Estruturas de Dados 2

Prof. Silvia Brandão

2024.1

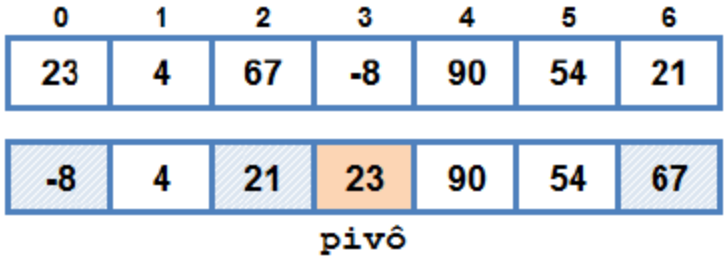
Métodos complexos de ordenação

- Os métodos eficientes são mais complexos nos detalhes, requerem um número menor de comparações.
- São projetados para trabalhar com uma quantidade maior de dados e possuem complexidade média igual a $C(n) = O(n \log n)$.

- **Exemplos**:
  - Quick sort, Merge sort, Shell sort, Heap sort, Radix sort, Gnome sort, Count sort, Bucket sort, Cocktail sort, Timsort.

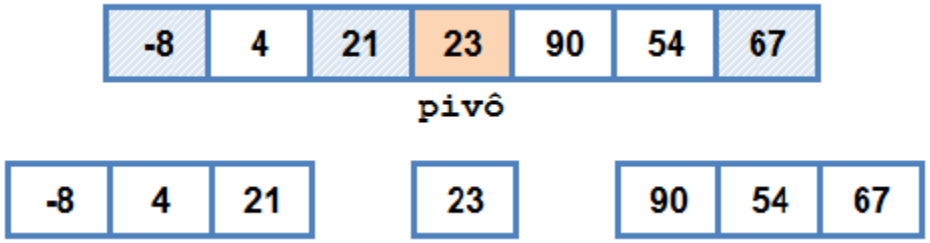
Algoritmo Quick Sort

- Criado por C. A. R. Hoare em 1960, é o método de ordenação interna mais rápido que se conhece para uma ampla variedade de situações.
- Provavelmente é o mais utilizado.
- Também conhecido como **ordenação por partição**
- É um algoritmo recursivo que usa a ideia de *dividir para conquistar* ao ordenar os dados
- Pode ser adaptado para realização de ordenação externa
- **Se baseia no problema da separação** *(partition subproblem)*



Algoritmo Quick Sort

- Funcionamento
  - **Um elemento é escolhido como pivô**
    - Valores menores do que o **pivô** ficam a esquerda; e valores maiores do que o **pivô** ficam a direita
    - Supondo o pivô na posição **x**, esse processo cria duas partições: **[0,...,x−1]** e **[x+1,...,n−1]**.
  - **Aplicar recursivamente a cada partição**
    - Até que cada partição contenha um único elemento



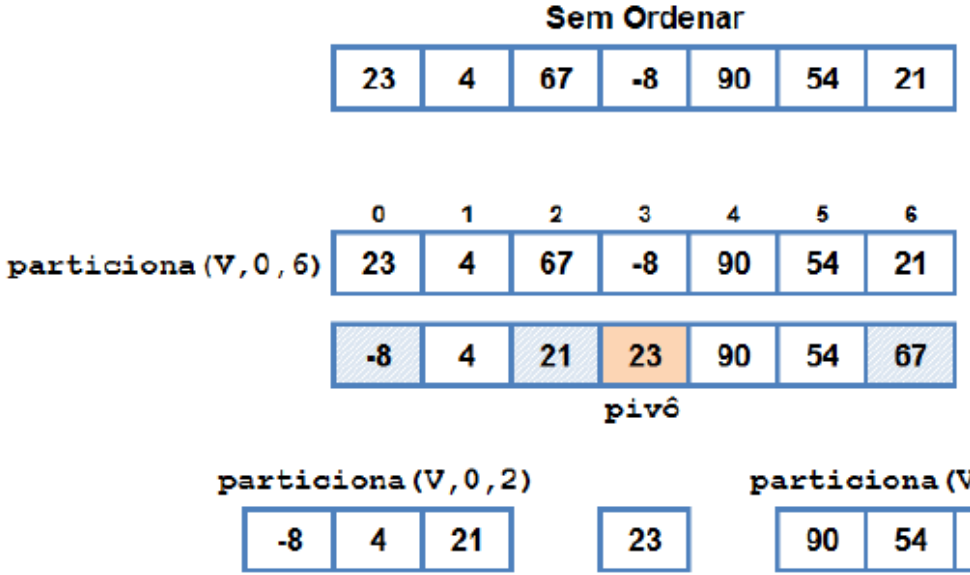
- Seu algoritmo usa 2 funções:
  - **quickSort**: divide os dados em arrays cada vez menores
  - **particiona**: calcula o pivô e rearranja os dados

