
```
1 int partition(int *v, int l, int r) {
2     int i, pivo = v[l], pos = r + 1;
3     for (i = r; i >= l; i--) {
4         if (v[i] >= pivo) {
5             pos--;
6             troca(&v[i], &v[pos]);
7         }
8     }
9     return pos;
10 }
```



## Como particionar um vetor?

- 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  $r$  ficam os maiores ou iguais ao pivô
- Se o elemento em  $i$  for maior ou igual ao pivô
  - diminuimos  $pos$  e realizamos uma troca de  $i$  com  $pos$
- No final, o pivô está em  $pos$

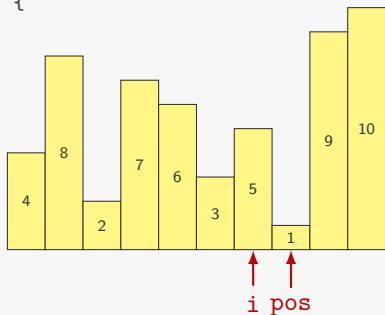
```
1 int partition(int *v, int l, int r) {
2   int i, pivo = v[l], pos = r + 1;
3   for (i = r; i >= l; i--) {
4     if (v[i] >= pivo) {
5       pos--;
6       troca(&v[i], &v[pos]);
7     }
8   }
9   return pos;
10 }
```



## Como particionar um vetor?

- 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  $r$  ficam os maiores ou iguais ao pivô
- Se o elemento em  $i$  for maior ou igual ao pivô
  - diminuimos  $pos$  e realizamos uma troca de  $i$  com  $pos$
- No final, o pivô está em  $pos$

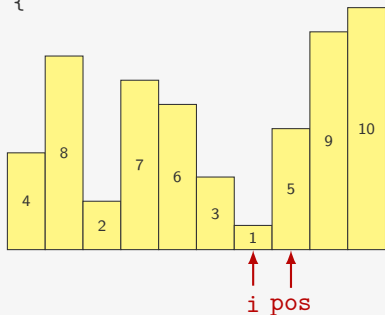
```
1 int partition(int *v, int l, int r) {
2     int i, pivo = v[l], pos = r + 1;
3     for (i = r; i >= l; i--) {
4         if (v[i] >= pivo) {
5             pos--;
6             troca(&v[i], &v[pos]);
7         }
8     }
9     return pos;
10 }
```



## Como particionar um vetor?

- 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  $r$  ficam os maiores ou iguais ao pivô
- Se o elemento em  $i$  for maior ou igual ao pivô
  - diminuimos  $pos$  e realizamos uma troca de  $i$  com  $pos$
- No final, o pivô está em  $pos$

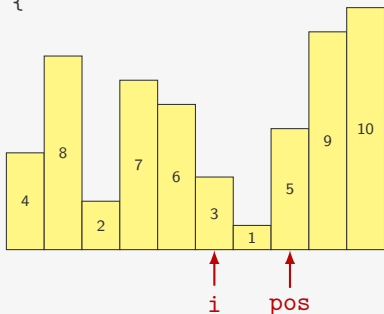
```
1 int partition(int *v, int l, int r) {
2   int i, pivo = v[l], pos = r + 1;
3   for (i = r; i >= l; i--) {
4     if (v[i] >= pivo) {
5       pos--;
6       troca(&v[i], &v[pos]);
7     }
8   }
9   return pos;
10 }
```



## Como particionar um vetor?

- 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  $r$  ficam os maiores ou iguais ao pivô
- Se o elemento em  $i$  for maior ou igual ao pivô
  - diminuimos  $pos$  e realizamos uma troca de  $i$  com  $pos$
- No final, o pivô está em  $pos$

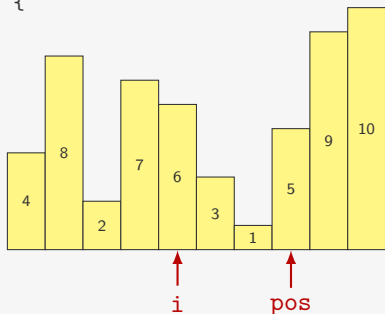
```
1 int partition(int *v, int l, int r) {
2     int i, pivo = v[l], pos = r + 1;
3     for (i = r; i >= l; i--) {
4         if (v[i] >= pivo) {
5             pos--;
6             troca(&v[i], &v[pos]);
7         }
8     }
9     return pos;
10 }
```



## Como particionar um vetor?

- 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  $r$  ficam os maiores ou iguais ao pivô
- Se o elemento em  $i$  for maior ou igual ao pivô
  - diminuimos  $pos$  e realizamos uma troca de  $i$  com  $pos$
- No final, o pivô está em  $pos$

```
1 int partition(int *v, int l, int r) {
2   int i, pivo = v[l], pos = r + 1;
3   for (i = r; i >= l; i--) {
4     if (v[i] >= pivo) {
5       pos--;
6       troca(&v[i], &v[pos]);
7     }
8   }
9   return pos;
10 }
```

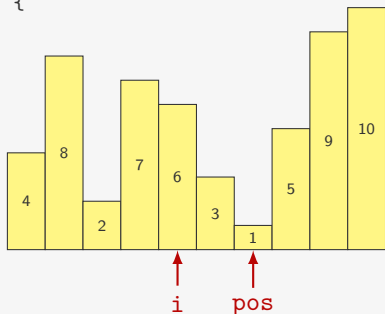




## Como particionar um vetor?

- 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  $r$  ficam os maiores ou iguais ao pivô
- Se o elemento em  $i$  for maior ou igual ao pivô
  - diminuimos  $\text{pos}$  e realizamos uma troca de  $i$  com  $\text{pos}$
- No final, o pivô está em  $\text{pos}$

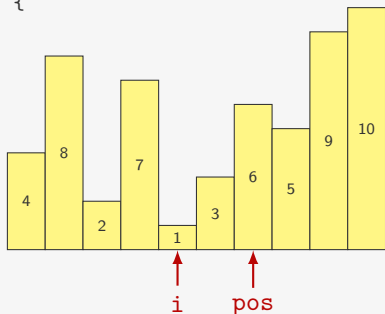
```
1 int partition(int *v, int l, int r) {
2   int i, pivo = v[l], pos = r + 1;
3   for (i = r; i >= l; i--) {
4     if (v[i] >= pivo) {
5       pos--;
6       troca(&v[i], &v[pos]);
7     }
8   }
9   return pos;
10 }
```



## Como particionar um vetor?

- 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  $r$  ficam os maiores ou iguais ao pivô
- Se o elemento em  $i$  for maior ou igual ao pivô
  - diminuimos  $pos$  e realizamos uma troca de  $i$  com  $pos$
- No final, o pivô está em  $pos$

