DCC
DEPARTAMENTO DE
CIÊNCIA DA COMPUTAÇÃO

UFMG
UNIVERSIDADE FEDERAL
DE MINAS GERAIS

# Estrutura de Dados

## Ordenação: QuickSort

Professores: Anisio Lacerda
Wagner Meira Jr.

# Introdução

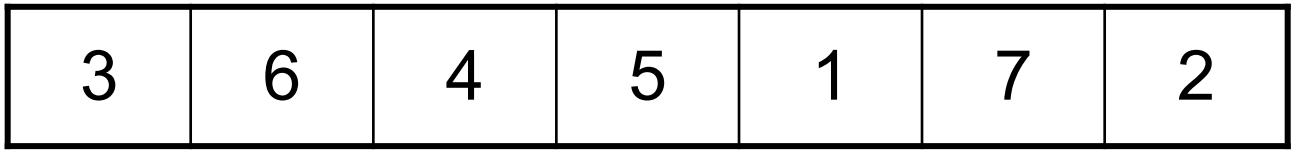
- O quicksort é um algoritmo de divisão e conquista (assim como o merge sort).
- Diferentemente do mergesort o vetor não é dividido necessariamente na metade.
  - Um elemento é escolhido e chamado de **pivô**.
  - Faremos um procedimento de **partição**, com o objetivo de dividir nosso vetor em dois subvetores.
  - Um subvetor contém apenas chaves menores ou iguais ao pivô, enquanto o outro contém chaves maiores ou iguais.
  - Repetimos recursivamente o processo para os dois subvetores.

# Quicksort - Partição

- Algoritmo para o particionamento:
  1. Escolha arbitrariamente um **pivô** x.
  2. Percorra o vetor com um índice **i** a partir da esquerda até que A[i] ≥ x.
  3. Percorra o vetor com um índice **j** a partir da direita até que A[j] ≤ x.
  4. Troque A[i] com A[j].
  5. Continue este processo (de 2 a 4) até os apontadores i e j se cruzarem.

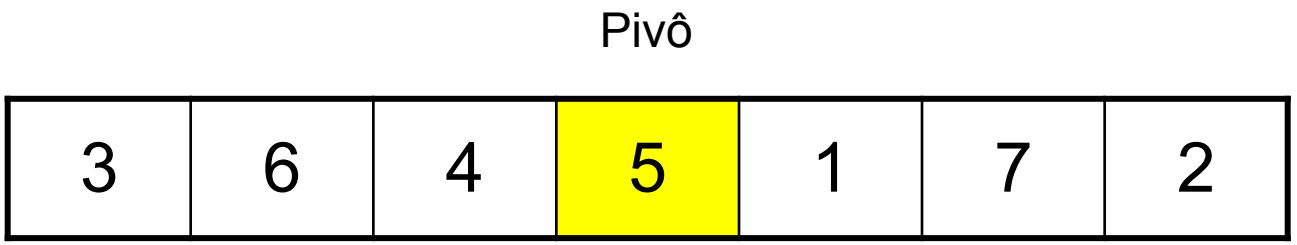
# Partição - Exemplo

*1. Escolha arbitrariamente um pivô x.*



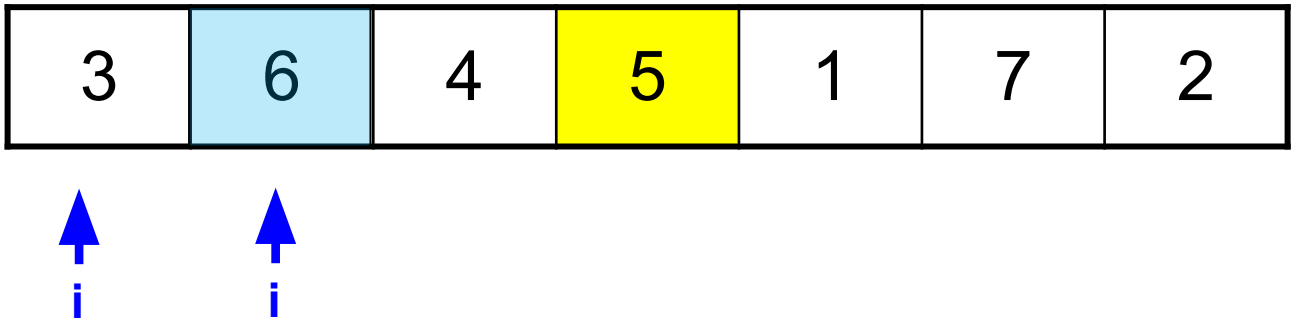
# Partição - Exemplo

1. *Escolha arbitrariamente um pivô*
   *Escolhemos elemento central do vetor A[(i + j) / 2] como pivô*



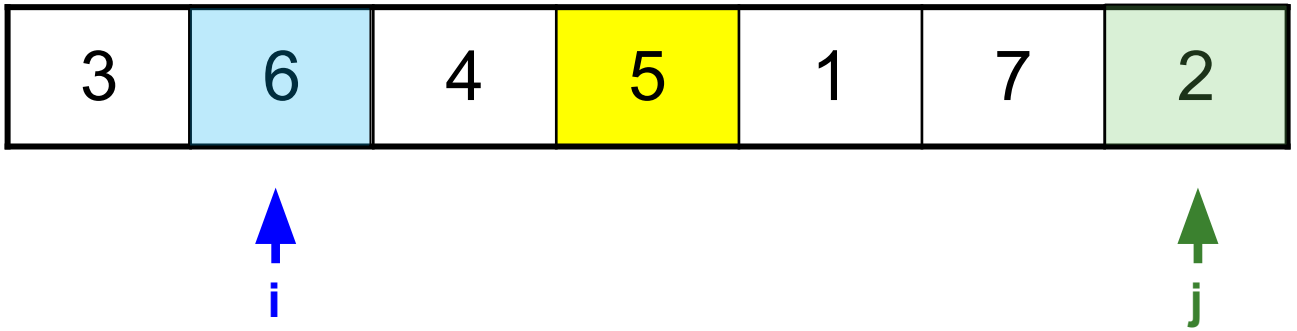
# Partição - Exemplo

*2. Percorra o vetor com um índice i a partir da esquerda até que A[i] ≥ x.*



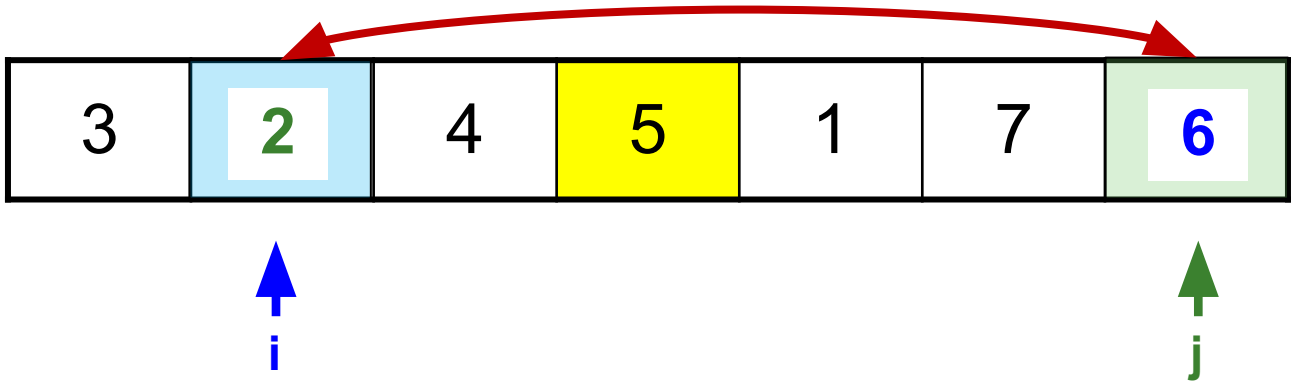
# Partição - Exemplo

*3. Percorra o vetor com um índice j a partir da direita até que A[j] ≤ x.*



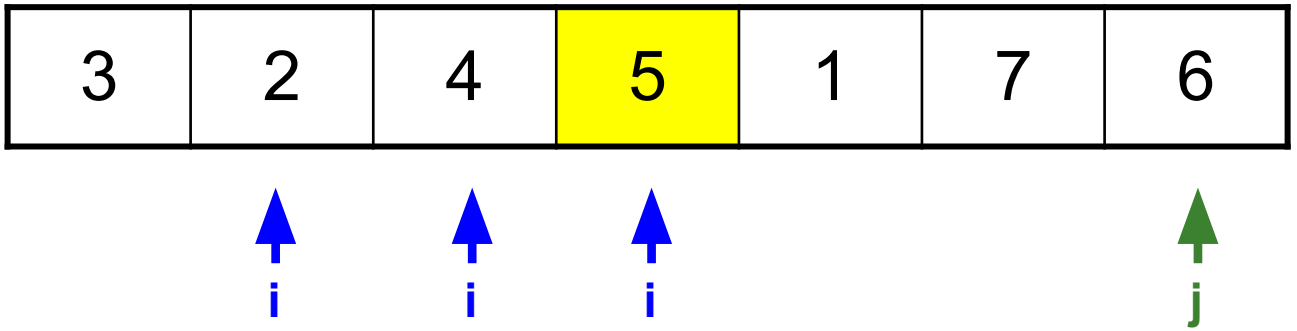
# Partição - Exemplo

*4. Troque A[i] com A[j].*



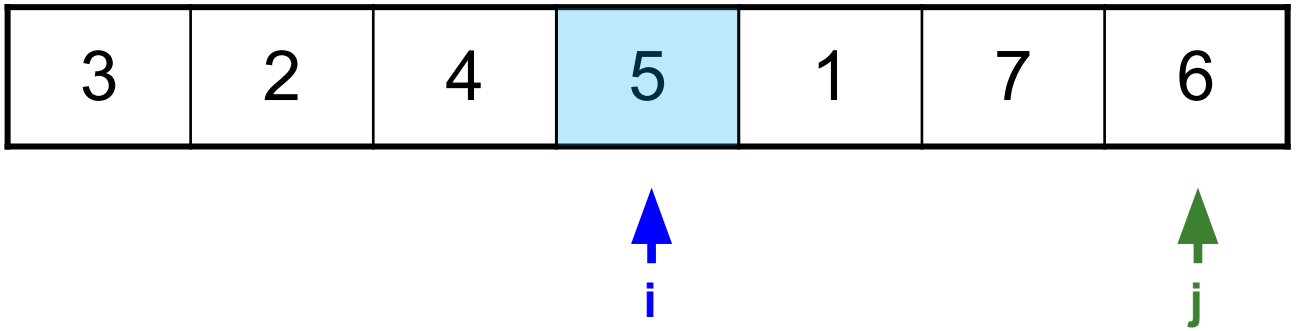
# Partição - Exemplo

*5. Continue este processo (de 2 a 4) até os apontadores i e j se cruzarem.*
*2. Percorra o vetor com um índice i a partir da esquerda até que A[i] ≥ x.*



# Partição - Exemplo

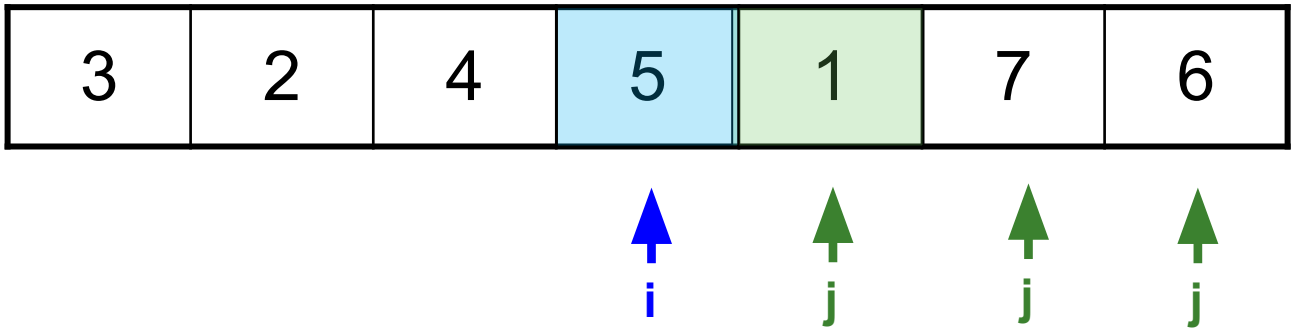
*5. Continue este processo (de 2 a 4) até os apontadores i e j se cruzarem.*
*2. Percorra o vetor com um índice i a partir da esquerda até que A[i] ≥ x.*



