# Ordenação: Quicksort

Algoritmos e Estruturas de Dados II

# Introdução

- É o algoritmo de ordenação interna mais rápido que se conhece para uma ampla variedade de situações.
- Provavelmente é o mais utilizado.
- A ideia básica é dividir o problema de ordenar um conjunto com n itens em dois problemas menores.
- Os problemas menores são ordenados independentemente.
- Os resultados são combinados para produzir a solução final.

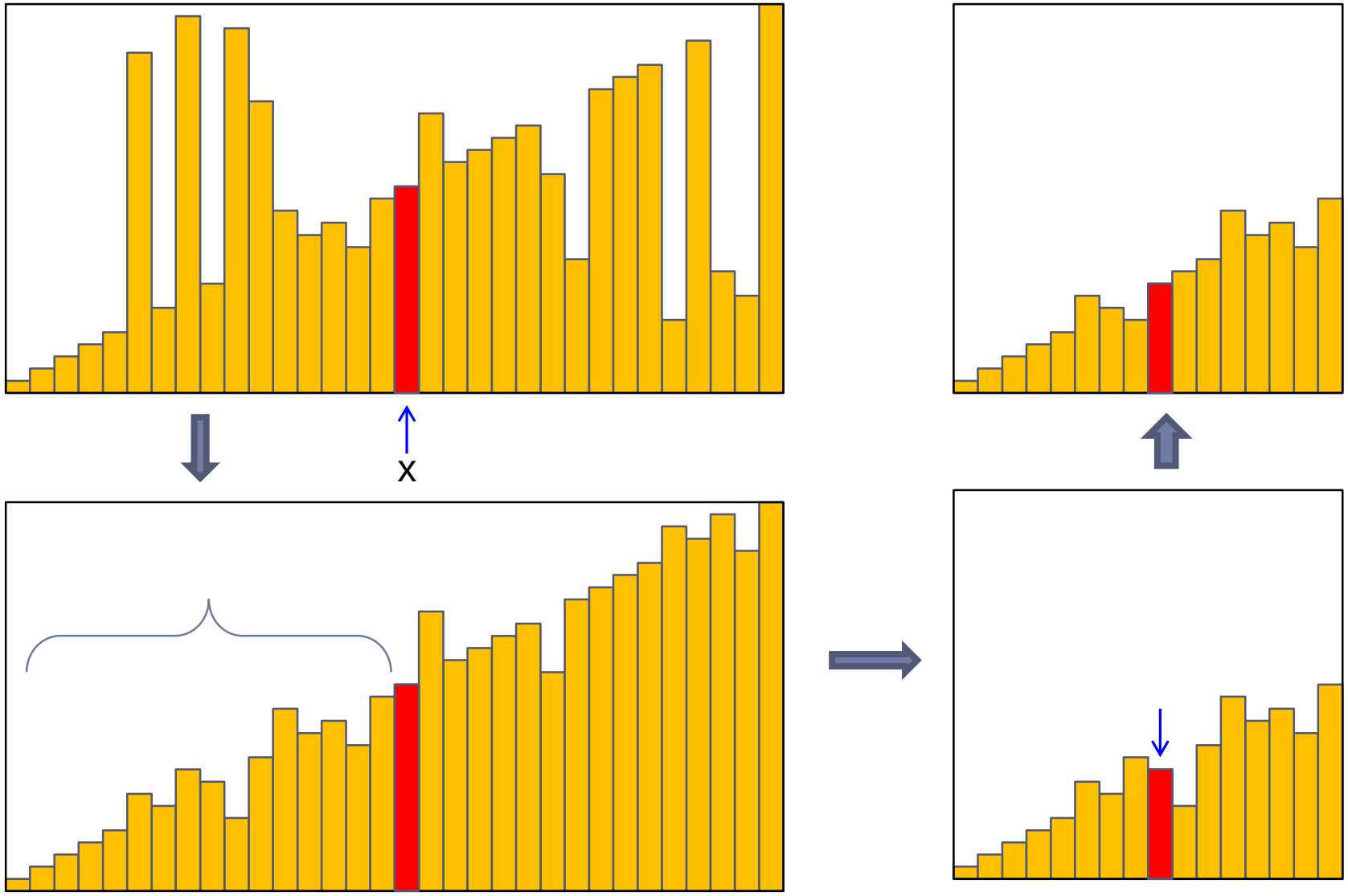
# Introdução

- A parte mais delicada do método é o processo de partição.