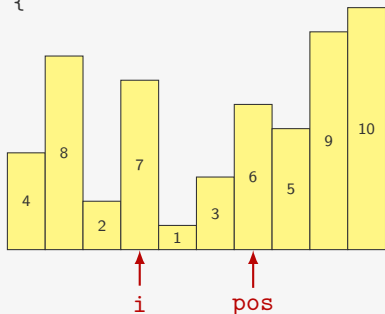
```
1 int partition(int *v, int l, int r) {
2     int i, pivo = v[l], pos = r + 1;
3     for (i = r; i >= l; i--) {
4         if (v[i] >= pivo) {
5             pos--;
6             troca(&v[i], &v[pos]);
7         }
8     }
9     return pos;
10 }
```



## Como particionar um vetor?

- 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  $r$  ficam os maiores ou iguais ao pivô
- Se o elemento em  $i$  for maior ou igual ao pivô
  - diminuimos  $pos$  e realizamos uma troca de  $i$  com  $pos$
- No final, o pivô está em  $pos$

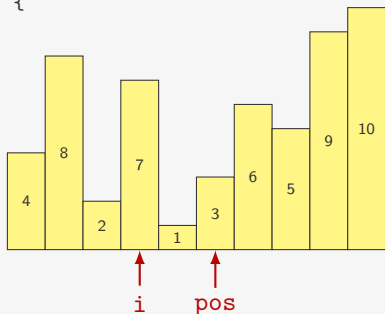
```
1 int partition(int *v, int l, int r) {
2     int i, pivo = v[l], pos = r + 1;
3     for (i = r; i >= l; i--) {
4         if (v[i] >= pivo) {
5             pos--;
6             troca(&v[i], &v[pos]);
7         }
8     }
9     return pos;
10 }
```



## Como particionar um vetor?

- 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  $r$  ficam os maiores ou iguais ao pivô
- Se o elemento em  $i$  for maior ou igual ao pivô
  - diminuimos  $pos$  e realizamos uma troca de  $i$  com  $pos$
- No final, o pivô está em  $pos$

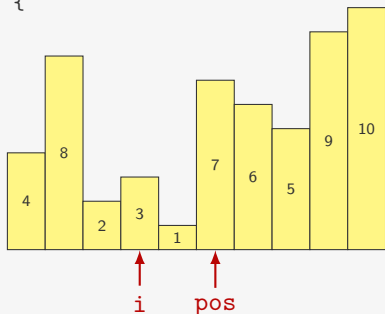
```
1 int partition(int *v, int l, int r) {
2   int i, pivo = v[l], pos = r + 1;
3   for (i = r; i >= l; i--) {
4     if (v[i] >= pivo) {
5       pos--;
6       troca(&v[i], &v[pos]);
7     }
8   }
9   return pos;
10 }
```



## Como particionar um vetor?

- 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  $r$  ficam os maiores ou iguais ao pivô
- Se o elemento em  $i$  for maior ou igual ao pivô
  - diminuimos  $pos$  e realizamos uma troca de  $i$  com  $pos$
- No final, o pivô está em  $pos$

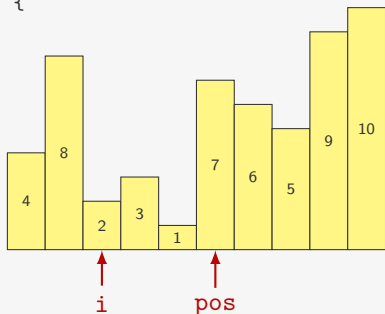
```
1 int partition(int *v, int l, int r) {
2     int i, pivo = v[l], pos = r + 1;
3     for (i = r; i >= l; i--) {
4         if (v[i] >= pivo) {
5             pos--;
6             troca(&v[i], &v[pos]);
7         }
8     }
9     return pos;
10 }
```



## Como particionar um vetor?

- 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  $r$  ficam os maiores ou iguais ao pivô
- Se o elemento em  $i$  for maior ou igual ao pivô
  - diminuimos  $pos$  e realizamos uma troca de  $i$  com  $pos$
- No final, o pivô está em  $pos$

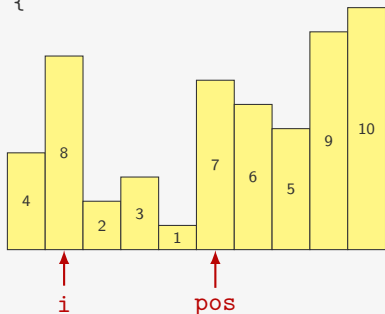
```
1 int partition(int *v, int l, int r) {
2   int i, pivo = v[l], pos = r + 1;
3   for (i = r; i >= l; i--) {
4     if (v[i] >= pivo) {
5       pos--;
6       troca(&v[i], &v[pos]);
7     }
8   }
9   return pos;
10 }
```



## Como particionar um vetor?

- 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  $r$  ficam os maiores ou iguais ao pivô
- Se o elemento em  $i$  for maior ou igual ao pivô
  - diminuimos  $pos$  e realizamos uma troca de  $i$  com  $pos$
- No final, o pivô está em  $pos$

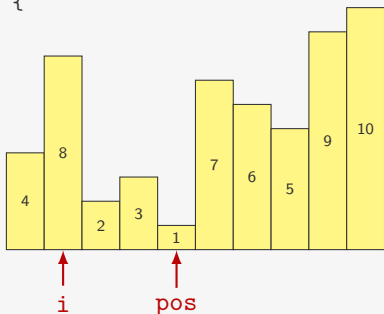
```
1 int partition(int *v, int l, int r) {
2     int i, pivo = v[l], pos = r + 1;
3     for (i = r; i >= l; i--) {
4         if (v[i] >= pivo) {
5             pos--;
6             troca(&v[i], &v[pos]);
7         }
8     }
9     return pos;
10 }
```



## Como particionar um vetor?

- 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  $r$  ficam os maiores ou iguais ao pivô
- Se o elemento em  $i$  for maior ou igual ao pivô
  - diminuimos  $pos$  e realizamos uma troca de  $i$  com  $pos$
- No final, o pivô está em  $pos$

```
1 int partition(int *v, int l, int r) {
2     int i, pivo = v[l], pos = r + 1;
3     for (i = r; i >= l; i--) {
4         if (v[i] >= pivo) {
5             pos--;
6             troca(&v[i], &v[pos]);
7         }
8     }
9     return pos;
10 }
```