# Partição - Exemplo

*5. Continue este processo (de 2 a 4) até os apontadores i e j se cruzarem.*

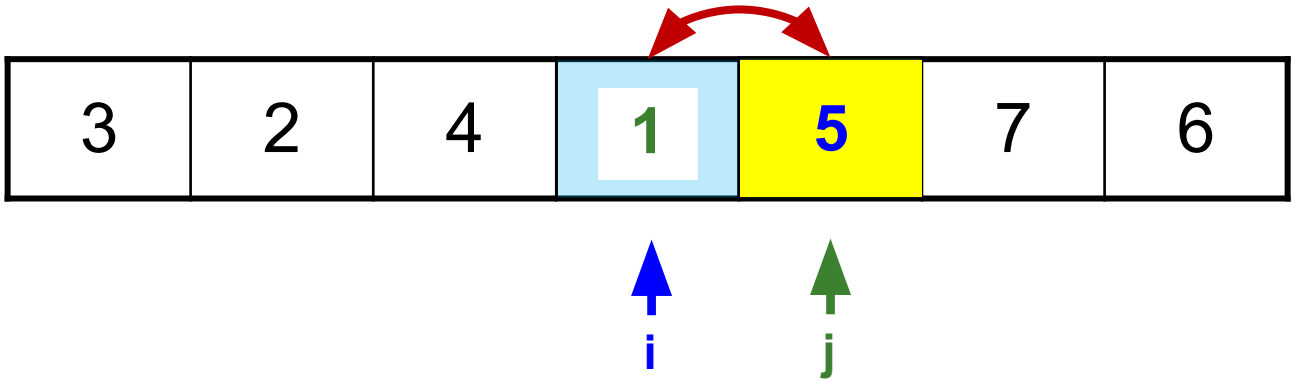
*3. Percorra o vetor com um índice j a partir da direita até que A[j] ≤ x.*



# Partição - Exemplo

*5. Continue este processo (de 2 a 4) até os apontadores i e j se cruzarem.*

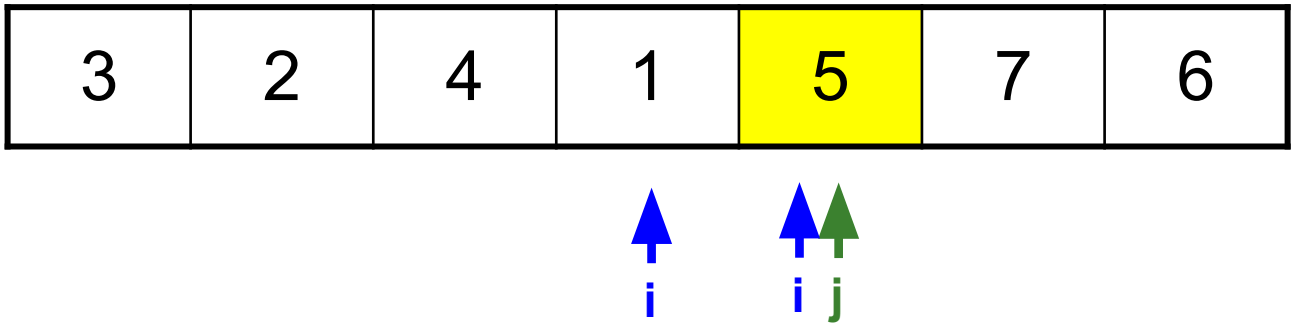
*4. Troque A[i] com A[j].*



# Partição - Exemplo

*5. Continue este processo (de 2 a 4) até os apontadores i e j se cruzarem.*

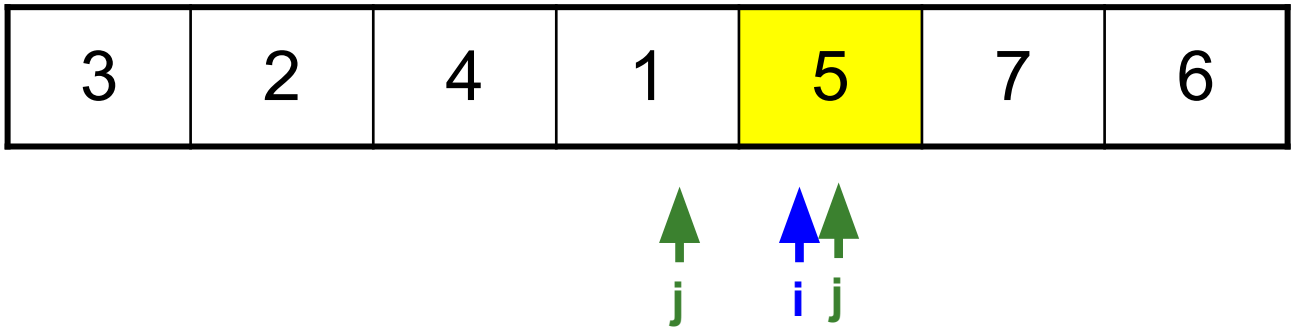
*2. Percorra o vetor com um índice i a partir da esquerda até que A[i] ≥ x.*



# Partição - Exemplo

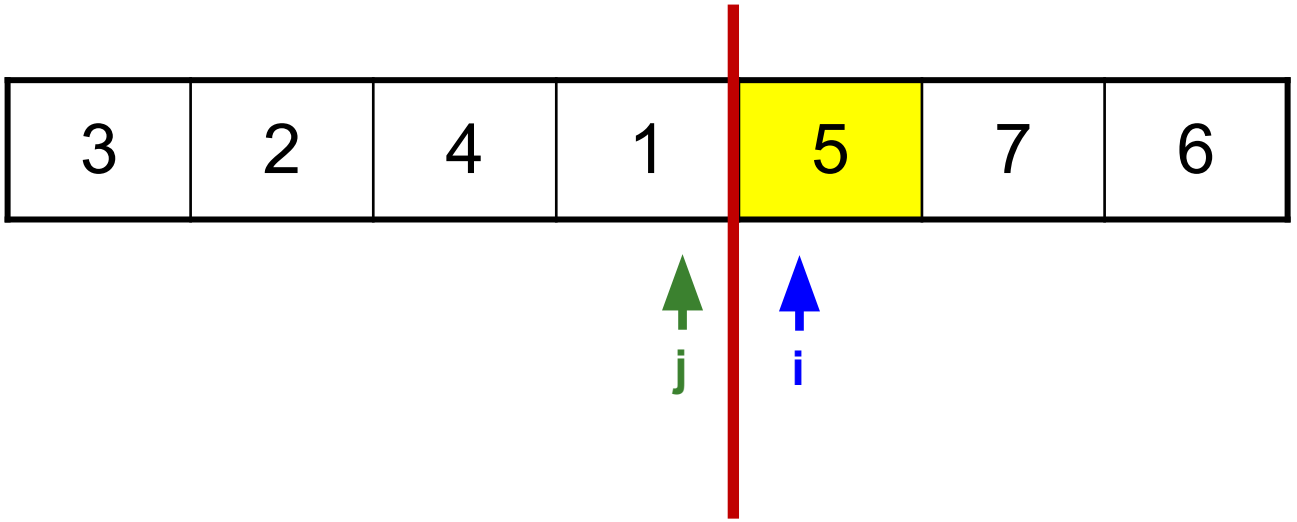
*5. Continue este processo (de 2 a 4) até os apontadores i e j se cruzarem.*

*3. Percorra o vetor com um índice j a partir da direita até que A[j] ≤ x.*



# Partição - Exemplo

*5. Continue este processo (de 2 a 4)* ***até os apontadores i e j se cruzarem.***




*Ponto de partição identificado*

A[Esq] —— A[j] elementos <= pivô

A[i] —— A[Dir] elementos >= pivô

# Quicksort – Após a Partição

- Ao final, do algoritmo de partição:
  - o vetor A[Esq..Dir] está particionado de tal forma que:
    - Os itens em A[Esq], A[Esq + 1], ..., A[j] são menores ou iguais a *x*;
    - Os itens em A[i], A[i +1], ..., A[Dir] são maiores ou iguais a *x*.

# Quicksort - Partição

```
void Particao(int Esq, int Dir,
              int *i, int *j, Item *A)
{
  Item x, w;
  *i = Esq; *j = Dir;
  x = A[(*i + *j)/2]; /* obtem o pivo x */
  do
  {
    while (x.Chave > A[*i].Chave) (*i)++;
    while (x.Chave < A[*j].Chave) (*j)--;
    if (*i <= *j)
    {
      w = A[*i];
      A[*i] = A[*j];
      A[*j] = w;
      (*i)++; (*j)--;
    }
  } while (*i <= *j);
}
```

Índices inicial e final do vetor

Vetor

Índices que vão percorrer o vetor (passados por referência)

# Quicksort - Partição

```
void Particao(int Esq, int Dir,
              int *i, int *j, Item *A)
{
  Item x, w;
  *i = Esq; *j = Dir;
  x = A[(*i + *j)/2]; /* obtem o pivo x */
  do
  {
    while (x.Chave > A[*i].Chave) (*i)++;
    while (x.Chave < A[*j].Chave) (*j)--;
    if (*i <= *j)
    {
      w = A[*i];
      A[*i] = A[*j];
      A[*j] = w;
      (*i)++; (*j)--;
    }
  } while (*i <= *j);
}
```

Inicializa índices i e j, que vão percorrer o vetor

Inicializa pivô com elemento central

Até que os índices se cruzem

# Quicksort - Partição

```
void Particao(int Esq, int Dir,
              int *i, int *j, Item *A)
{
  Item x, w;
  *i = Esq; *j = Dir;
  x = A[(*i + *j)/2]; /* obtem o pivo x */
  do
  {
    while (x.Chave > A[*i].Chave) (*i)++;
    while (x.Chave < A[*j].Chave) (*j)--;
    if (*i <= *j)
    {
      w = A[*i];
      A[*i] = A[*j];
      A[*j] = w;
      (*i)++; (*j)--;
    }
  } while (*i <= *j);
}
```

i procura elemento maior que pivô

j procura elemento menor que pivô

# Quicksort - Partição

```
void Particao(int Esq, int Dir,
              int *i, int *j, Item *A)
{
  Item x, w;
  *i = Esq; *j = Dir;
  x = A[(*i + *j)/2]; /* obtem o pivo x */
  do
  {
    while (x.Chave > A[*i].Chave) (*i)++;
    while (x.Chave < A[*j].Chave) (*j)--;
    if (*i <= *j)
    {
      w = A[*i];
      A[*i] = A[*j];
      A[*j] = w;
      (*i)++; (*j)--;
    }
  } while (*i <= *j);
}
```

Se i e j ainda não se cruzaram, então:

Troca os elementos de índice i e j de lugar

i e j passam para próxima posição

# Quicksort

- O anel interno da função de Partição é extremamente simples.

- Razão pela qual o algoritmo Quicksort é tão rápido.

# Quicksort - Função

```
/* Entra aqui o procedimento Particao */
void Ordena(int Esq, int Dir, Item *A)
{ int i, int j;
  Particao(Esq, Dir, &i, &j, A);
  if (Esq < j) Ordena(Esq, j, A);
  if (i < Dir) Ordena(i, Dir, A);
}

void QuickSort(Item *A, int n)
{
  Ordena(0, n-1, A);
}
```

# Quicksort - Função

```
/* Entra aqui o procedimento Particao */
void Ordena(int Esq, int Dir, Item *A)
{ int i, int j;
  Particao(Esq, Dir, &i, &j, A);
  if (Esq < j) Ordena(Esq, j, A);
  if (i < Dir) Ordena(i, Dir, A);
}

void QuickSort(Item *A, int n)
{
  Ordena(0, n-1, A);
}
```

Vetor

Intervalo do vetor a ser ordenado

# Quicksort - Função

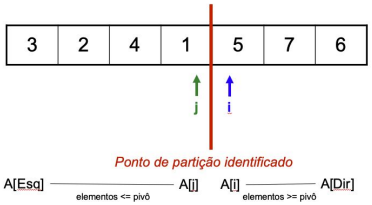
```
/* Entra aqui o procedimento Particao */
void Ordena(int Esq, int Dir, Item *A)
{ int i, int j;
  Particao(Esq, Dir, &i, &j, A);
  if (Esq < j) Ordena(Esq, j, A);
  if (i < Dir) Ordena(i, Dir, A);
}

void QuickSort(Item *A, int n)
{
  Ordena(0, n-1, A);
}
```

Índices i e j, que vão percorrer o vetor

Chama partição, com índices i e j passados por referência

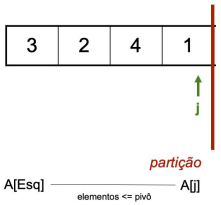
# Quicksort - Função

```
/* Entra aqui o procedimento Particao */
void Ordena(int Esq, int Dir, Item *A)
{ int i, int j;
   Particao(Esq, Dir, &i, &j, A);
   if (Esq < j) Ordena(Esq, j, A);
   if (i < Dir) Ordena(i, Dir, A);
}

void QuickSort(Item *A, int n)
{
   Ordena(0, n-1, A);
}
```



