Engenharia de Computação

Pesquisa e Classificação de Dados

Aula 6 – QuickSort

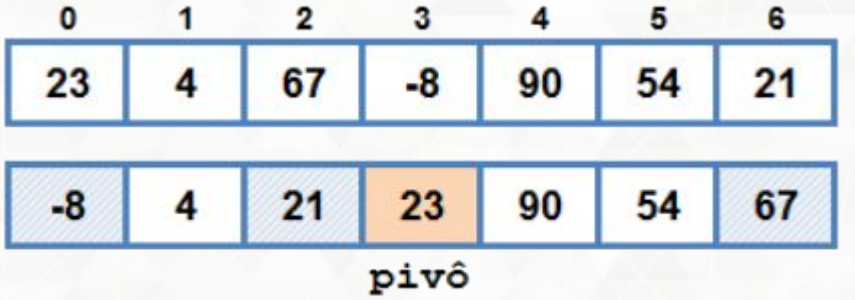
Prof. Muriel de Souza Godoi
muriel@utfpr.edu.br

QuickSort

- Também conhecido como ordenação por partição
  - É outro algoritmo recursivo que usa a ideia de dividir para conquistar para ordenar os dados
  - Se baseia no problema da separação
    - Em inglês, *partition subproblem*

QuickSort - Funcionamento

- Problema da separação
  - Em inglês, *partition subproblem*
  - Consiste em rearranjar o vetor usando um valor como pivô
    - Valores menores do que o pivô ficam a esquerda
    - Valores maiores do que o pivô ficam a direita