Algoritmo Quick Sort

```
def quickSort(V, inicio, fim):
    if(fim > inicio):
        pivo = particiona(V, inicio, fim)
        quickSort(V, inicio, pivo-1)
        quickSort(V, pivo+1, fim)
```

Separa os dados em 2 partições

Chama a função para as 2 metades



Algoritmo Quick Sort

```
def particiona(V, inicio, final):
    esq = inicio
    dir = final
    pivo = V[inicio]
    while(esq < dir):
        while(esq <= final and V[esq] <= pivo):
            esq = esq + 1

        while(dir >= 0 and V[dir] > pivo):
            dir = dir - 1

        if(esq < dir):
            aux = V[esq]
            V[esq] = V[dir]
            V[dir] = aux

    V[inicio] = V[dir]
    V[dir] = pivo
    return dir
```

Avança posição da esquerda

Recua posição da direita

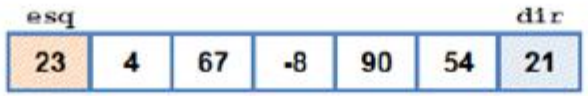
Trocar esq e dir

Simulando:

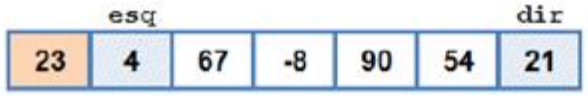
https://youtu.be/ywWBy6J5gz8

Passo a passo – função particiona

particiona(V,0,6)



V[esq] <= pivo)
incrementa esq

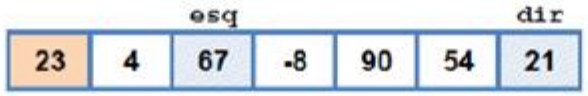


V[esq] <= pivo)
incrementa esq

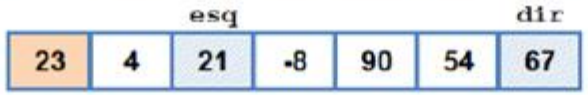
Primeira chamada

while(esq < dir)

V[esq] > pivo
comparar dir



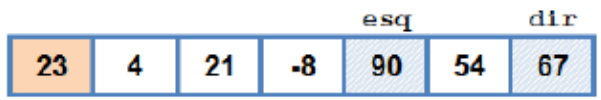
V[dir] < pivo
trocar esq e dir de lugar



esq < dir:
continua o while

V[esq] <= pivo)
incrementa esq

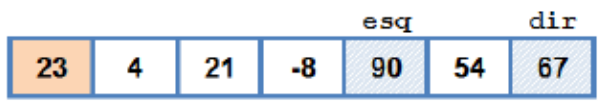
V[esq] <= pivo)
incrementa esq



V[esq] > pivo
comparar dir

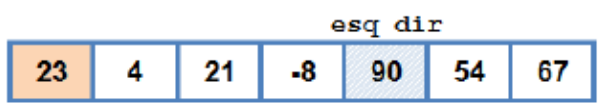
Segunda chamada

while(esq < dir)