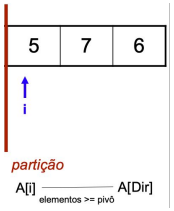
# Quicksort - Função

```
/* Entra aqui o procedimento Particao */
void Ordena(int Esq, int Dir, Item *A)
{ int i, int j;
  Particao(Esq, Dir, &i, &j, A);
  if (Esq < j) Ordena(Esq, j, A);
  if (i < Dir) Ordena(i, Dir, A);
}

void QuickSort(Item *A, int n)
{
  Ordena(0, n-1, A);
}
```



# Quicksort - Função

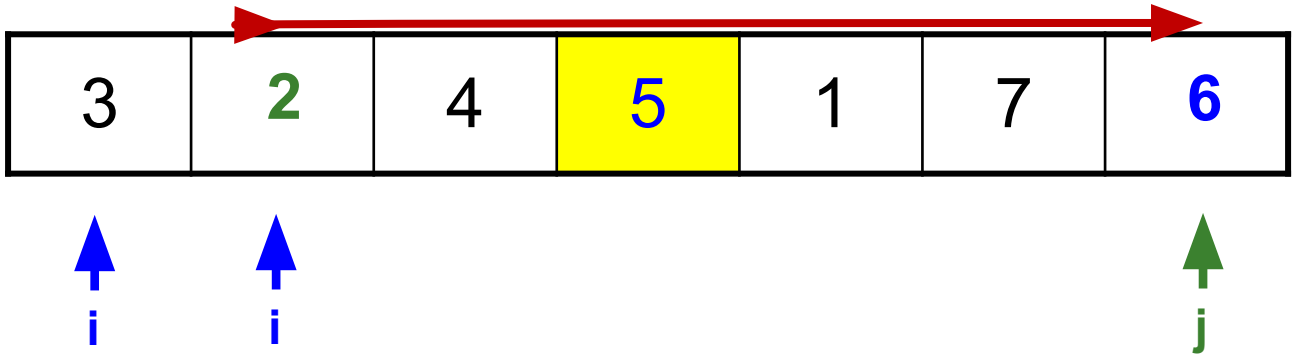
```
/* Entra aqui o procedimento Particao */
void Ordena(int Esq, int Dir, Item *A)
{ int i, int j;
  Particao(Esq, Dir, &i, &j, A);
  if (Esq < j) Ordena(Esq, j, A);
  if (i < Dir) Ordena(i, Dir, A);
}

void QuickSort(Item *A, int n)
{
  Ordena(0, n-1, A);
}
```



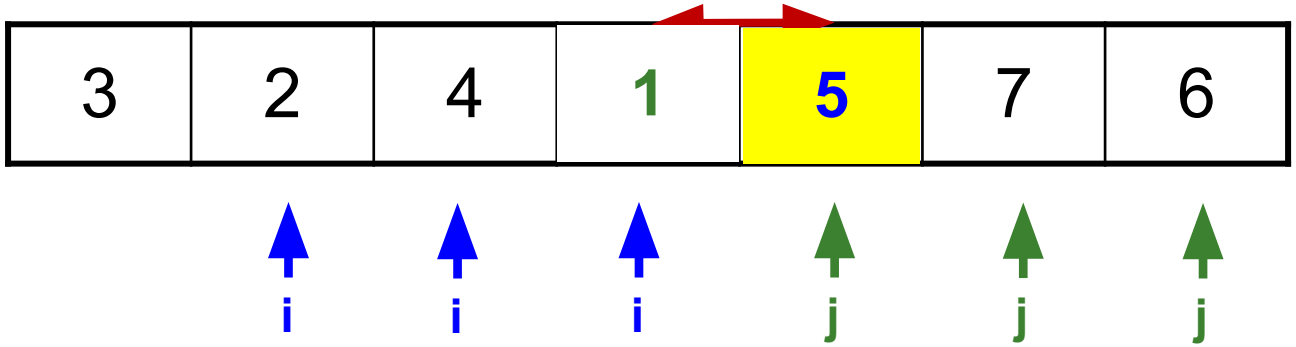
# Quicksort - Função

```
/* Entra aqui o procedimento Particao */
void Ordena(int Esq, int Dir, Item *A)
{ int i, int j;
  Particao(Esq, Dir, &i, &j, A);
  if (Esq < j) Ordena(Esq, j, A);
  if (i < Dir) Ordena(i, Dir, A);
}

void QuickSort(Item *A, int n)
{
  Ordena(0, n-1, A);
}
```