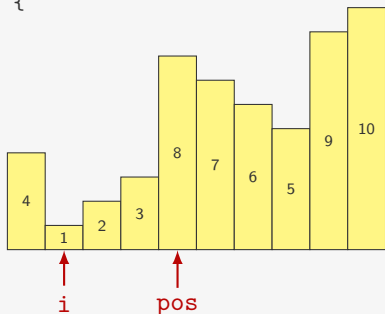




## Como particionar um vetor?

- 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  $r$  ficam os maiores ou iguais ao pivô
- Se o elemento em  $i$  for maior ou igual ao pivô
  - diminuimos  $pos$  e realizamos uma troca de  $i$  com  $pos$
- No final, o pivô está em  $pos$

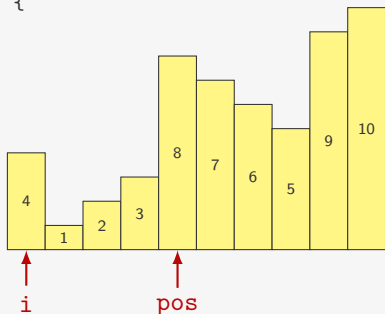
```
1 int partition(int *v, int l, int r) {
2     int i, pivo = v[l], pos = r + 1;
3     for (i = r; i >= l; i--) {
4         if (v[i] >= pivo) {
5             pos--;
6             troca(&v[i], &v[pos]);
7         }
8     }
9     return pos;
10 }
```



## Como particionar um vetor?

- 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  $r$  ficam os maiores ou iguais ao pivô
- Se o elemento em  $i$  for maior ou igual ao pivô
  - diminuimos  $pos$  e realizamos uma troca de  $i$  com  $pos$
- No final, o pivô está em  $pos$

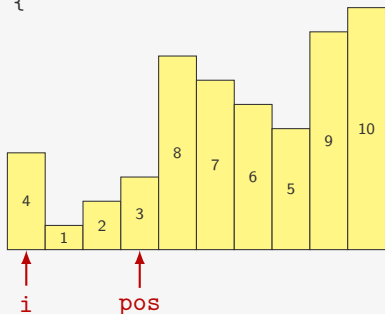
```
1 int partition(int *v, int l, int r) {
2     int i, pivo = v[l], pos = r + 1;
3     for (i = r; i >= l; i--) {
4         if (v[i] >= pivo) {
5             pos--;
6             troca(&v[i], &v[pos]);
7         }
8     }
9     return pos;
10 }
```



## Como particionar um vetor?

- 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  $r$  ficam os maiores ou iguais ao pivô
- Se o elemento em  $i$  for maior ou igual ao pivô
  - diminuimos  $pos$  e realizamos uma troca de  $i$  com  $pos$
- No final, o pivô está em  $pos$

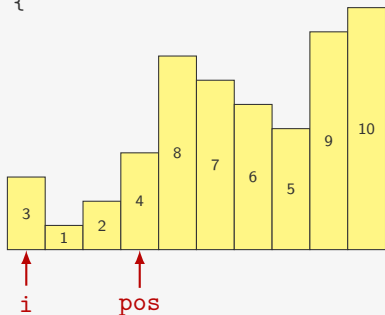
```
1 int partition(int *v, int l, int r) {
2     int i, pivo = v[l], pos = r + 1;
3     for (i = r; i >= l; i--) {
4         if (v[i] >= pivo) {
5             pos--;
6             troca(&v[i], &v[pos]);
7         }
8     }
9     return pos;
10 }
```



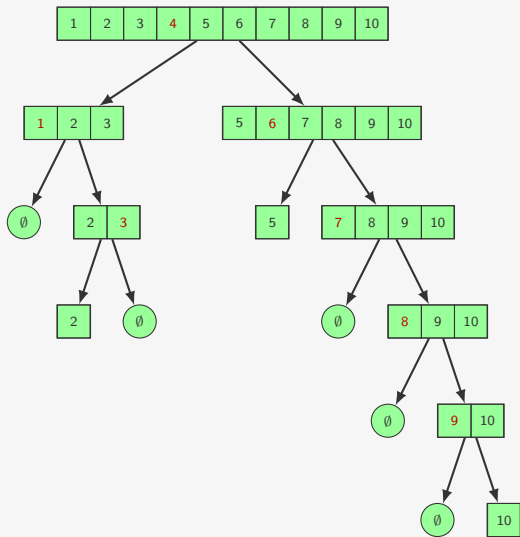
## Como particionar um vetor?

- 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  $r$  ficam os maiores ou iguais ao pivô
- Se o elemento em  $i$  for maior ou igual ao pivô
  - diminuimos  $pos$  e realizamos uma troca de  $i$  com  $pos$
- No final, o pivô está em  $pos$

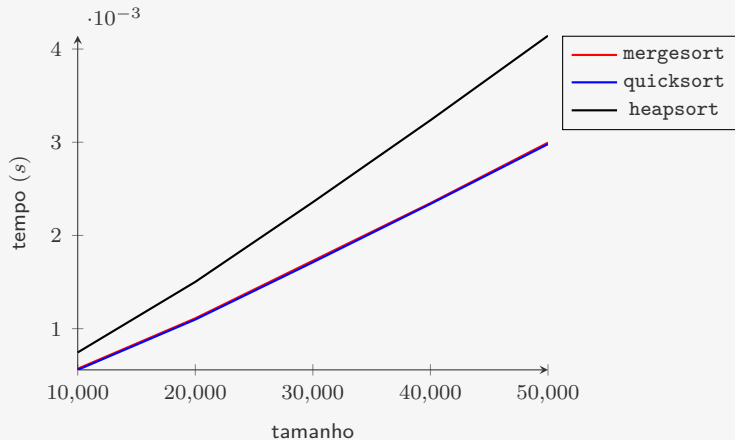
```
1 int partition(int *v, int l, int r) {
2     int i, pivo = v[l], pos = r + 1;
3     for (i = r; i >= l; i--) {
4         if (v[i] >= pivo) {
5             pos--;
6             troca(&v[i], &v[pos]);
7         }
8     }
9     return pos;
10 }
```