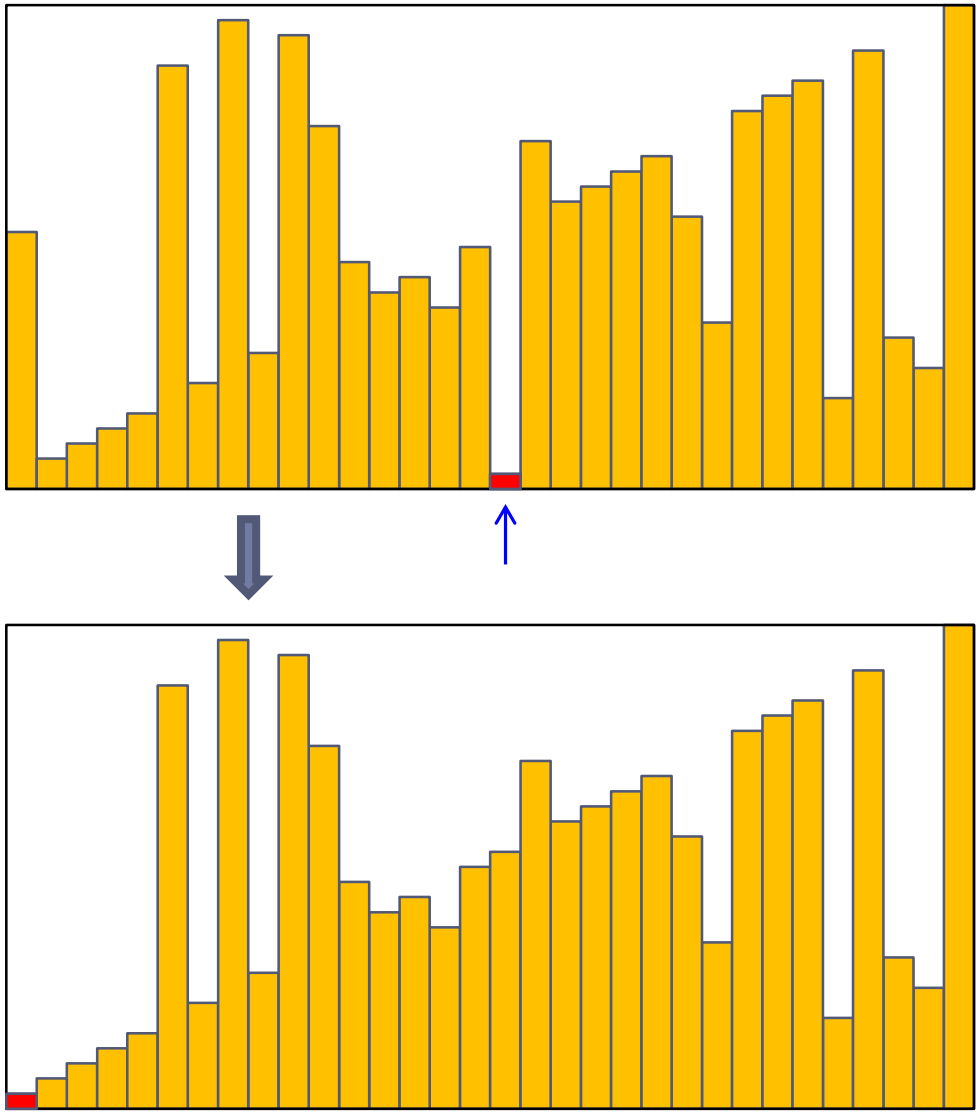
- O vetor A [Esq ... Dir] é rearranjado por meio da escolha arbitrária de um **pivô** x.