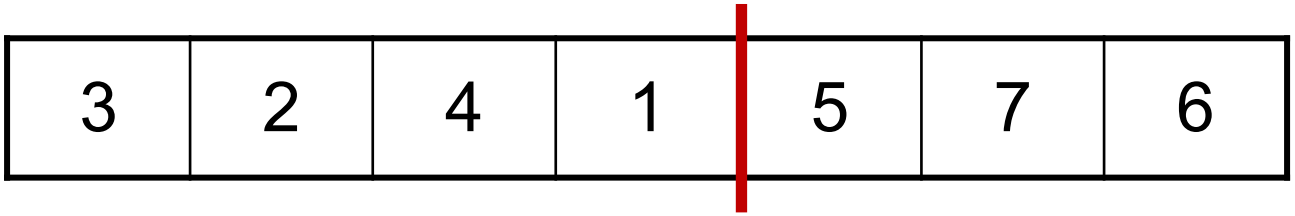
# Quicksort - Exemplo

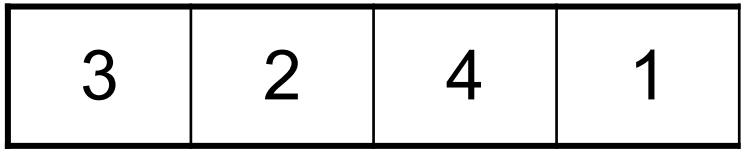


# Quicksort - Exemplo

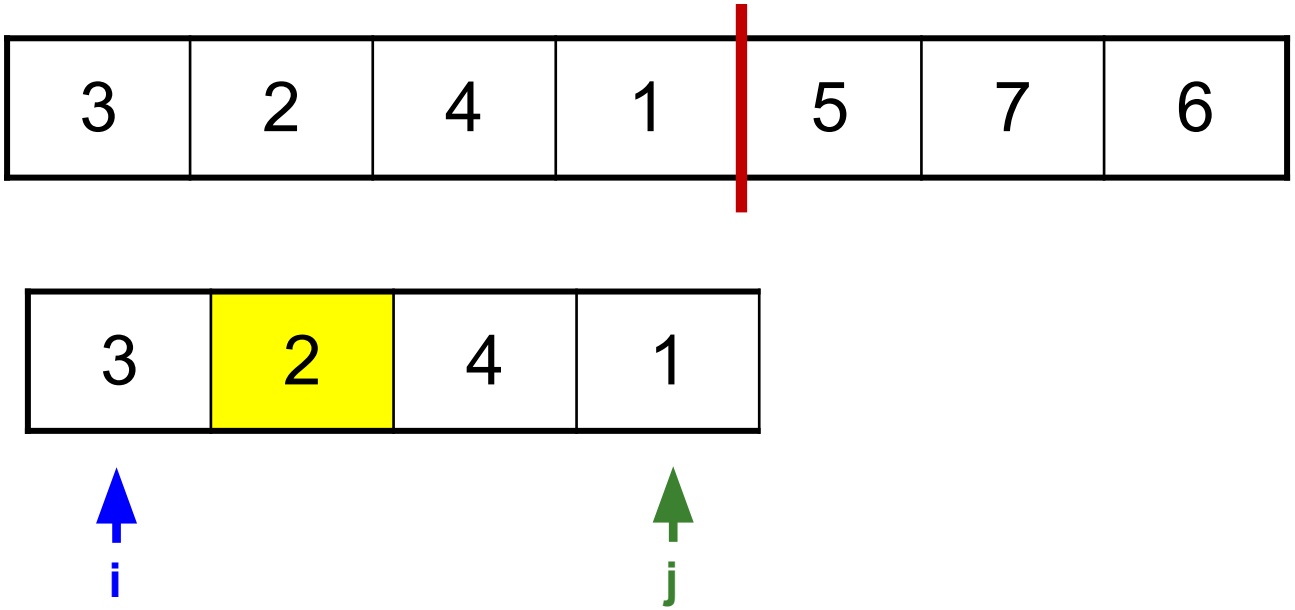


# Quicksort - Exemplo

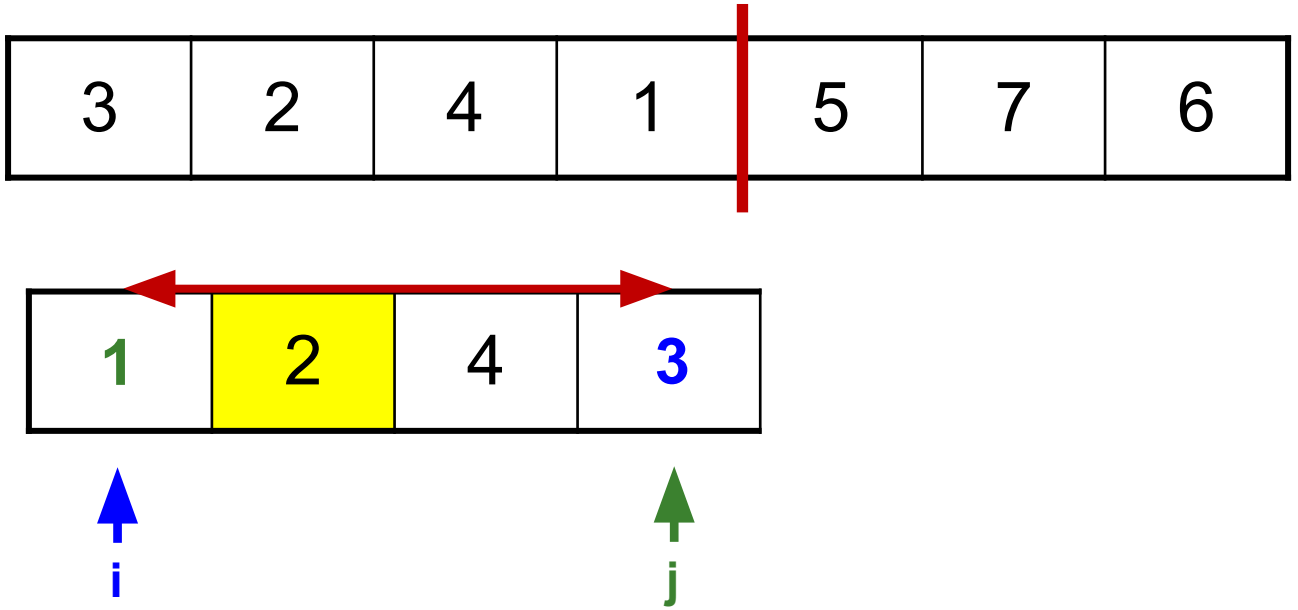


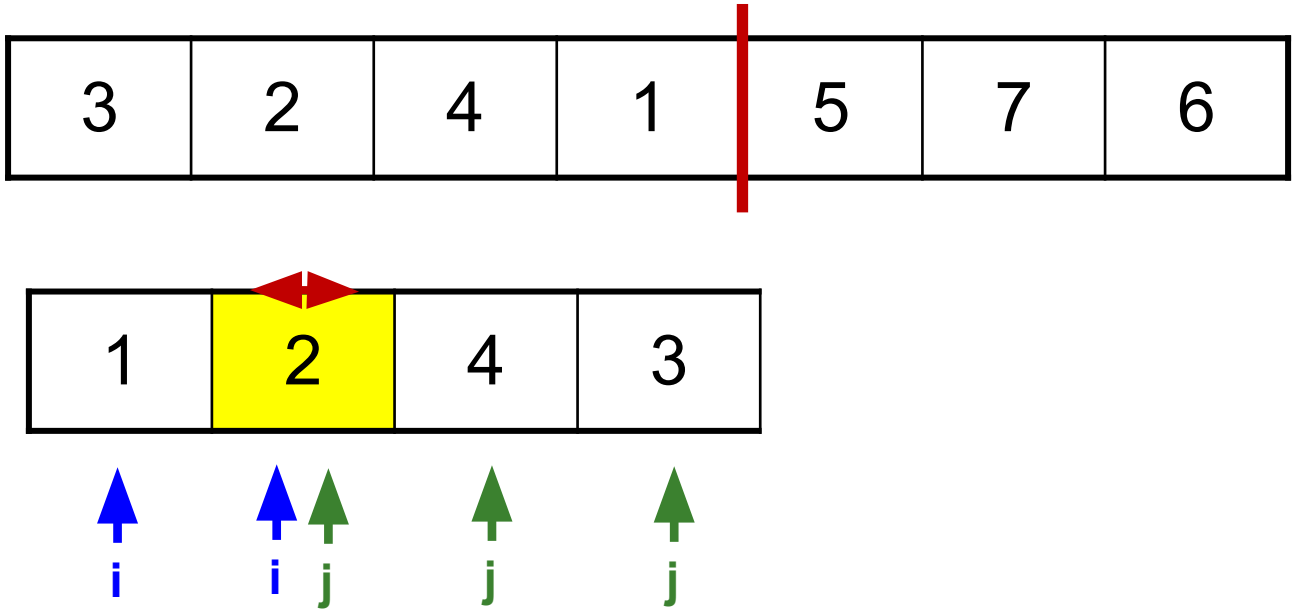
# Quicksort - Exemplo



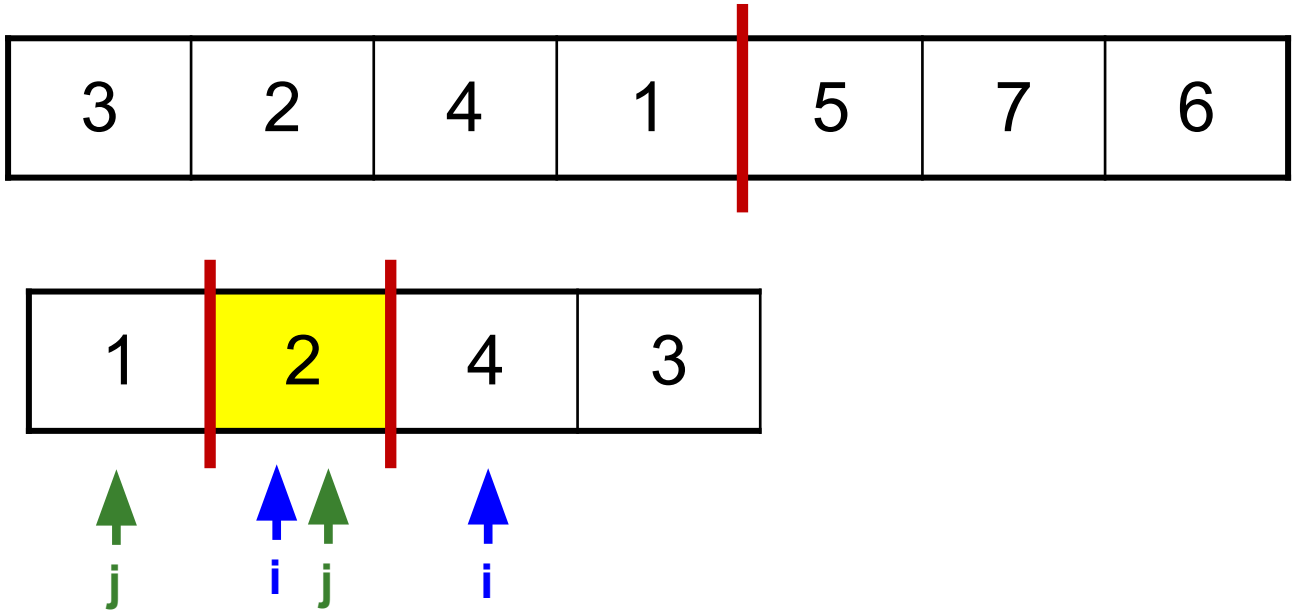
# Quicksort - Exemplo



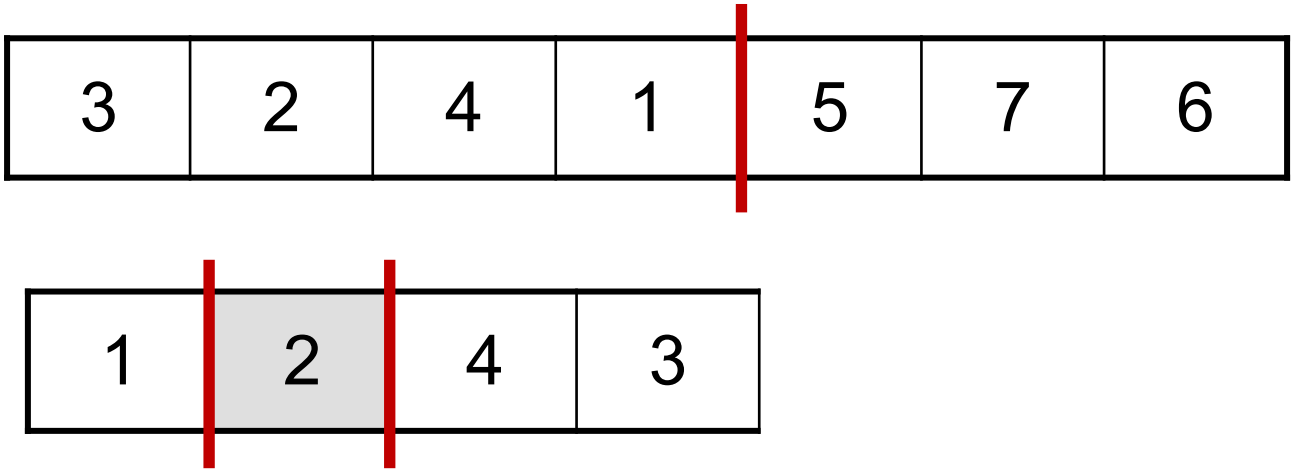
# Quicksort - Exemplo



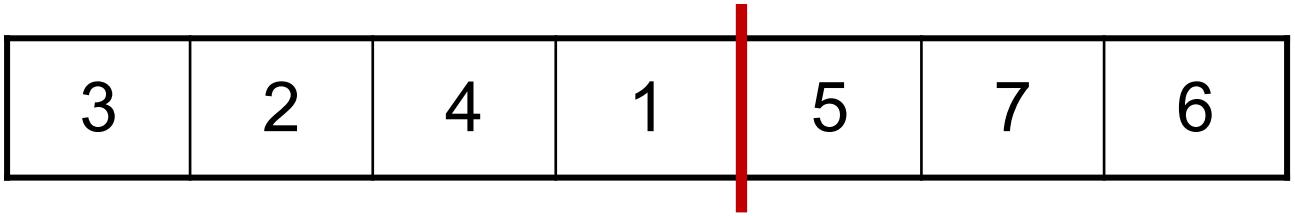
# Quicksort - Exemplo

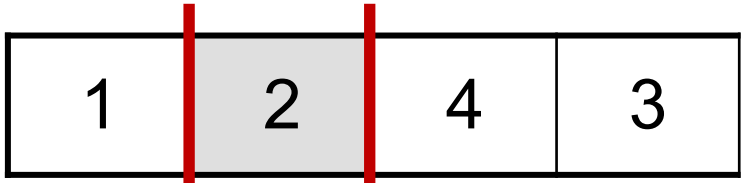


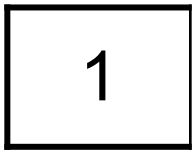
# Quicksort - Exemplo



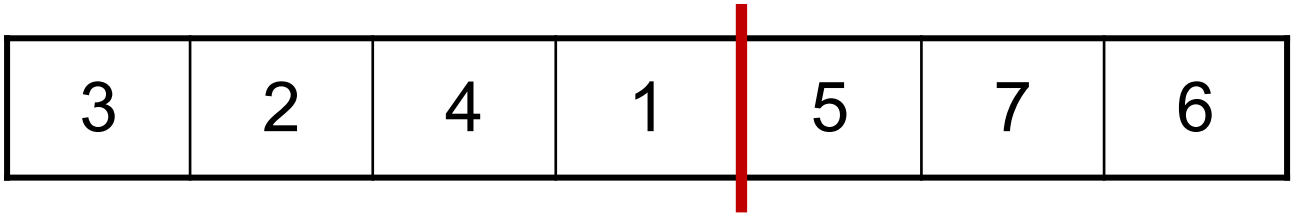
# Quicksort - Exemplo

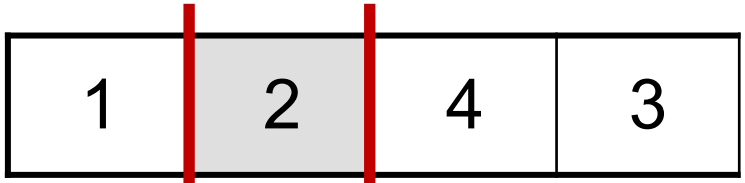