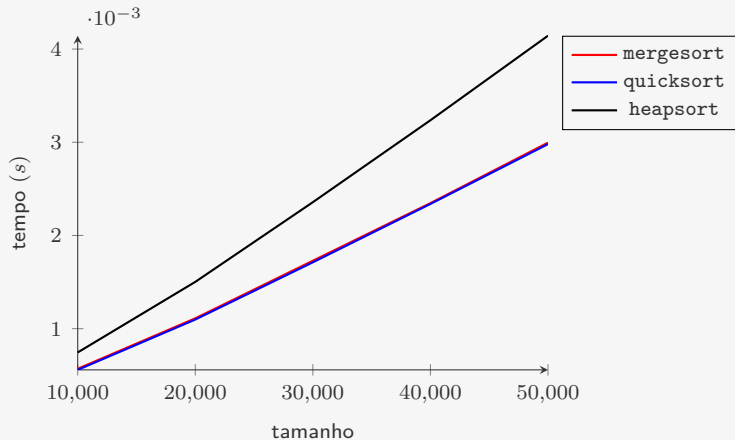
# Simulação do Quicksort



# Comparação com o mergesort e heapsort

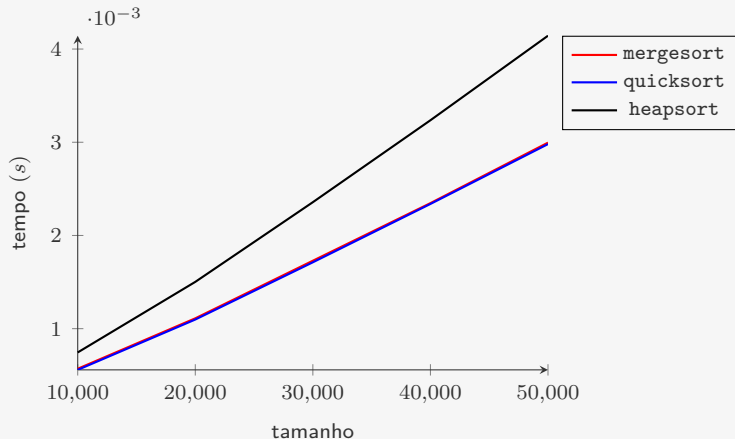


## Comparação com o mergesort e heapsort



O **quicksort** foi levemente mais rápido do que o **mergesort**

## Comparação com o mergesort e heapsort

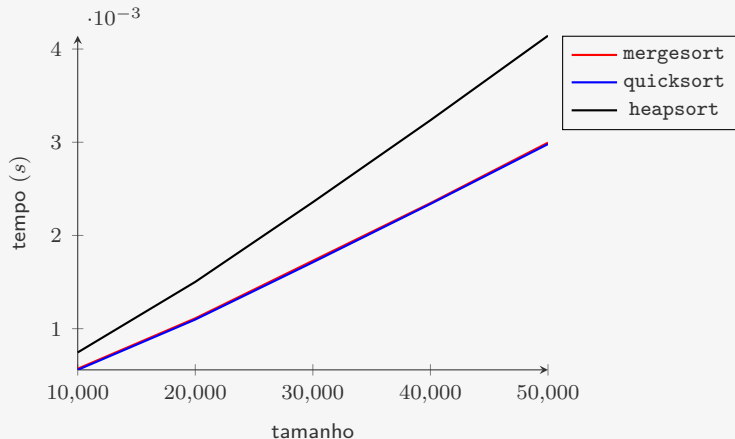


O **quicksort** foi levemente mais rápido do que o **mergesort**

- Mas ainda poderíamos otimizar o código dos três...



## Comparação com o mergesort e heapsort



O **quicksort** foi levemente mais rápido do que o **mergesort**

- Mas ainda poderíamos otimizar o código dos três...
- Ou seja, um poderia ficar melhor do que o outro