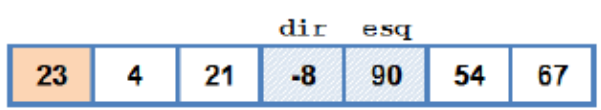


V[dir] > pivo)
decrementa dir

V[dir] > pivo)
decrementa dir

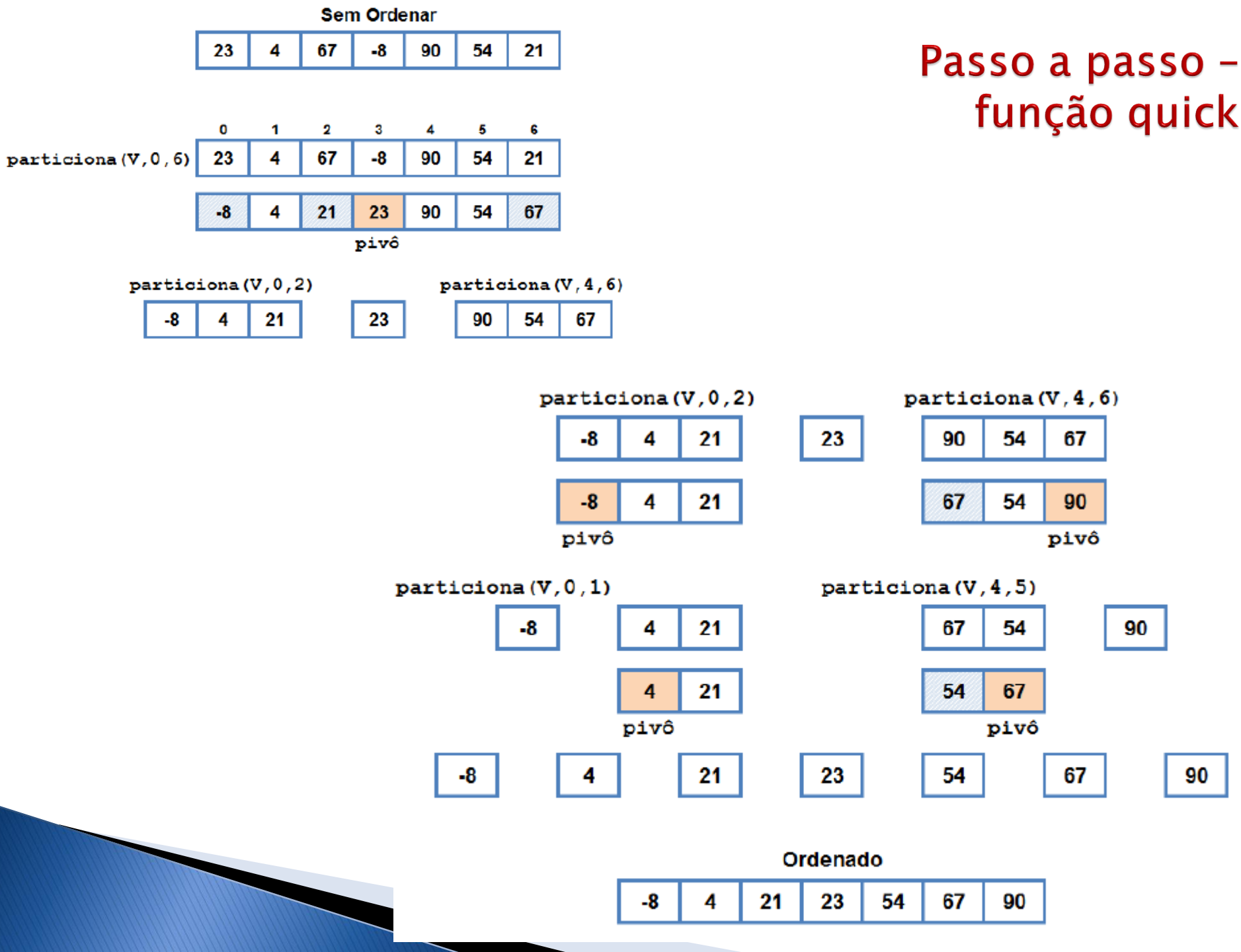


V[dir] > pivo)
decrementa dir



V[dir] < pivo e dir < esq
terminar o while

Passo a passo – função quick



Algoritmo Quick Sort

- **Complexidade**
  - Considerando um array com N elementos, o tempo de execução é:
    - **O(n log n),** melhor caso e caso médio;
    - **O(n^2),** pior caso; ocorre quando, sistematicamente, o pivô é escolhido como sendo um dos extremos de um arquivo já ordenado.
  - Em geral, é um algoritmo muito rápido. Porém, é lento em alguns casos especiais, como, por exemplo, quando o particionamento não é balanceado

Algoritmo Quick Sort

- **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.
  - Apesar de seu pior caso ser quadrático, costuma ser a melhor opção prática para ordenação de grandes conjuntos de dados.

Algoritmo Quick Sort

- **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.**
  - No caso de um particionamento não balanceado, o *insertion sort* acaba sendo mais eficiente que o *quick sort.*
  - O pior caso do *quick sort* ocorre quando o array já está ordenado, uma situação onde a complexidade é *O(n)* no *insertion sort*

Algoritmo Quick Sort

- **Desvantagens:**
  - Como escolher o pivô?
    - Existem várias abordagens diferentes
    - No pior caso, o pivô divide o array de **n** elementos em dois: uma partição com **n-1** elementos e outra com **0** elementos.
  - Particionamento não é balanceado
    - Quando isso acontece a cada nível da recursão, temos o tempo de execução de $O(n^2)$

TAREFA: implementação do método quick sort

Referência

- Viana, Daniel. **Conheça os principais algoritmos de ordenação**. https://www.treinaweb.com.br/blog/conheca-os-principais-algoritmos-de-ordenacao

- Sorting. https://www.cs.usfca.edu/~galles/visualization/Algorithms.html