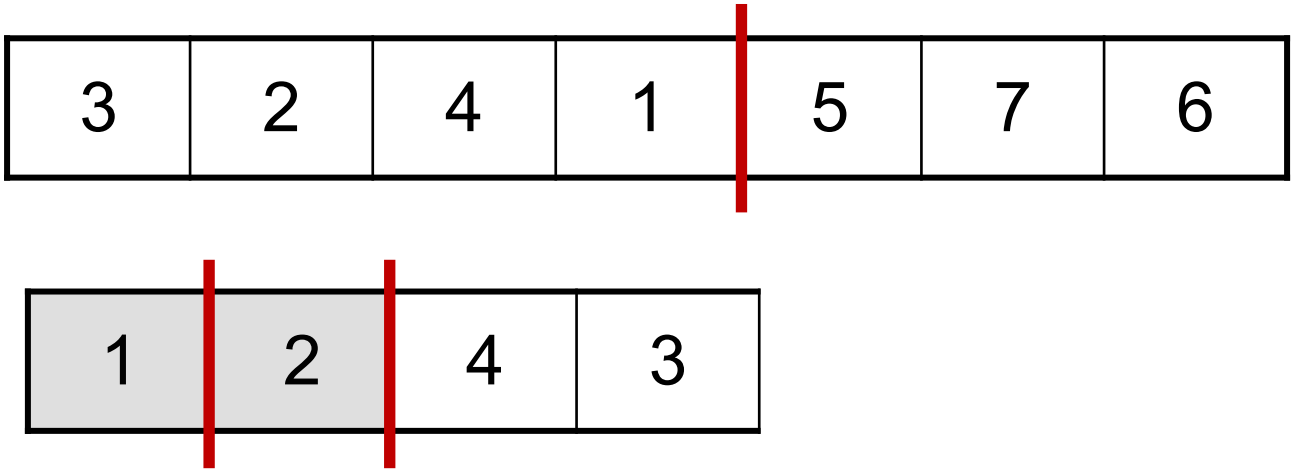



# Quicksort - Exemplo

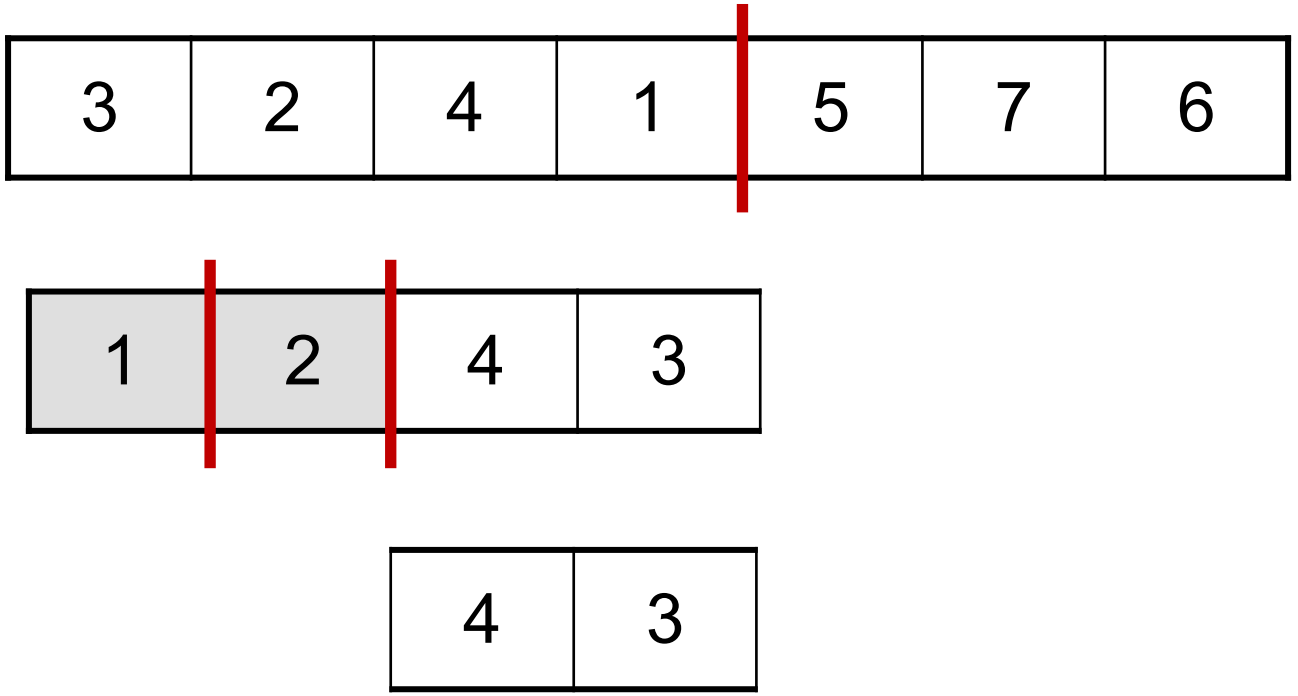


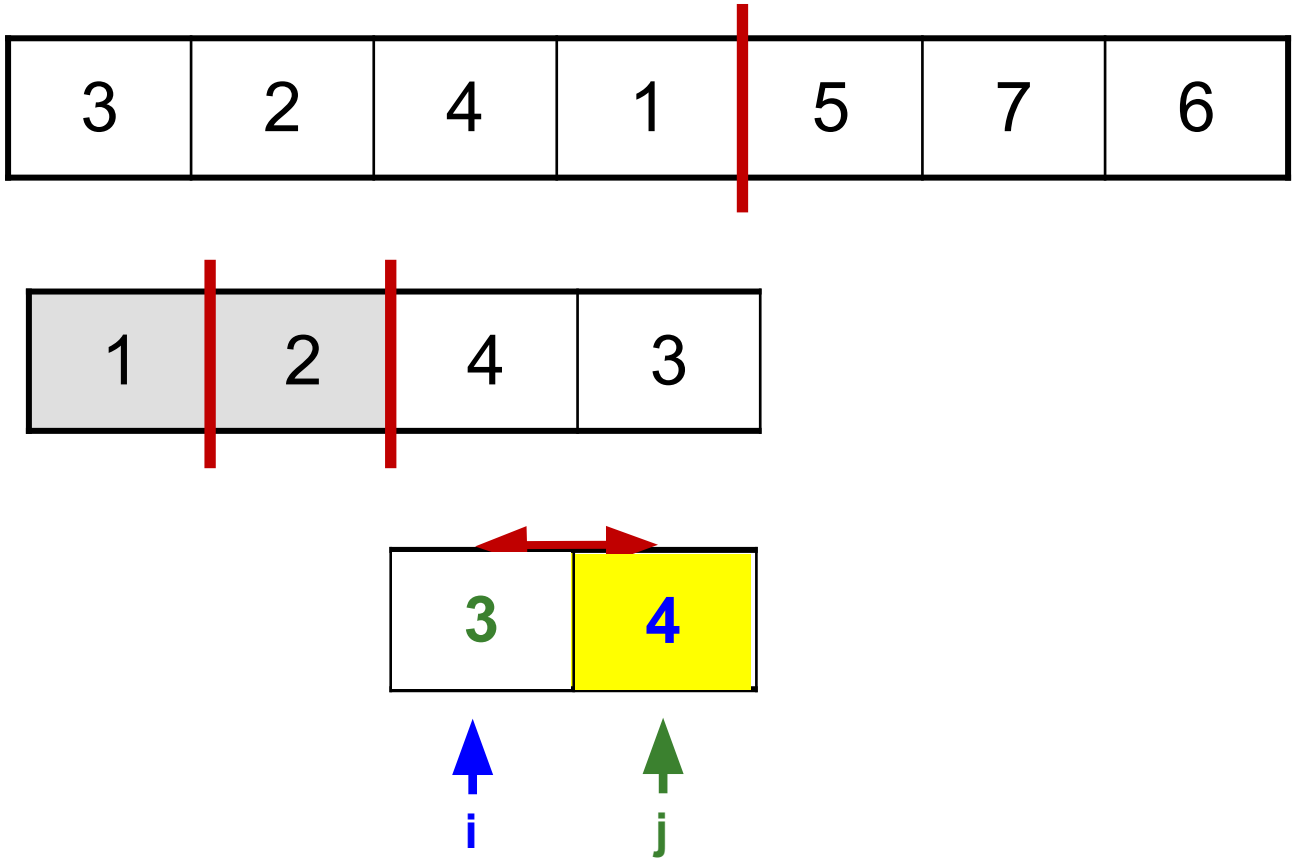

# Quicksort - Exemplo



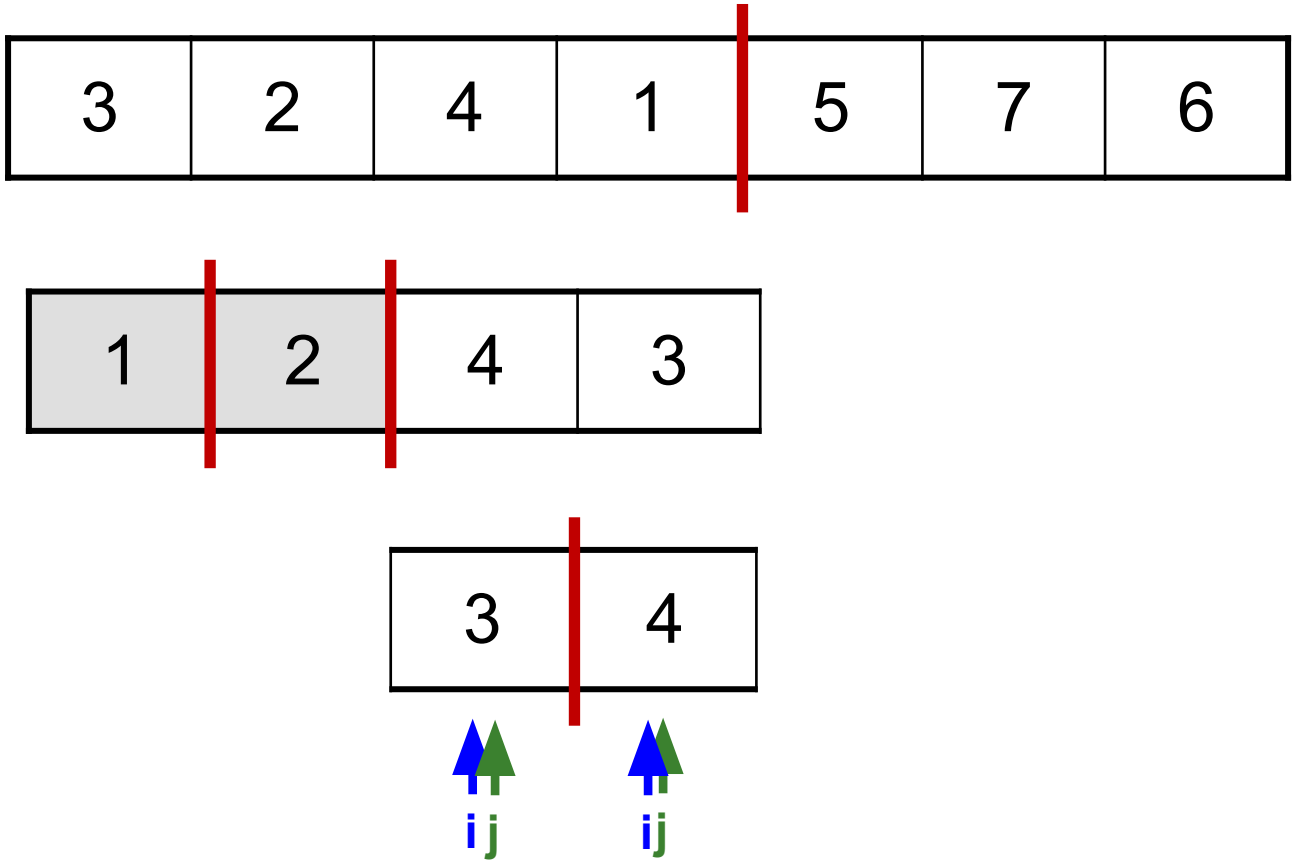
# Quicksort - Exemplo



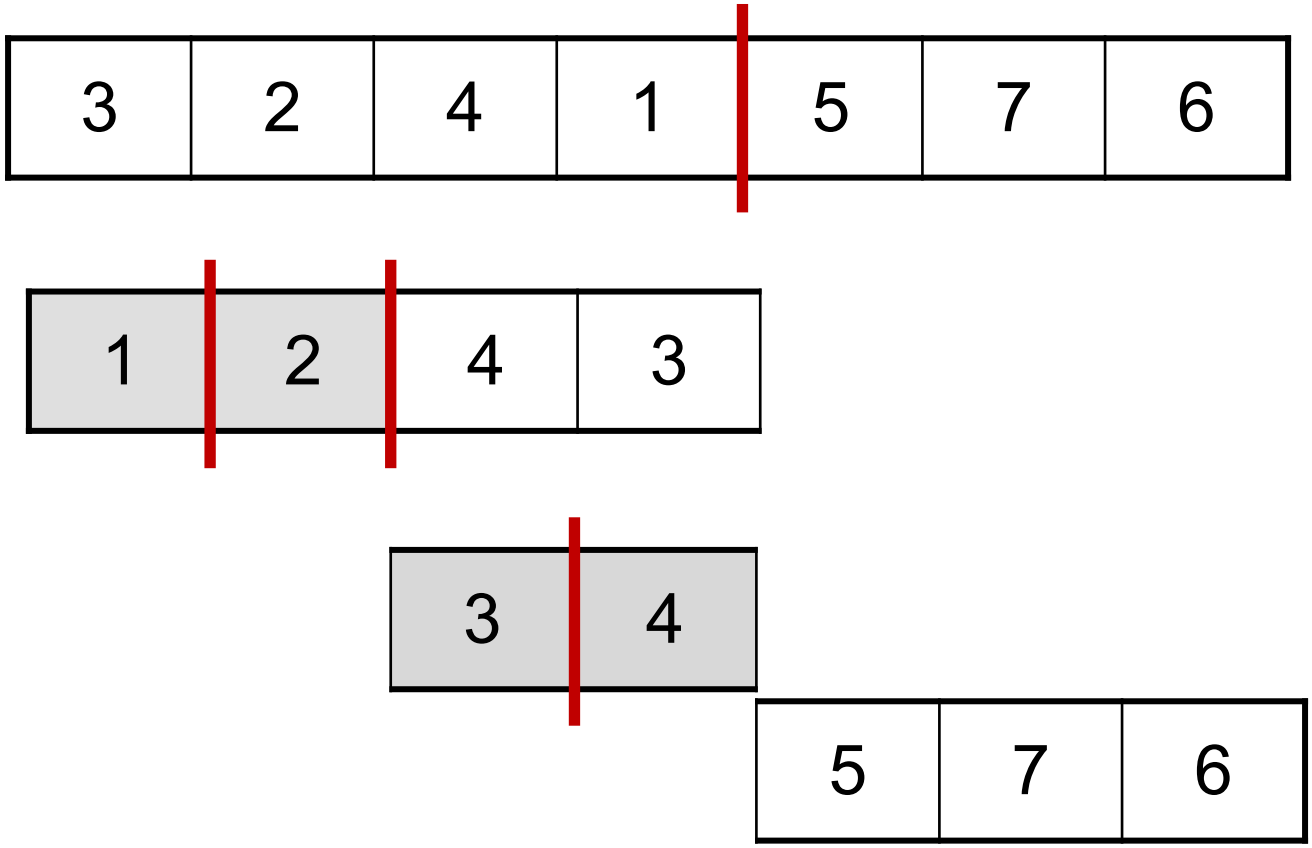
# Quicksort - Exemplo



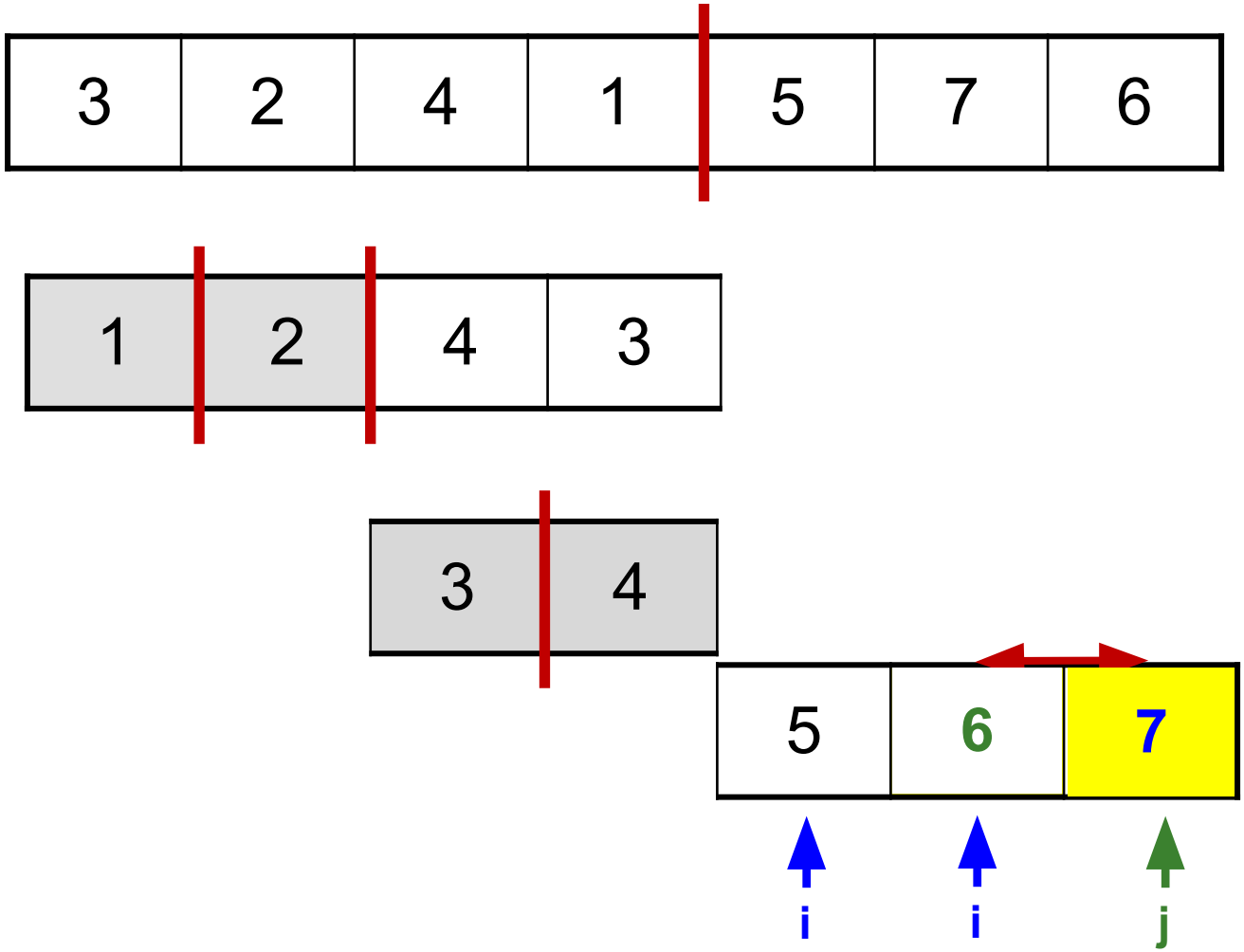
# Quicksort - Exemplo



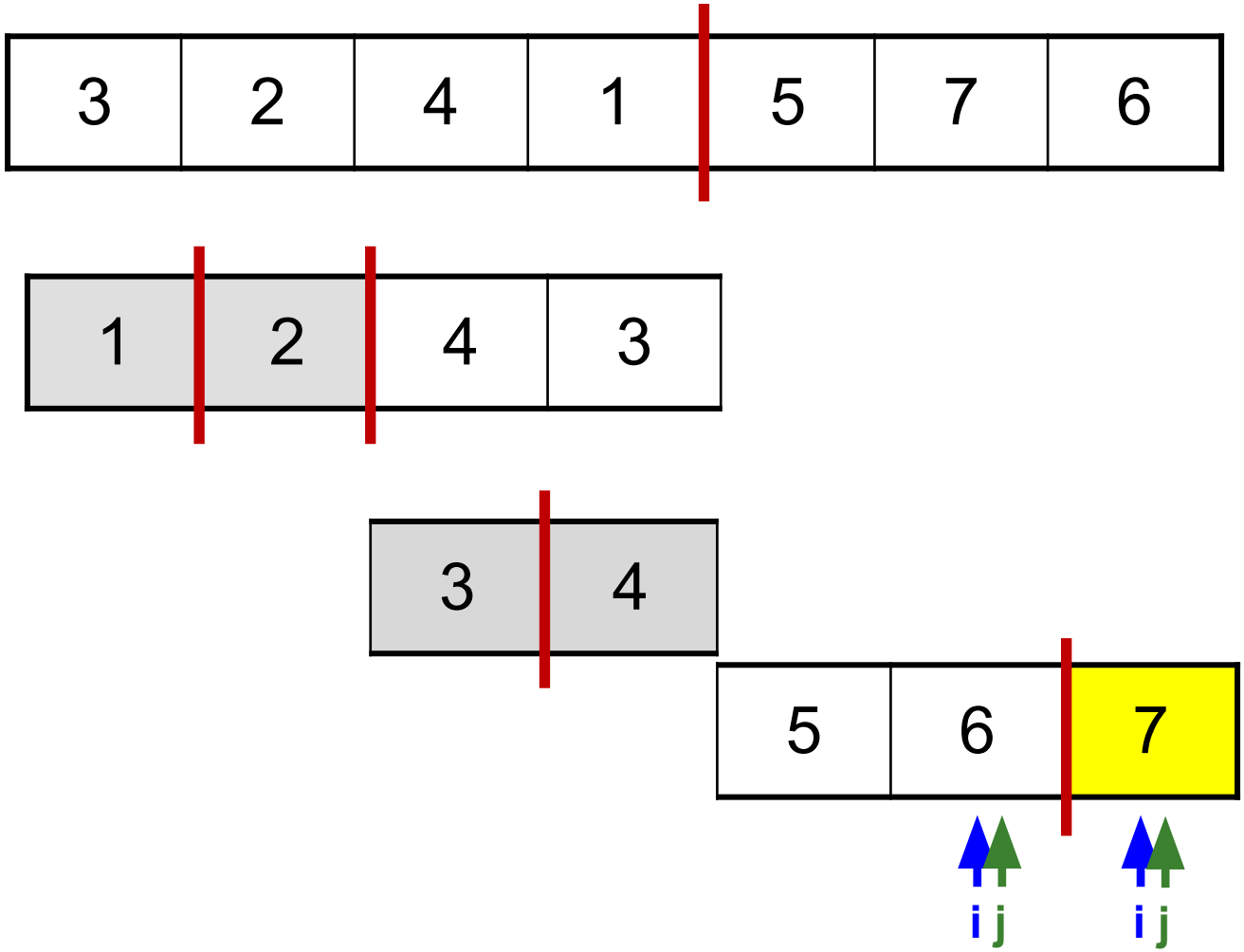
# Quicksort - Exemplo



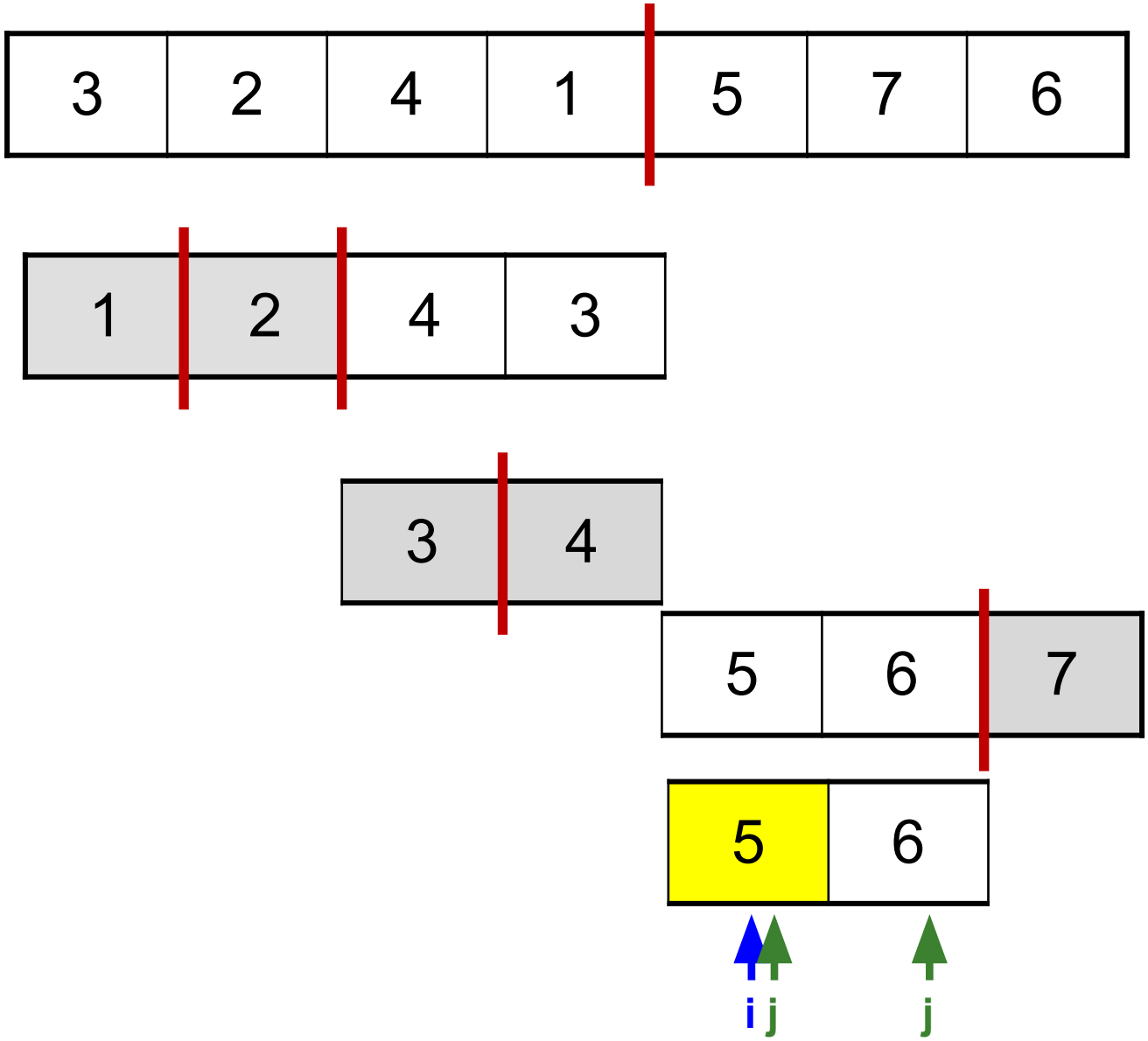
# Quicksort - Exemplo



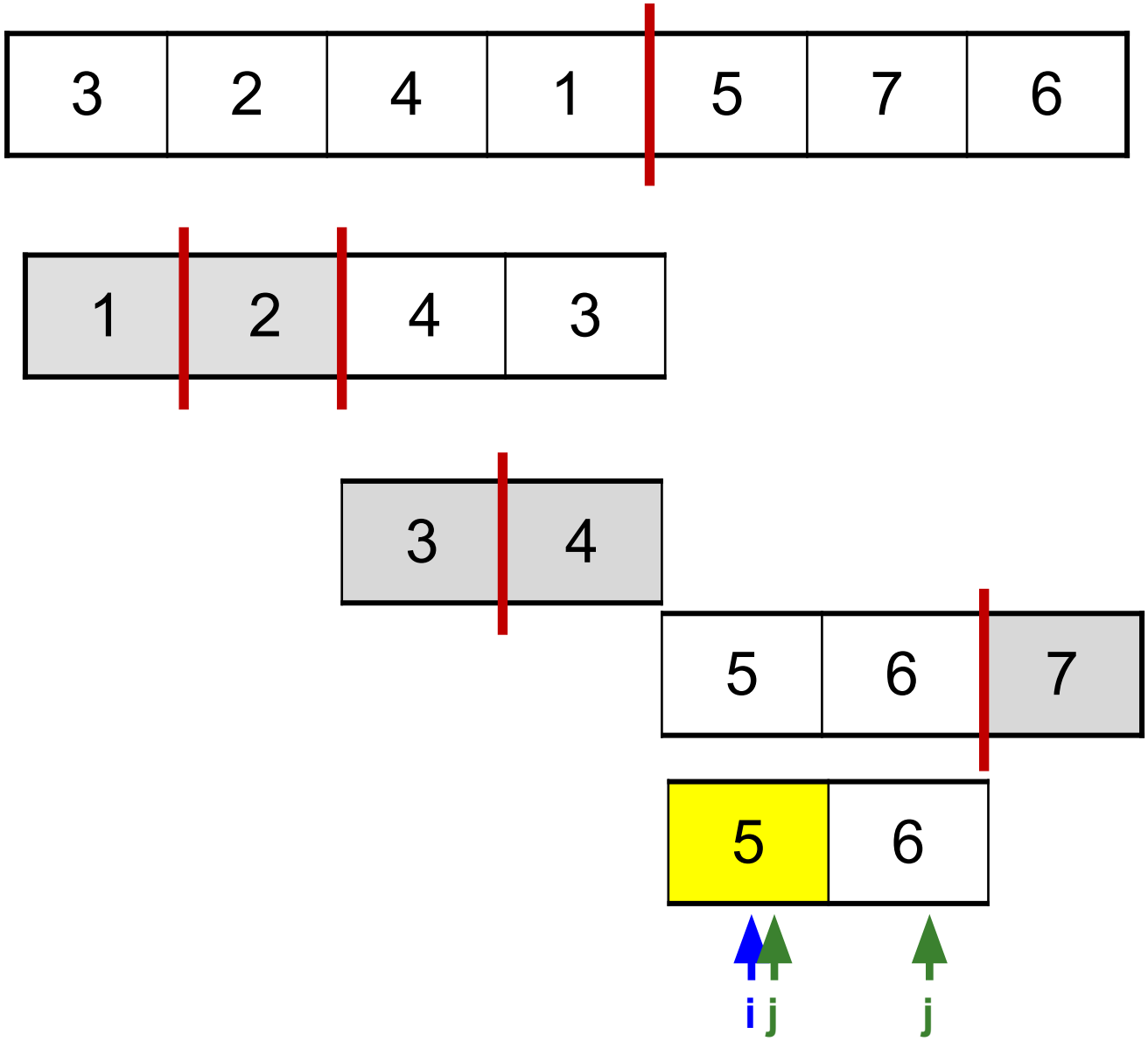
# Quicksort - Exemplo



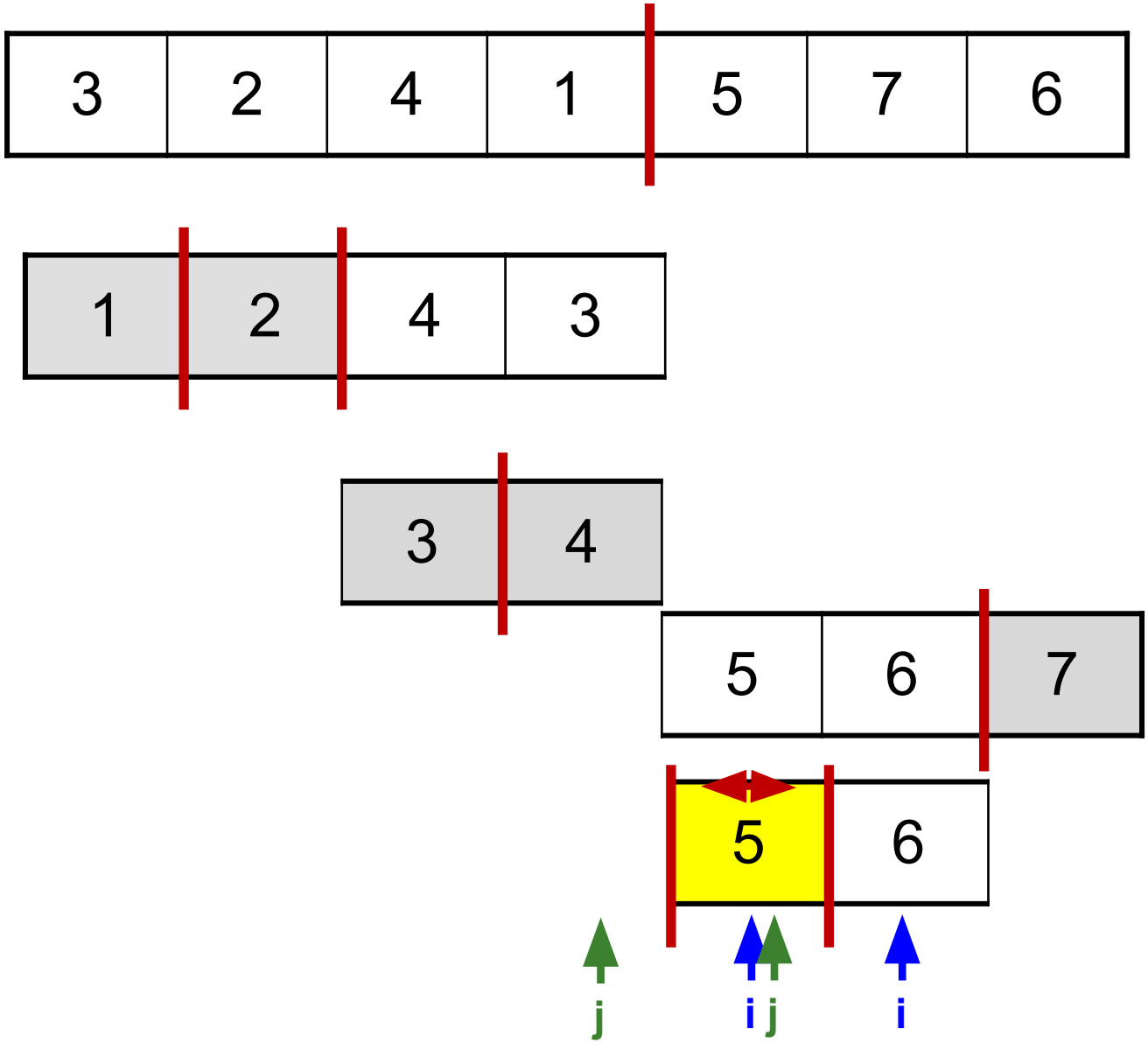
# Quicksort - Exemplo



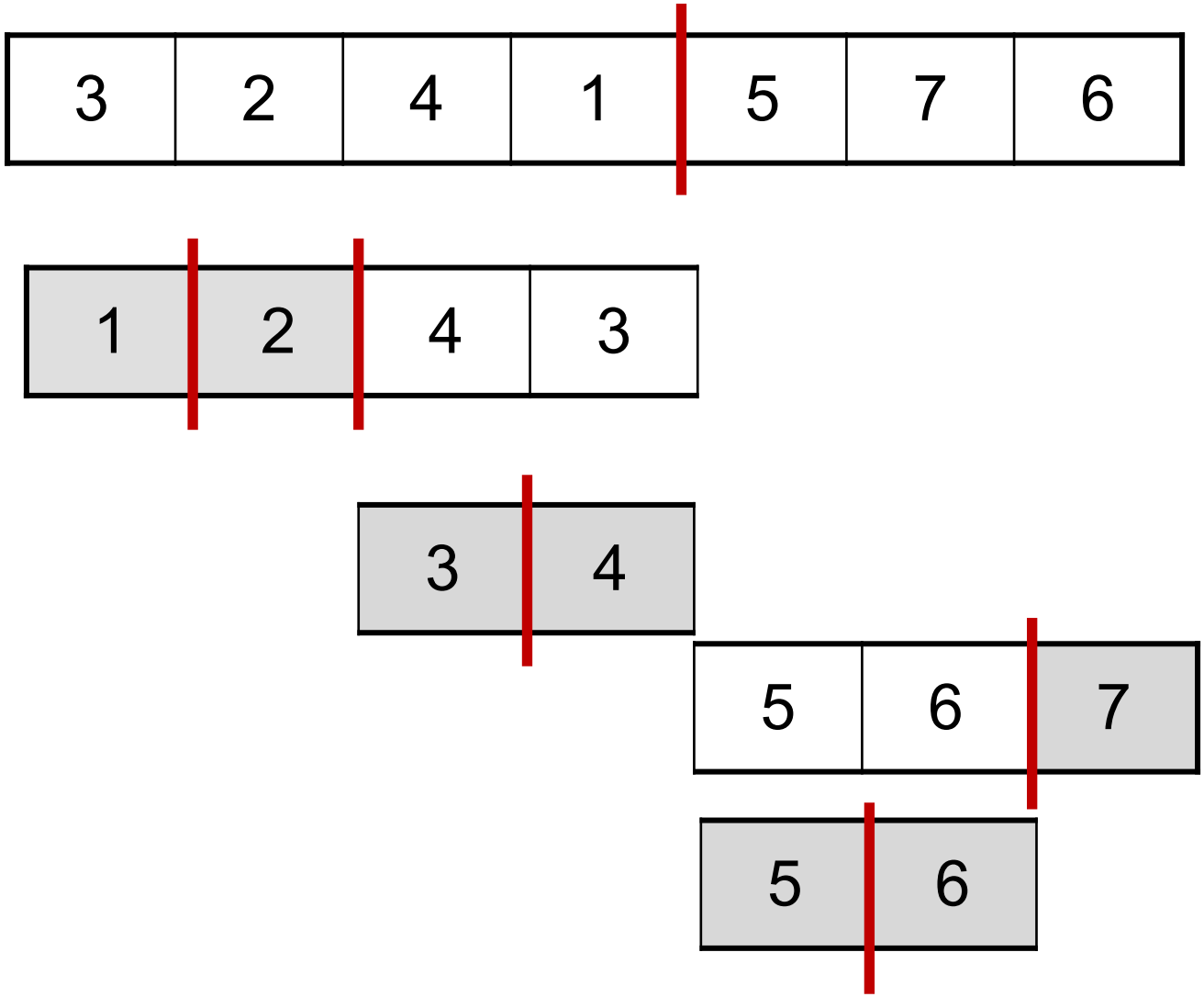
# Quicksort - Exemplo



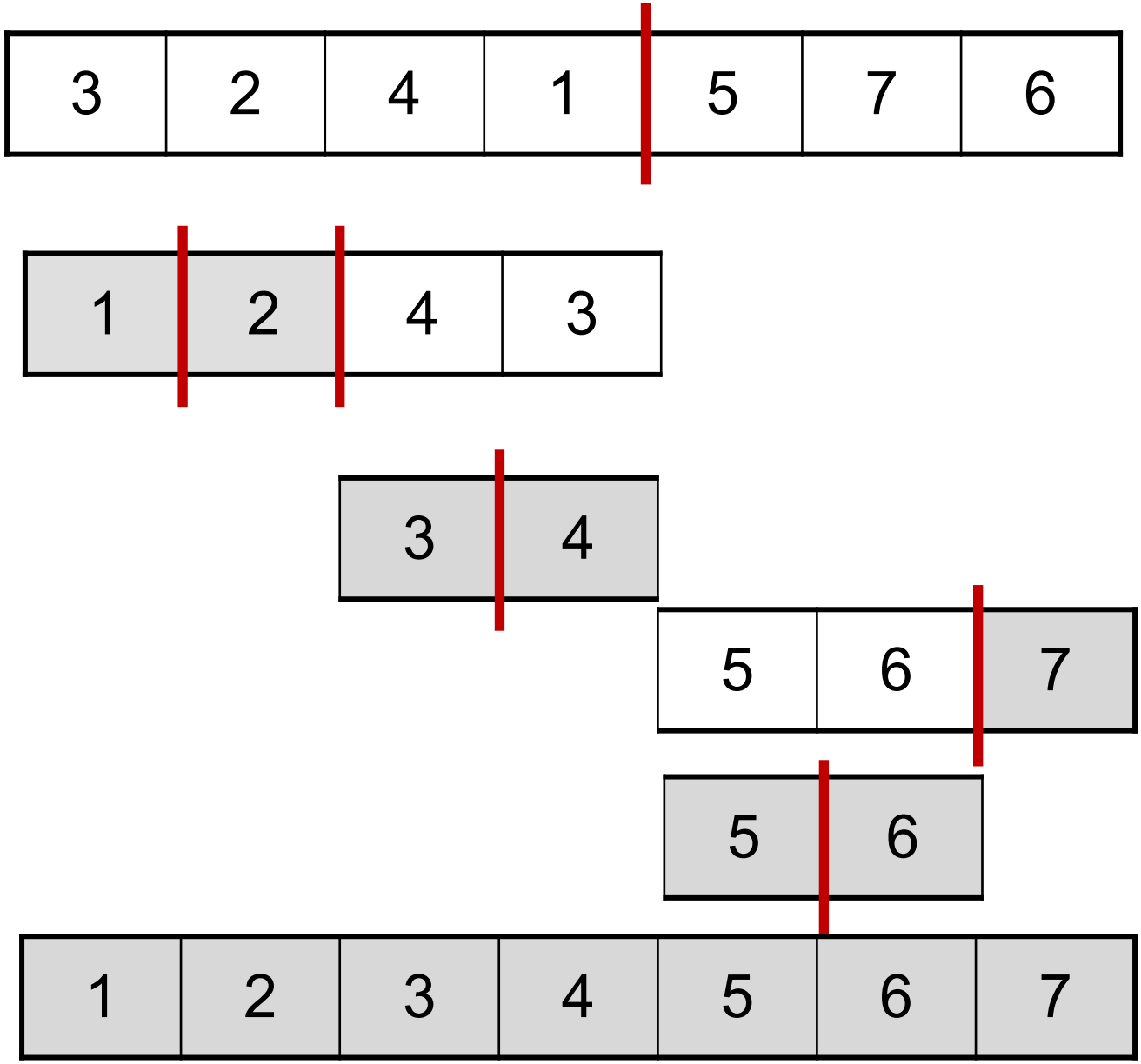