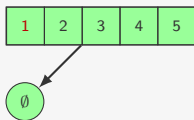
## Pior caso do QuickSort

1	2	3	4	5
---	---	---	---	---

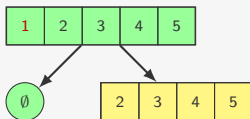
## Pior caso do QuickSort

1	2	3	4	5
---	---	---	---	---

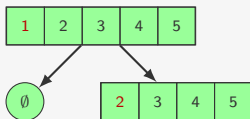
## Pior caso do QuickSort



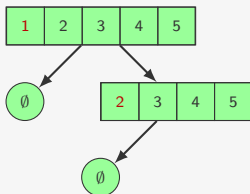
## Pior caso do QuickSort



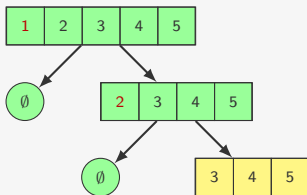
## Pior caso do QuickSort



## Pior caso do QuickSort

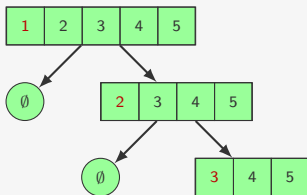


## Pior caso do QuickSort

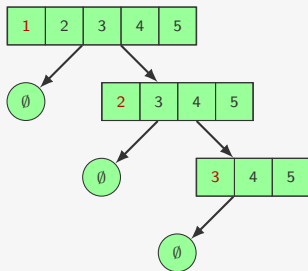




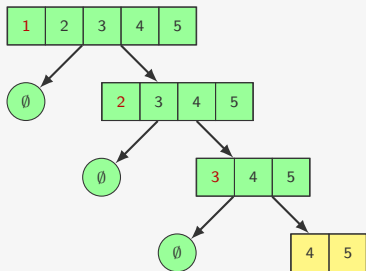
## Pior caso do QuickSort



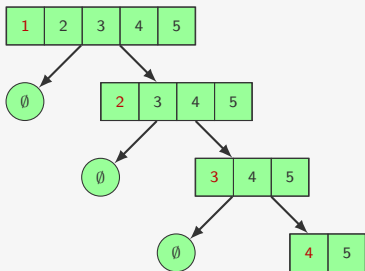
## Pior caso do QuickSort



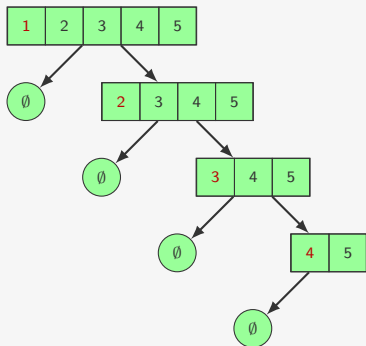
## Pior caso do QuickSort



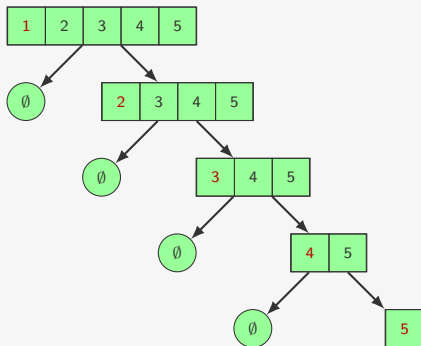
## Pior caso do QuickSort



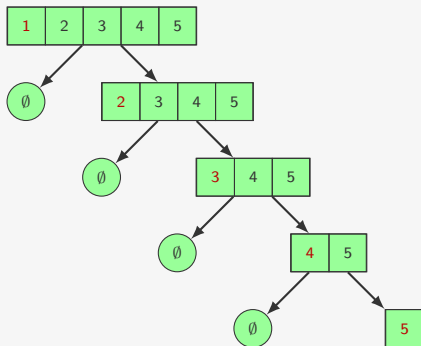
## Pior caso do QuickSort



## Pior caso do QuickSort

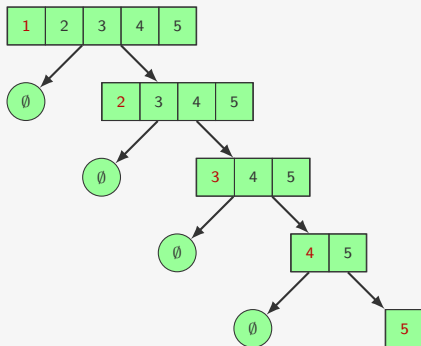


# Pior caso do QuickSort



$c \cdot n$

## Pior caso do QuickSort

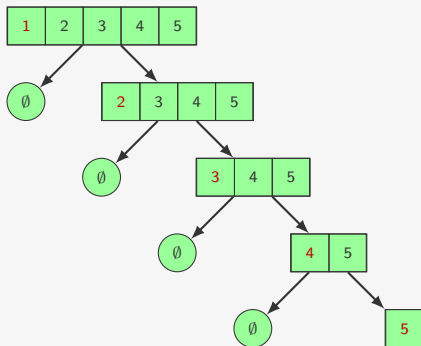


$$c \cdot n$$

$$c \cdot (n - 1)$$



## Pior caso do QuickSort

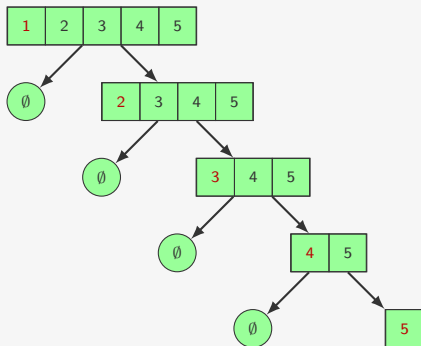


$$c \cdot n$$

$$c \cdot (n - 1)$$

$$c \cdot (n - 2)$$

## Pior caso do QuickSort



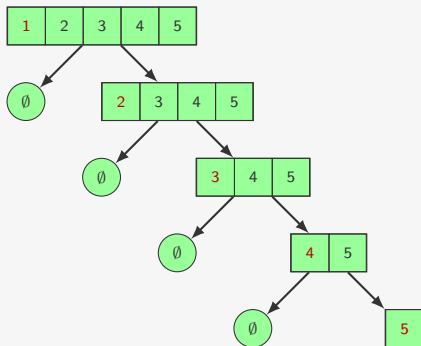
$$c \cdot n$$

$$c \cdot (n - 1)$$

$$c \cdot (n - 2)$$

...

## Pior caso do QuickSort



$$c \cdot n$$

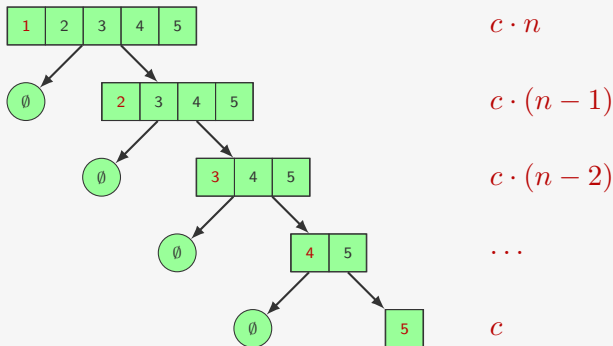
$$c \cdot (n - 1)$$

$$c \cdot (n - 2)$$

...

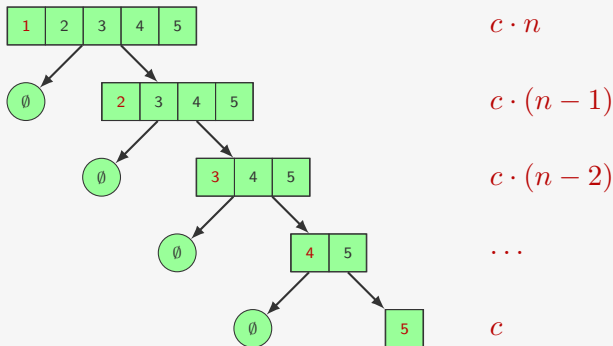
$$c$$

## Pior caso do QuickSort



O tempo de execução do Quicksort é, no pior caso:

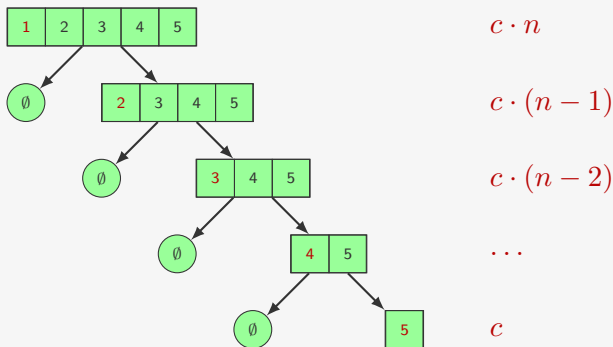
## Pior caso do QuickSort



O tempo de execução do Quicksort é, no pior caso:

$$c \cdot n + c \cdot (n - 1) + \dots + c$$

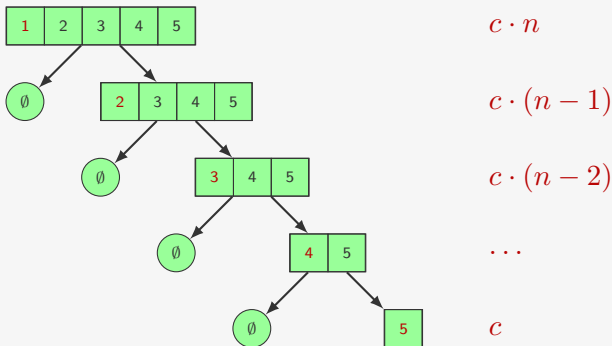
## Pior caso do QuickSort



O tempo de execução do Quicksort é, no pior caso:

$$c \cdot n + c \cdot (n - 1) + \dots + c = c \sum_{i=0}^{n-1} (n - i)$$

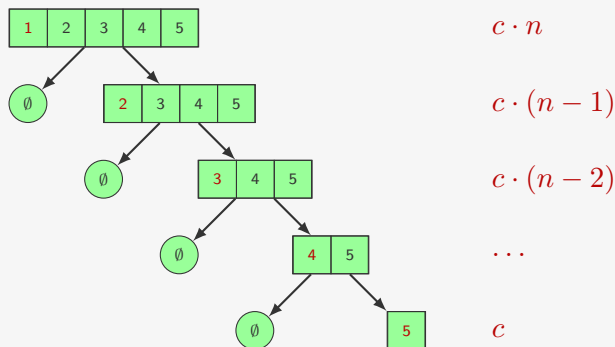
## Pior caso do QuickSort



O tempo de execução do Quicksort é, no pior caso:

$$c \cdot n + c \cdot (n-1) + \dots + c = c \sum_{i=0}^{n-1} (n-i) = c \sum_{j=1}^n j$$