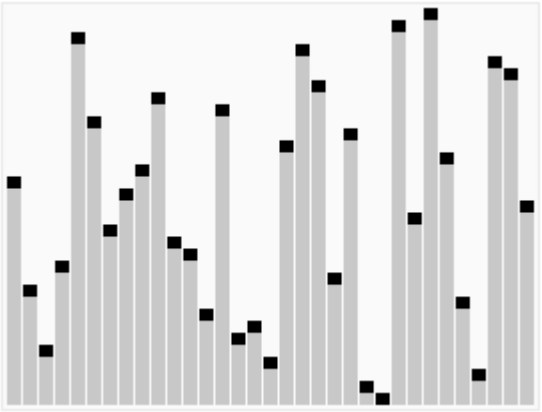
- O vetor A é particionado em duas partes:
  - Parte esquerda: chaves ≤ x.
  - Parte direita: chaves > x.

# Exemplo



# Dificuldades na escolha do pivô





# Algoritmo – Particionamento

- Algoritmo para particionamento:
  1. Escolha arbitrariamente um pivô x, troque com elemento mais à direita.
  2. Seta p (divisão da partição) para esquerda do vetor
  3. Percorre o vetor da esquerda para direita
     1. Se elemento A[i] ≤ pivô x
        1. Troca A[i] pelo elemento A[p]
        2. Incrementa  posição da divisão da partição, p
  4. Troca pivô da direita para atual divisão da partição
  5. Retorna a posição que divide a partição.

# Algoritmo – Após Partições

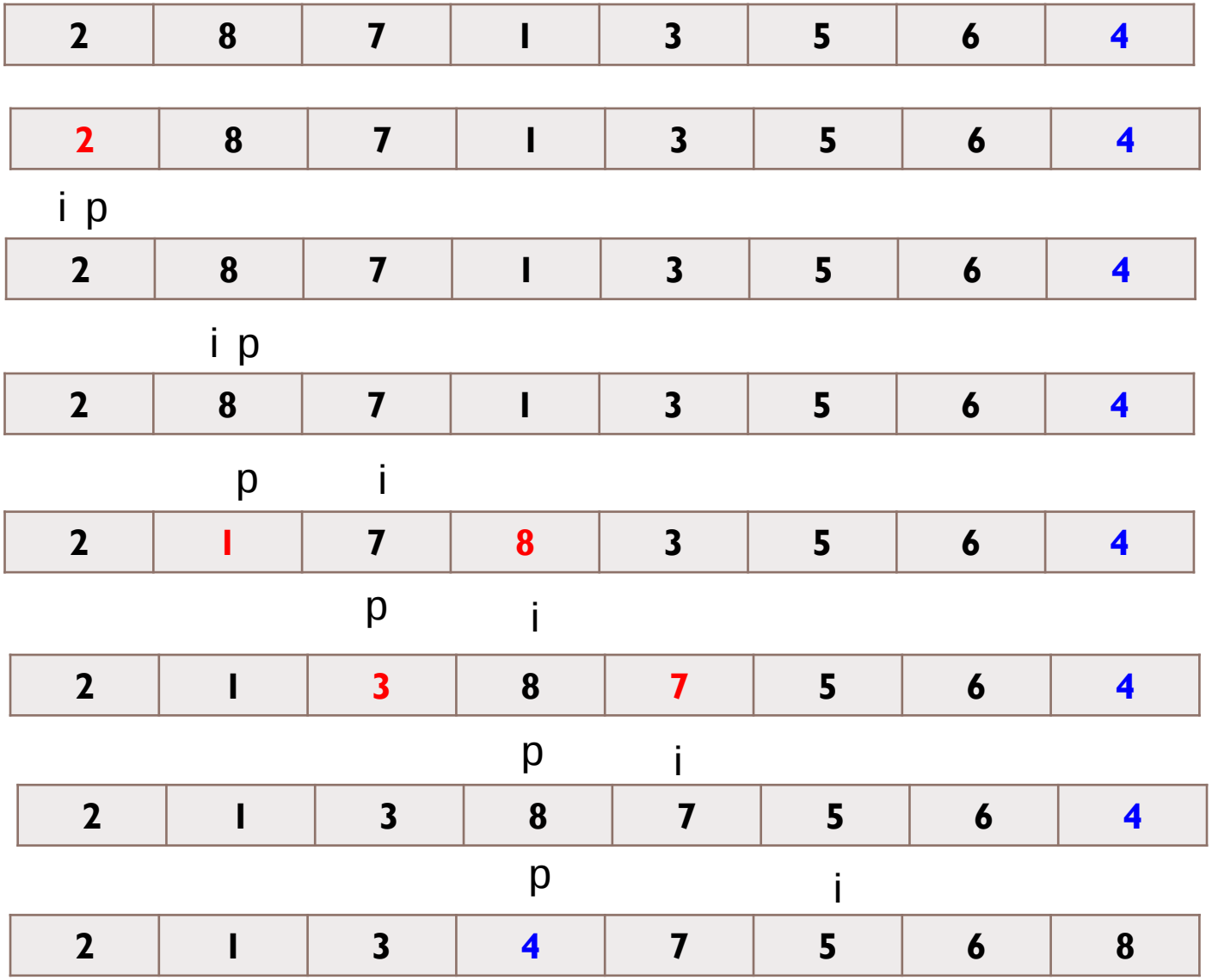
- Ao final de cada particionamento, os elementos do vetor A[esq … dir] estão
  - A[esq], A[esq +1], …, A[j]: menores ou iguais a x
  - A[i], A[i+1],…,A[dir]: maiores que x