# Quicksort - Exemplo



# Quicksort - Exemplo

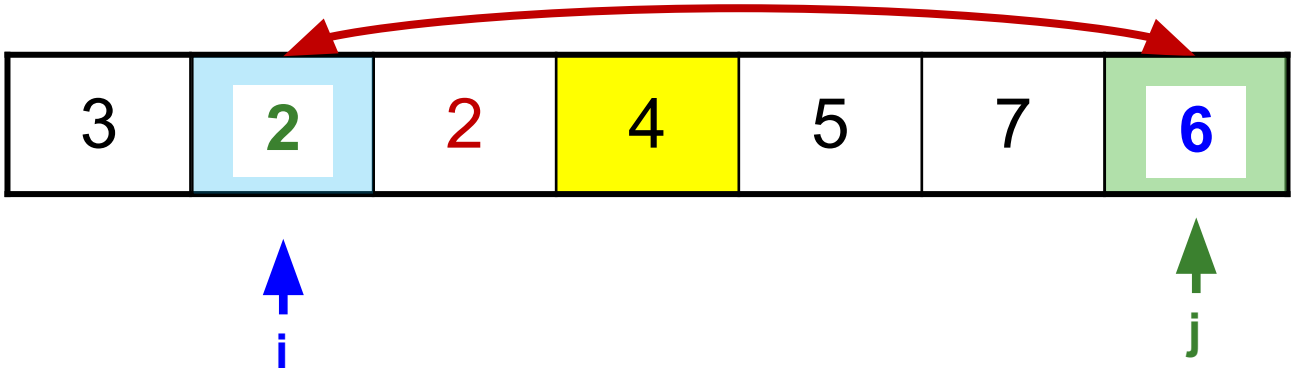


# Quicksort - Exemplo

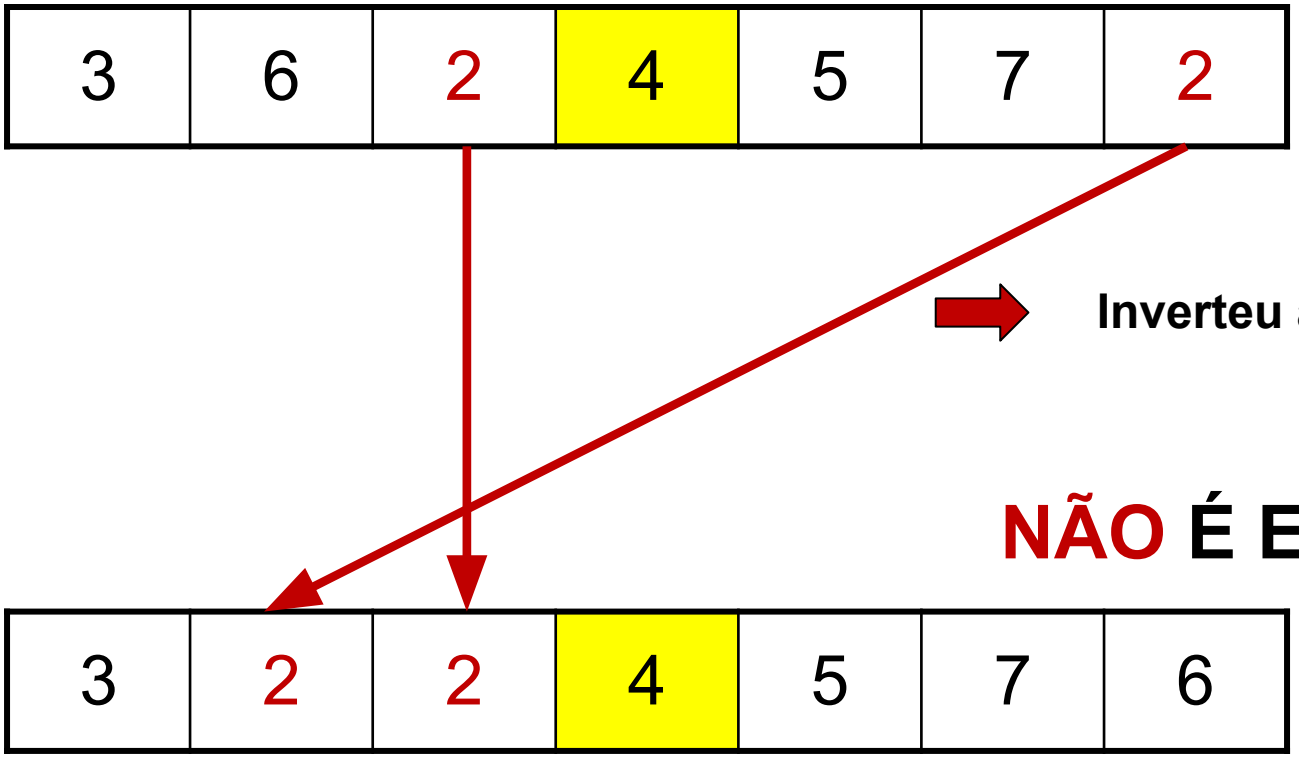


# QUICKSORT - ANÁLISE

# O Quicksort é estável?



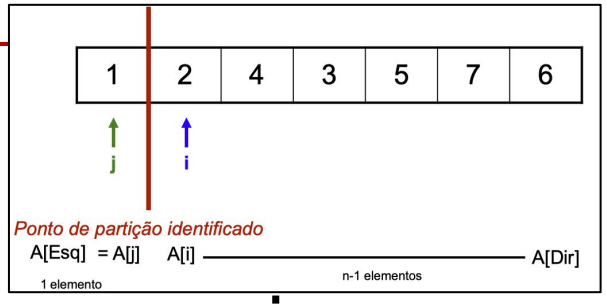
# O Quicksort é estável?



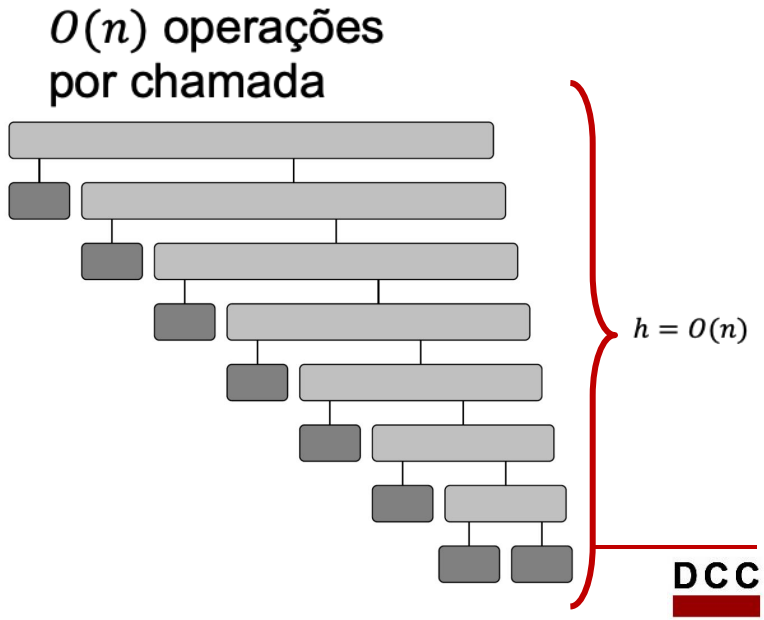

**Inverteu a posição dos '2's**

**NÃO É ESTÁVEL!**

# Qual o pior caso?




- Vimos que se o pivô escolhido for o maior ou menor elemento do vetor, a partição será 1 e n-1.
- Se para todas as partições da chamada, acontecer esta situação, será ordenado um elemento por vez.



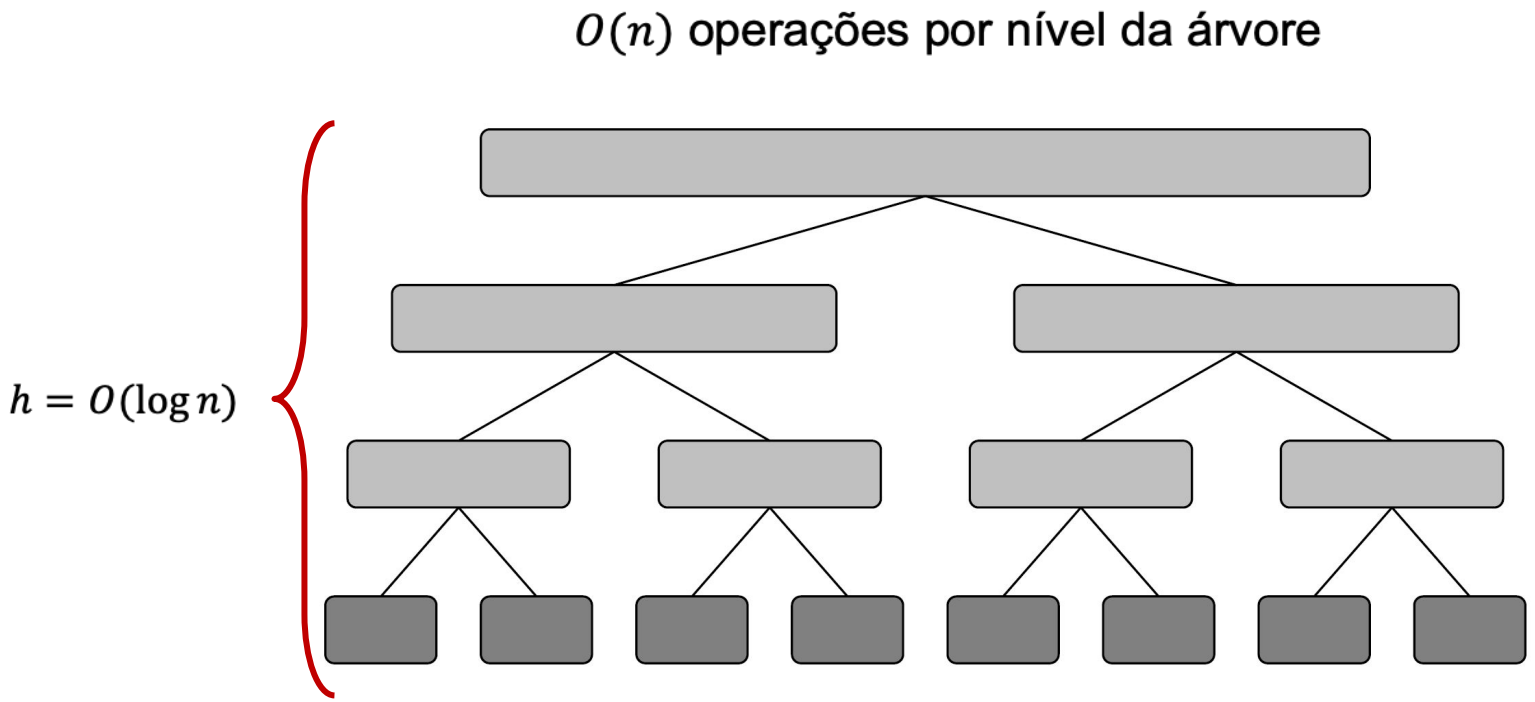
# Análise – Pior caso

- O pior caso ocorre quando, sistematicamente, o pivô é escolhido como sendo um dos extremos de um arquivo.