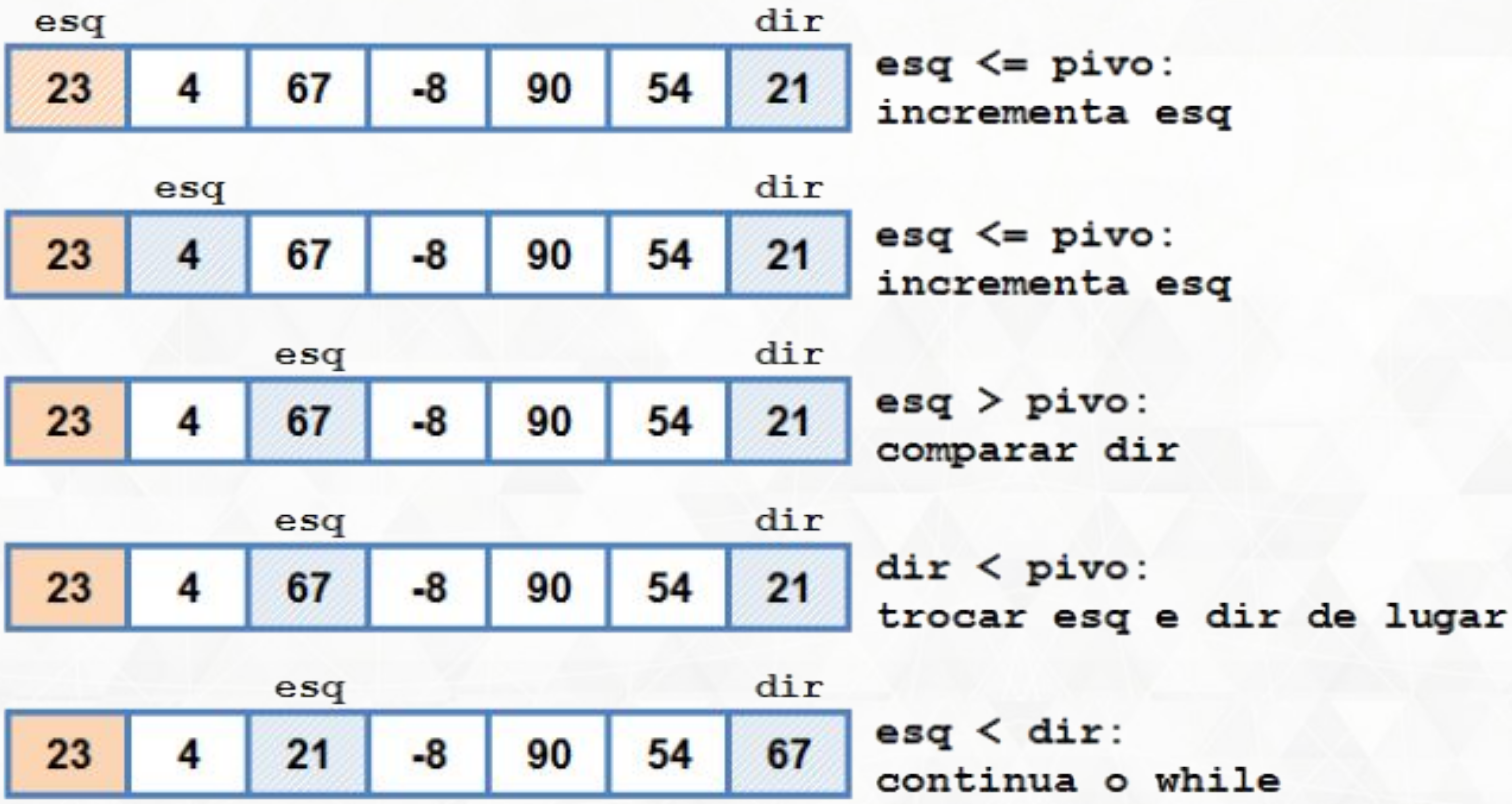


QuickSort - Funcionamento

- Um elemento é escolhido como **pivô**
- Valores menores do que o pivô são colocados antes dele e os maiores, depois
  - 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

QuickSort – Passo a passo

particiona(V,0,6)