- Uma vez obtidas as partições, cada uma deve ser ordenada – recursivamente.

# Particionamento

```
int particiona(Item a[], int e, int d) {
int i, p; /* contém índice da divisão da partição */
Item pivo = a[d];

   p = e;
   for (i = e; i < d; i++) {
      if (a[i].chave <= pivo.chave)
         Troca(a[i],a[p++]);
   }
   Troca(a[p],a[d]);

   return p;
}
```

# Particionamento - Exemplo



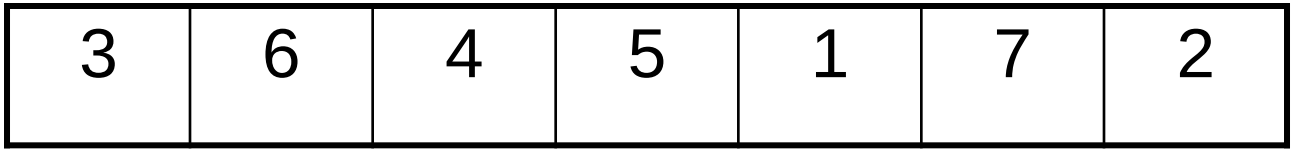
# Algoritmo – Recursão

- Cuidados para que a execução termine
  - Não chamar a recursão para apenas um elemento.
  - Chamar a recursão para partições menores que a atual.