  - Por exemplo, se escolhe o pivô como 1º ou último elemento do vetor, e este está ordenado
- Isto faz com que o procedimento Ordena seja chamado recursivamente n vezes, eliminando apenas um item em cada chamada.

$$T(n) = n + T(n-1) \Rightarrow C(n) = O(n^2)$$

# Qual o Melhor caso?

- Quando o vetor é dividido em 2 partes iguais



# Análise – Melhor caso

- O melhor caso ocorre quando cada partição divide o conjunto em duas partes iguais.

$$T(n) = 2T(n/2) + n \Rightarrow C(n) = O(n \log n)$$

# Análise – Caso médio

- **Caso médio** de acordo com Sedgewick e Flajolet (1996, p. 17):

  $C(n) \approx 1{,}386n \log n - 0{,}846n,$

- Isso significa que em média o tempo de execução do Quicksort é ***O(n log n)***.

# Quicksort

- Vantagens:
  - É extremamente eficiente para ordenar arquivos de dados.
  - Necessita de apenas uma pequena pilha como memória auxiliar.
  - Requer cerca de *n log n* comparações em média para ordenar n itens.

- Desvantagens:
  - Tem um pior caso $O(n^2)$ comparações.
  - Sua implementação é muito delicada e difícil:
    - Um pequeno engano pode levar a efeitos inesperados para algumas entradas de dados.
  - O método não é **estável**.

# Melhorias no Quicksort

- Escolha do pivô: mediana de três
  - Evita o pior caso
- Utilizar um algoritmo simples (seleção, inserção) para partições de tamanho pequeno
- Quicksort não recursivo
  - Evita o custo de várias chamadas recursivas