## Pior caso do QuickSort

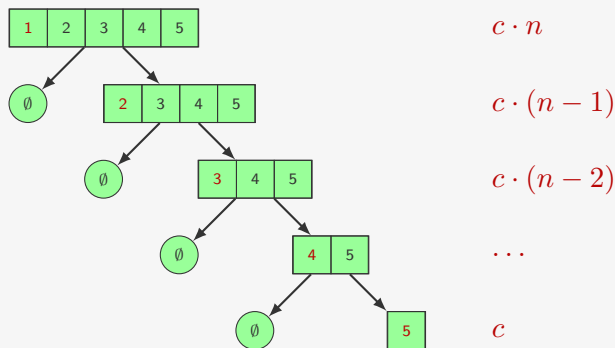


O tempo de execução do Quicksort é, no pior caso:

$$c \cdot n + c \cdot (n - 1) + \dots + c = c \sum_{i=0}^{n-1} (n - i) = c \sum_{j=1}^n j = c \frac{n(n + 1)}{2}$$



## Pior caso do QuickSort



O tempo de execução do Quicksort é, no pior caso:

$$c \cdot n + c \cdot (n - 1) + \dots + c = c \sum_{i=0}^{n-1} (n - i) = c \sum_{j=1}^n j = c \frac{n(n + 1)}{2} = O(n^2)$$

## Caso médio do QuickSort

Se o QuickSort é  $O(n^2)$ , como ele foi melhor que o HeapSort no experimento?

## Caso médio do QuickSort

Se o QuickSort é  $O(n^2)$ , como ele foi melhor que o HeapSort no experimento?

- Se o vetor for uma permutação aleatória de  $n$  números

## Caso médio do QuickSort

Se o QuickSort é  $O(n^2)$ , como ele foi melhor que o HeapSort no experimento?

- Se o vetor for uma permutação aleatória de  $n$  números
- então o tempo médio (esperado) do QuickSort é  $O(n \lg n)$

## Caso médio do QuickSort

Se o QuickSort é  $O(n^2)$ , como ele foi melhor que o HeapSort no experimento?

- Se o vetor for uma permutação aleatória de  $n$  números
- então o tempo médio (esperado) do QuickSort é  $O(n \lg n)$ 
  - Nesse caso, o pivô particiona bem o vetor

## Caso médio do QuickSort

Se o QuickSort é  $O(n^2)$ , como ele foi melhor que o HeapSort no experimento?

- Se o vetor for uma permutação aleatória de  $n$  números
- então o tempo médio (esperado) do QuickSort é  $O(n \lg n)$ 
  - Nesse caso, o pivô particiona bem o vetor

Ou seja, o pior caso do QuickSort é “raro” nesse experimento

## Caso médio do QuickSort

Se o QuickSort é  $O(n^2)$ , como ele foi melhor que o HeapSort no experimento?

- Se o vetor for uma permutação aleatória de  $n$  números
- então o tempo médio (esperado) do QuickSort é  $O(n \lg n)$ 
  - Nesse caso, o pivô particiona bem o vetor

Ou seja, o pior caso do QuickSort é “raro” nesse experimento

- Isso nem sempre é verdade

## Caso médio do QuickSort

Se o QuickSort é  $O(n^2)$ , como ele foi melhor que o HeapSort no experimento?

- Se o vetor for uma permutação aleatória de  $n$  números
- então o tempo médio (esperado) do QuickSort é  $O(n \lg n)$ 
  - Nesse caso, o pivô particiona bem o vetor

Ou seja, o pior caso do QuickSort é “raro” nesse experimento

- Isso nem sempre é verdade
  - as vezes, os dados estão parcialmente ordenados



## Caso médio do QuickSort

Se o QuickSort é  $O(n^2)$ , como ele foi melhor que o HeapSort no experimento?

- Se o vetor for uma permutação aleatória de  $n$  números
- então o tempo médio (esperado) do QuickSort é  $O(n \lg n)$ 
  - Nesse caso, o pivô particiona bem o vetor

Ou seja, o pior caso do QuickSort é “raro” nesse experimento

- Isso nem sempre é verdade
  - as vezes, os dados estão parcialmente ordenados
  - exemplo: inserção em blocos em um vetor ordenado

## Caso médio do QuickSort

Se o QuickSort é  $O(n^2)$ , como ele foi melhor que o HeapSort no experimento?

- Se o vetor for uma permutação aleatória de  $n$  números
- então o tempo médio (esperado) do QuickSort é  $O(n \lg n)$ 
  - Nesse caso, o pivô particiona bem o vetor

Ou seja, o pior caso do QuickSort é “raro” nesse experimento

- Isso nem sempre é verdade
  - as vezes, os dados estão parcialmente ordenados
  - exemplo: inserção em blocos em um vetor ordenado

Vamos ver duas formas de mitigar esse problema

## Mediana de Três

No `quicksort` escolhemos como pivô o elemento da esquerda

## Mediana de Três

No `quicksort` escolhemos como pivô o elemento da esquerda

- Poderíamos escolher o elemento da direita ou do meio

## Mediana de Três

No `quicksort` escolhemos como pivô o elemento da esquerda

- Poderíamos escolher o elemento da direita ou do meio
- Melhor ainda, podemos escolher a mediana dos três

## Mediana de Três

No `quicksort` escolhemos como pivô o elemento da esquerda

- Poderíamos escolher o elemento da direita ou do meio
- Melhor ainda, podemos escolher a mediana dos três
  - já que a mediana do vetor particiona ele no meio

# Mediana de Três

No `quicksort` escolhemos como pivô o elemento da esquerda

- Poderíamos escolher o elemento da direita ou do meio
- Melhor ainda, podemos escolher a mediana dos três
  - já que a mediana do vetor particiona ele no meio

```
1 void quicksort_mdt(int *v, int l, int r) {
2     int i;
3     if(r <= l) return;
4     troca(&v[(l+r)/2], &v[l+1]);
5     if(v[l] > v[l+1])
6         troca(&v[l], &v[l+1]);
7     if(v[l] > v[r])
8         troca(&v[l], &v[r]);
9     if(v[l+1] > v[r])
10        troca(&v[l+1], &v[r]);
11    i = partition(v, l+1, r-1);
12    quicksort_mdt(v, l, i-1);
13    quicksort_mdt(v, i+1, r);
14 }
```