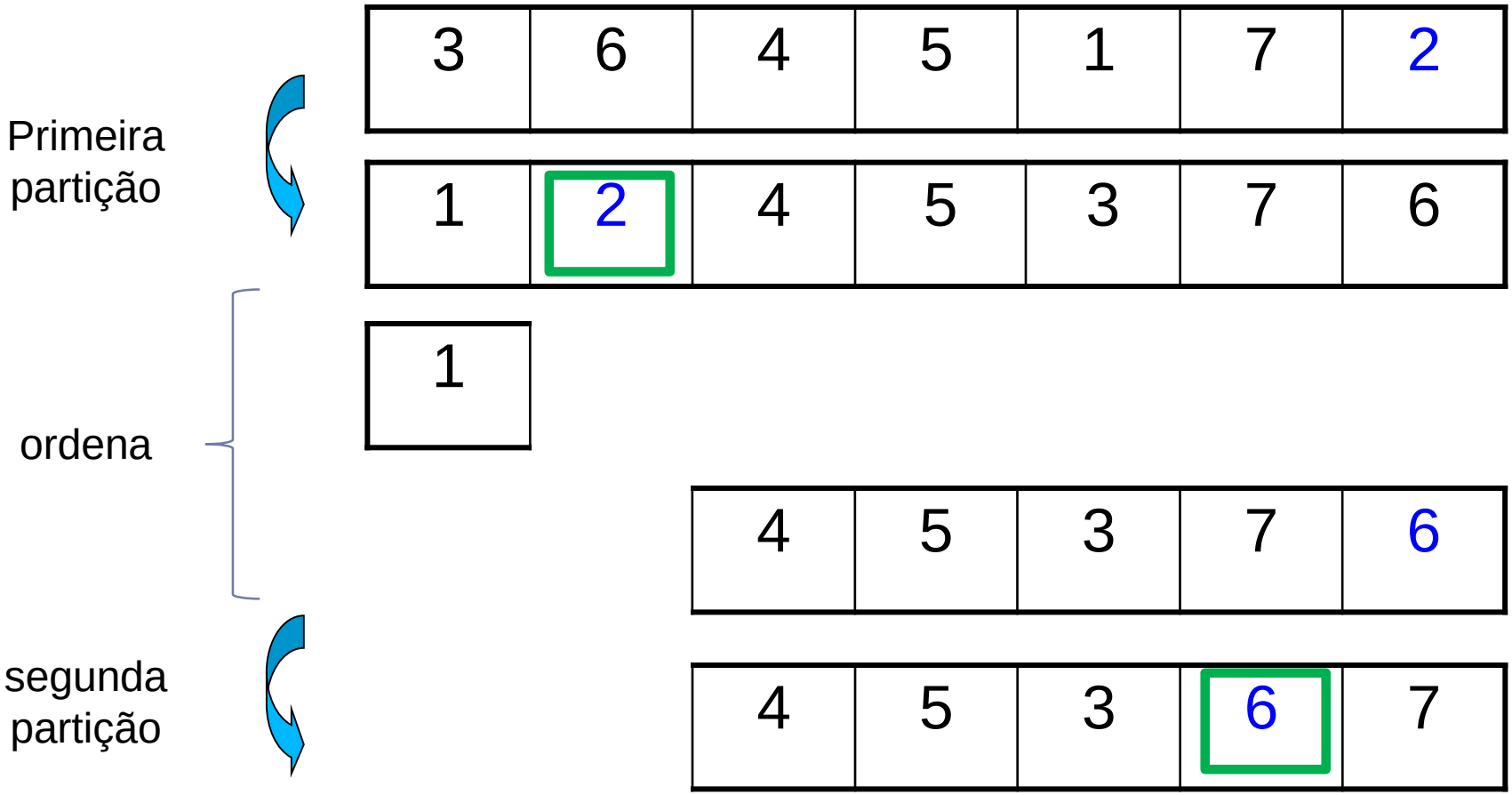
```
void ordena(Item a[], int e, int d) {
int i;
   if (d < e) return; /* condição de parada */
   i = particiona(a, e, d); /* i: índice do pivô */
   ordena(a, e, i-1);  /* ordena partição esquerda */
   ordena(a, i+1, d);  /* ordena partição direita */
}

void quicksort(Item a[], int n) {
   ordena(a, 0, n-1);
}
```