- trocamos `v[(l+r)/2]` com `v[l+1]`
- ordenamos `v[l]`, `v[l+1]` e `v[r]`
- particionamos `v[l+1], ..., v[r-1]`
  - `v[l]` já é menor que o pivô
  - `v[r]` já é maior que o pivô

# Quicksort Aleatorizado



# Quicksort Aleatorizado

```
1 int pivo_aleatorio(int l, int r) {
2     return l + (int)((r-l+1)*(rand() / ((double)RAND_MAX + 1)));
3 }
4
5 void quicksort_ale(int *v, int l, int r) {
6     int i;
7     if(r <= l) return;
8     troca(&v[pivo_aleatorio(l,r)], &v[l]);
9     i = partition(v, l, r);
10    quicksort_ale(v, l, i-1);
11    quicksort_ale(v, i+1, r);
12 }
```

# Quicksort Aleatorizado

```
1 int pivo_aleatorio(int l, int r) {
2     return l + (int)((r-l+1)*(rand() / ((double)RAND_MAX + 1)));
3 }
4
5 void quicksort_ale(int *v, int l, int r) {
6     int i;
7     if(r <= l) return;
8     troca(&v[pivo_aleatorio(l,r)], &v[l]);
9     i = partition(v, l, r);
10    quicksort_ale(v, l, i-1);
11    quicksort_ale(v, i+1, r);
12 }
```

O tempo de execução depende dos pivôs sorteados

# Quicksort Aleatorizado

```
1 int pivo_aleatorio(int l, int r) {
2     return l + (int)((r-l+1)*(rand() / ((double)RAND_MAX + 1)));
3 }
4
5 void quicksort_ale(int *v, int l, int r) {
6     int i;
7     if(r <= l) return;
8     troca(&v[pivo_aleatorio(l,r)], &v[l]);
9     i = partition(v, l, r);
10    quicksort_ale(v, l, i-1);
11    quicksort_ale(v, i+1, r);
12 }
```

O tempo de execução depende dos pivôs sorteados

- O tempo médio é  $O(n \lg n)$

# Quicksort Aleatorizado

```
1 int pivo_aleatorio(int l, int r) {
2     return l + (int)((r-l+1)*(rand() / ((double)RAND_MAX + 1)));
3 }
4
5 void quicksort_ale(int *v, int l, int r) {
6     int i;
7     if(r <= l) return;
8     troca(&v[pivo_aleatorio(l,r)], &v[l]);
9     i = partition(v, l, r);
10    quicksort_ale(v, l, i-1);
11    quicksort_ale(v, i+1, r);
12 }
```

O tempo de execução depende dos pivôs sorteados

- O tempo médio é  $O(n \lg n)$ 
  - as vezes é lento, as vezes é rápido

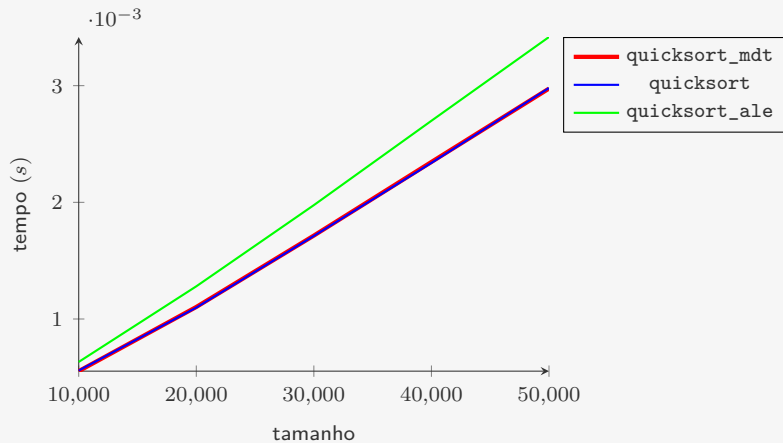
# Quicksort Aleatorizado

```
1 int pivo_aleatorio(int l, int r) {
2     return l + (int)((r-l+1)*(rand() / ((double)RAND_MAX + 1)));
3 }
4
5 void quicksort_ale(int *v, int l, int r) {
6     int i;
7     if(r <= l) return;
8     troca(&v[pivo_aleatorio(l,r)], &v[l]);
9     i = partition(v, l, r);
10    quicksort_ale(v, l, i-1);
11    quicksort_ale(v, i+1, r);
12 }
```

O tempo de execução depende dos pivôs sorteados

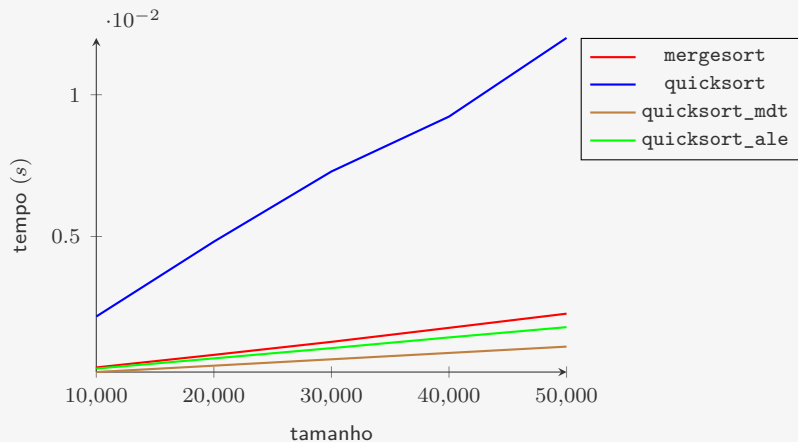
- O tempo médio é  $O(n \lg n)$ 
  - as vezes é lento, as vezes é rápido
  - mas não depende do vetor dado

## Experimentos - Vetores aleatórios



`quicksort_ale` adiciona um overhead desnecessário

## Experimentos - vetores quase ordenados



0,5% de trocas entre pares escolhidos aleatoriamente

- **quicksort\_mdt** é melhor
  - é esperado já que para vetores ordenados ele é  $O(n \lg n)$

## Conclusão

O MergeSort é um algoritmo de ordenação  $O(n \lg n)$



## Conclusão

O MergeSort é um algoritmo de ordenação  $O(n \lg n)$

- Em geral, melhor do que o HeapSort

## Conclusão

O MergeSort é um algoritmo de ordenação  $O(n \lg n)$

- Em geral, melhor do que o HeapSort
- Mas precisa de espaço adicional  $O(n)$

## Conclusão

O MergeSort é um algoritmo de ordenação  $O(n \lg n)$

- Em geral, melhor do que o HeapSort
- Mas precisa de espaço adicional  $O(n)$

O QuickSort é um algoritmo de ordenação  $O(n^2)$

## Conclusão

O MergeSort é um algoritmo de ordenação  $O(n \lg n)$

- Em geral, melhor do que o HeapSort
- Mas precisa de espaço adicional  $O(n)$

O QuickSort é um algoritmo de ordenação  $O(n^2)$

- Mas ele pode ser rápido na prática

## Conclusão

O MergeSort é um algoritmo de ordenação  $O(n \lg n)$

- Em geral, melhor do que o HeapSort
- Mas precisa de espaço adicional  $O(n)$

O QuickSort é um algoritmo de ordenação  $O(n^2)$

- Mas ele pode ser rápido na prática
- Leva tempo  $O(n \lg n)$  (em média) para ordenar uma permutação aleatória

## Conclusão

O MergeSort é um algoritmo de ordenação  $O(n \lg n)$

- Em geral, melhor do que o HeapSort
- Mas precisa de espaço adicional  $O(n)$

O QuickSort é um algoritmo de ordenação  $O(n^2)$

- Mas ele pode ser rápido na prática
- Leva tempo  $O(n \lg n)$  (em média) para ordenar uma permutação aleatória
- Sua versão aleatorizada é  $O(n \lg n)$  em média

## Conclusão

O MergeSort é um algoritmo de ordenação  $O(n \lg n)$

- Em geral, melhor do que o HeapSort
- Mas precisa de espaço adicional  $O(n)$

O QuickSort é um algoritmo de ordenação  $O(n^2)$

- Mas ele pode ser rápido na prática
- Leva tempo  $O(n \lg n)$  (em média) para ordenar uma permutação aleatória
- Sua versão aleatorizada é  $O(n \lg n)$  em média
  - Não importa qual é o vetor de entrada

## Conclusão

O MergeSort é um algoritmo de ordenação  $O(n \lg n)$

- Em geral, melhor do que o HeapSort
- Mas precisa de espaço adicional  $O(n)$

O QuickSort é um algoritmo de ordenação  $O(n^2)$

- Mas ele pode ser rápido na prática
- Leva tempo  $O(n \lg n)$  (em média) para ordenar uma permutação aleatória
- Sua versão aleatorizada é  $O(n \lg n)$  em média
  - Não importa qual é o vetor de entrada
- Usar a mediana de três elementos como pivô pode melhorar o resultado



## Conclusão

O MergeSort é um algoritmo de ordenação  $O(n \lg n)$

- Em geral, melhor do que o HeapSort
- Mas precisa de espaço adicional  $O(n)$

O QuickSort é um algoritmo de ordenação  $O(n^2)$

- Mas ele pode ser rápido na prática
- Leva tempo  $O(n \lg n)$  (em média) para ordenar uma permutação aleatória
- Sua versão aleatorizada é  $O(n \lg n)$  em média
  - Não importa qual é o vetor de entrada
- Usar a mediana de três elementos como pivô pode melhorar o resultado
- Precisa de espaço adicional  $O(n)$  para a pilha de recursão

# Comparação Assintótica

Algoritmo	Melhor Caso	Caso Médio	Pior Caso	Memória
BubbleSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
InsertionSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$

# Comparação Assintótica

Algoritmo	Melhor Caso	Caso Médio	Pior Caso	Memória
BubbleSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
InsertionSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
HeapSort	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	$O(1)$
MergeSort	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	$O(n)$
QuickSort	$O(n \lg n)$	$O(n \lg n)$	$O(n^2)$	$O(n)$

## Exercício

Faça uma versão do MergeSort para listas ligadas.

# Solução

```
1 p_no merge(p_no l1, p_no l2) {
2   p_no l = NULL;
3   if (l1 == NULL)
4     return l2;
5   if (l2 == NULL)
6     return l1;
7   if (l1->dado <= l2->dado) {
8     l = l1;
9     l->prox = merge(l1->prox, l2);
10  } else {
11    l = l2;
12    l->prox = merge(l1, l2->prox);
13  }
14  return l;
15 }
```

# Solução

```
16
17 p_no merge_sort(p_no lista) {
18     if (lista == NULL || lista->prox == NULL)
19         return lista;
20     p_no lento = lista, rapido = lista->prox;
21     while (rapido != NULL && rapido->prox != NULL) {
22         // avança lento uma posição e rapido duas
23         lento = lento->prox;
24         rapido = rapido->prox->prox;
25     }
26     p_no l1 = lista, l2 = lento->prox;
27     lento->prox = NULL; // separa a lista em duas
28     return merge(merge_sort(l1), merge_sort(l2));
29 }
```

## Exercício

Faça uma versão do QuickSort que seja boa para quando há muitos elementos repetidos no vetor.

- A ideia é particionar o vetor em três partes: **menores**, **iguais** e **maiores** que o pivô

# Solução

```
1 void partition(int *v, int l, int r, int *ml, int *mr) {
2     int pivo = v[l];
3     *ml = *mr = r + 1;
4     for (int i = r; i >= l; i--) {
5         if (v[i] >= pivo) {
6             (*ml)--;
7             troca(&v[i], &v[*ml]);
8             if (v[*ml] > pivo) {
9                 (*mr)--;
10                troca(&v[*ml], &v[*mr]);
11            }
12        }
13 }
14
15
16 void quicksort(int *v, int l, int r) {
17     int ml, mr;
18     if (r <= l) return;
19     partition(v, l, r, &ml, &mr);
20     quicksort(v, l, ml - 1);
21     quicksort(v, mr + 1, r);
22 }
```



## Exercício

Implemente a função

```
void mergeAB(int *v, int *a, int n, int *b, int m)
```

que dados vetores **a** e **b** de tamanho **n** e **m** faz a intercalação de **a** e **b** e armazena no vetor **v**. Suponha que **v** já está alocado e que tem tamanho maior ou igual a **n+m**.

# Solução

```
1 void mergeAB(int *v, int *a, int n, int *b, int m) {
2     int i = 0, j = 0, k = 0;
3     while (i < n && j < m)
4         if (a[i] < b[j])
5             v[k++] = a[i++];
6         else
7             v[k++] = b[j++];
8     while (i < n)
9         v[k++] = a[i++];
10    while (j < m)
11        v[k++] = b[j++];
12 }
```

Dúvidas?