# Quicksort - Exemplo

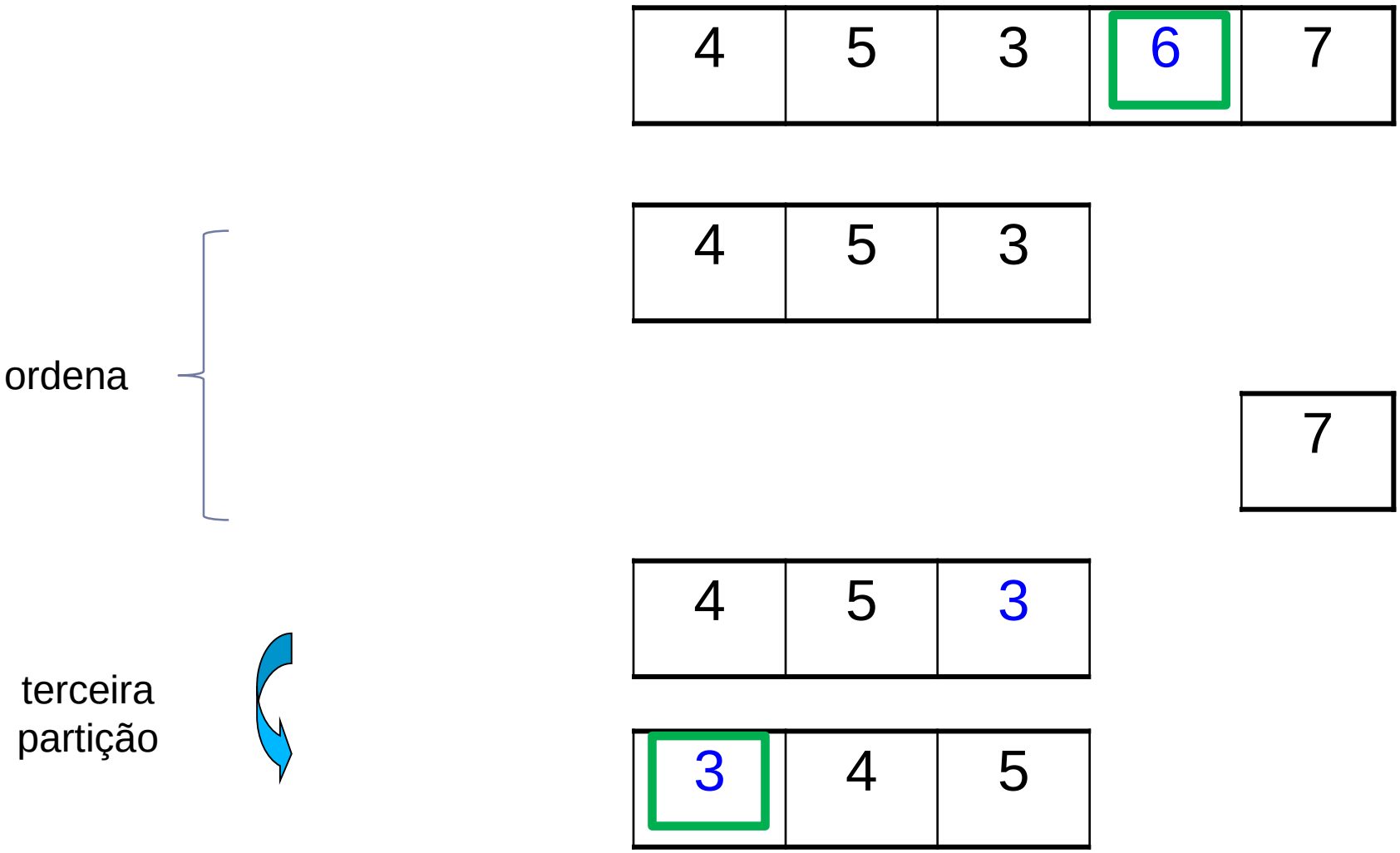
- O pivô x é escolhido como sendo A[d].
- Exemplo:



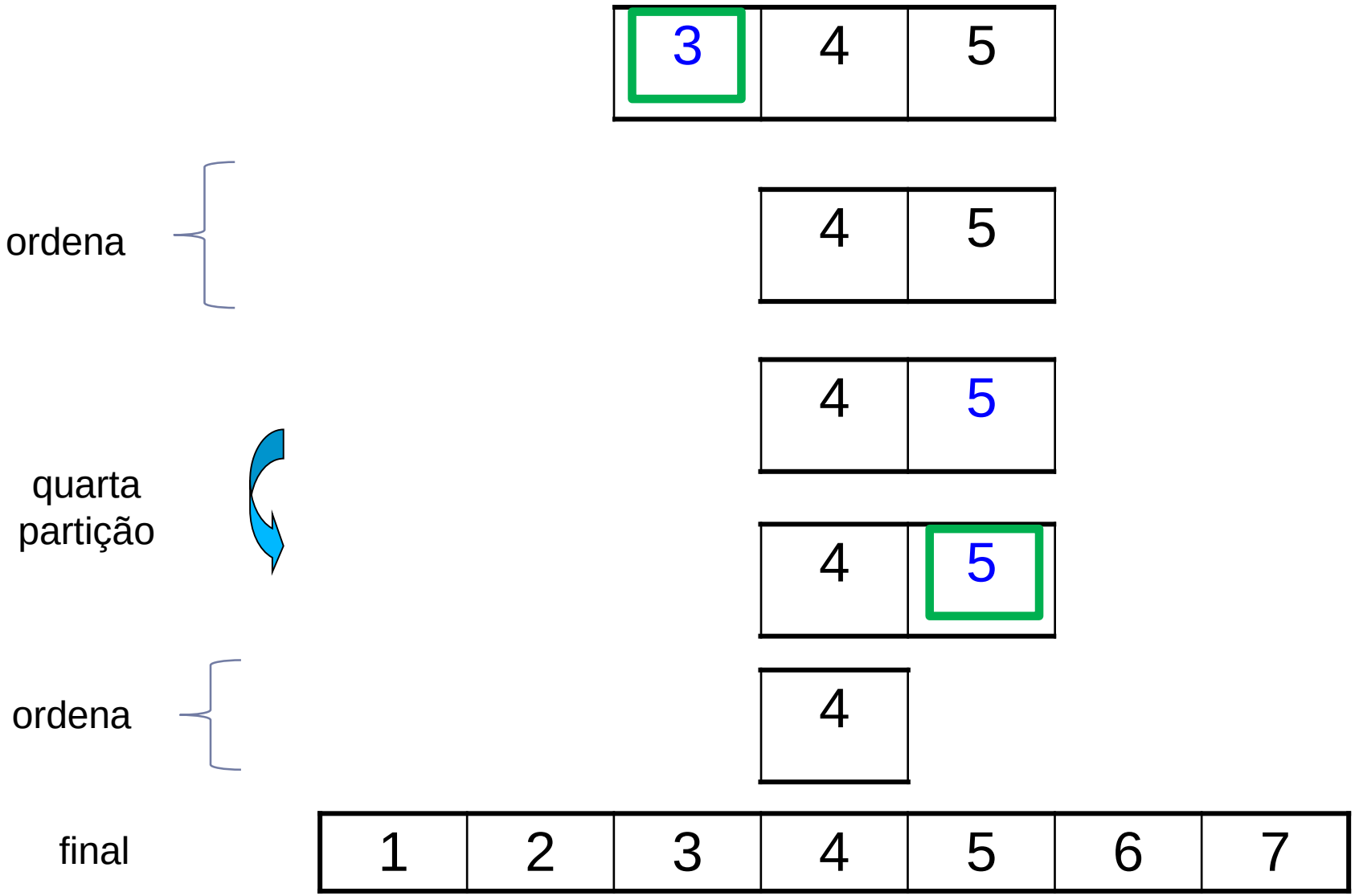
# Quicksort - Exemplo



# Quicksort - Exemplo



# Quicksort - Exemplo



# Quicksort

- **Características**
  - Qual o pior caso para o Quicksort? Qual sua ordem de complexidade?
  - Qual o melhor caso?
  - O algoritmo é estável?

# Quicksort – Análise

- Seja C(n) a função que conta o número de comparações.
- **Pior caso:**

  $C(n) = O(n^2)$

- O pior caso ocorre quando, sistematicamente, o pivô é escolhido como sendo um dos extremos de um arquivo já ordenado.
- Isto faz com que o procedimento Ordena seja chamado recursivamente n vezes, eliminando apenas um item em cada chamada.

# Quicksort – Análise

- **Melhor caso:**

  C(n) = 2C(n/2) + n = O(n log n)

- Esta situação ocorre quando cada partição divide o arquivo em duas partes iguais.

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

$$C(n) = n + \frac{1}{n}\sum_{i=0}^{n-1}(C(i) + C(n-i))$$

  C(n) ≈ 1,386n log n – 0,846n,

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

# 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

# Quicksort não recursivo

```
void quicksortNR(Item a[], int n) {
int i, e, d;
TipoPilha p;

   FPVazia(&p); Empilha(&p, n-1); Empilha(&p, 0);

   while (Vazia(&p) == 0) {
      e = Desempilha(&p); d = Desempilha(&p);
      i = particiona(a, e, d);

      /* partição esquerda */
      if (i-1 > e) { Empilha(&p, i-1); Empilha(&p, e);}
      /* partição direita */
      if (d > i+1) { Empilha(&p, d); Empilha(&p, i+1); }
   }
```

# 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**.

# Seleção usando Quicksort

- Objetivo
  - Encontrar o k-ésimo menor elemento sem ordenar todo o vetor.
- Algoritmo coloca o elemento ordenado na k-ésima posição do vetor.

```
void seleciona(Item a[], int e, int d, int k) {
int i;
   if (d <= e) return; /* condição de parada */
   i = particiona(a, e, d); /* i: índice do pivô */
   if (k < i) seleciona(a, e, i-1,k); /*k-th na esquerda*/
   if (k > i) seleciona(a, i+1, d);  /* k-th na direita */
}
```