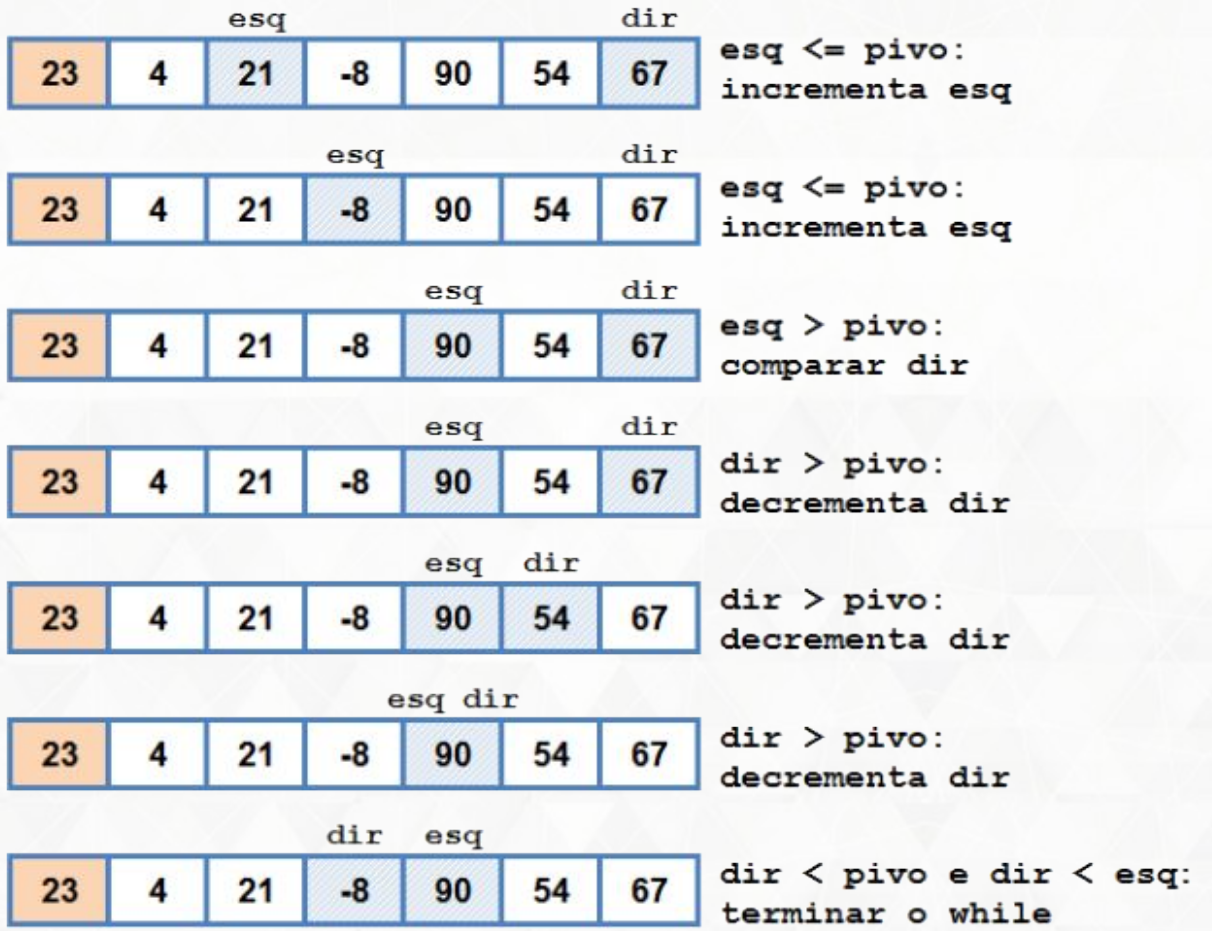
Primeira chamada

`while(esq < dir)`

QuickSort – Passo a passo

Segunda chamada

`while(esq < dir)`

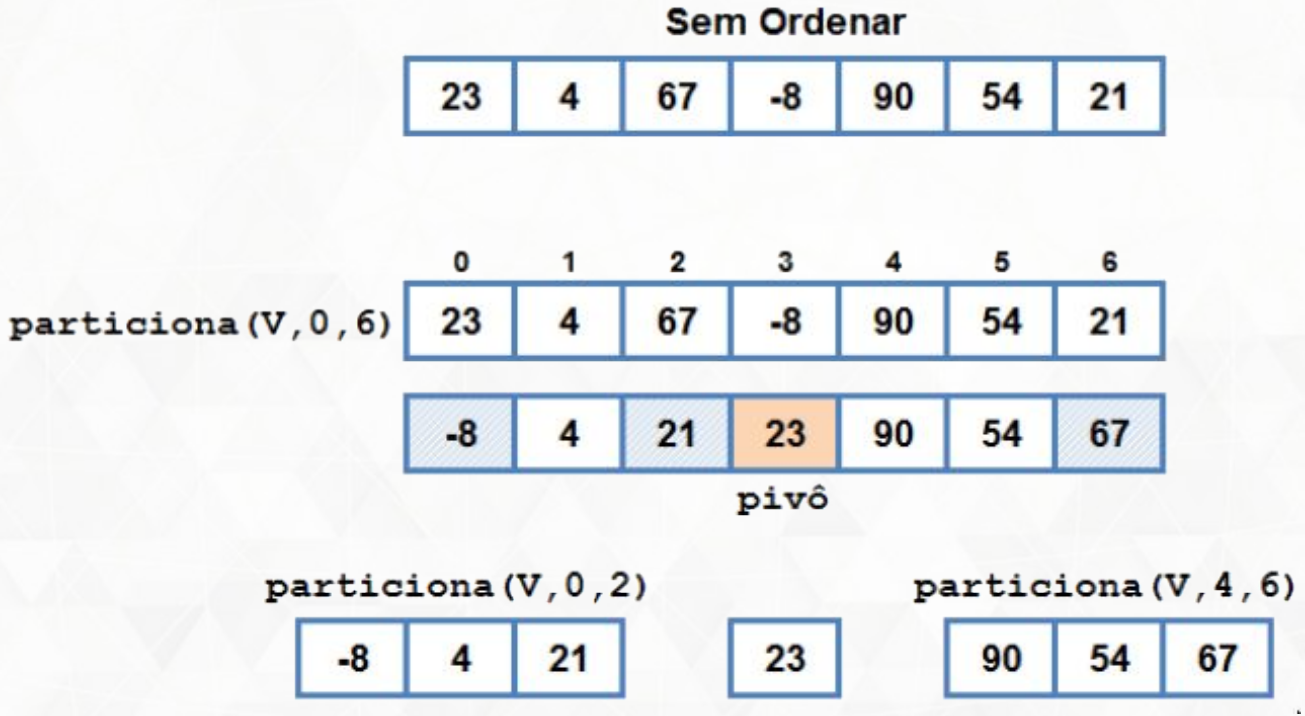


QuickSort – Passo a passo

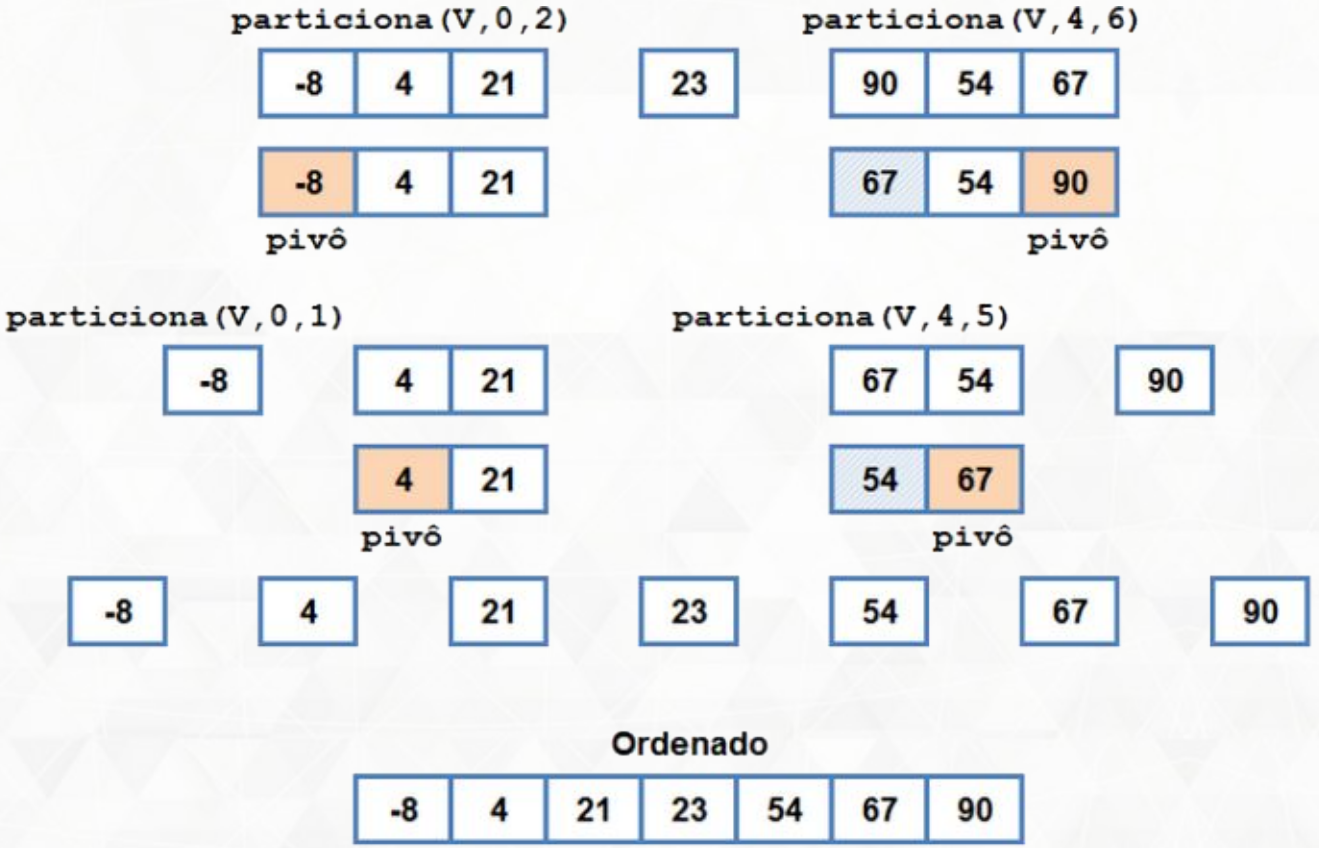
Trocar dir e inicio de lugar:
dir é o pivô

dir é o pivô

QuickSort – Passo a passo



QuickSort – Passo a passo



QuickSort - Pseudocódigo

- O algoritmo usa **2 funções**
  - **quickSort**: divide os dados em vetores cada vez menores
  - **particiona**: Elege um pivô e particiona de maneira na qual:
    - Todos os elementos menores que o pivô estão antes dele;
    - Todos os elementos maiores que o pivô estão depois dele;

QuickSort - Pseudocódigo

- **quickSort(v, inicio, fim)**
  - Se inicio < fim então:
    - pivo ← particiona(v, inicio, fim)
    - quickSort(v, inicio, pivo-1)
    - quickSort(v, pivo+1, fim)

QuickSort - Pseudocódigo

- **particiona(v, inicio, fim)**
  - esq ← inicio
  - dir ← fim
  - pivo ← v[inicio]
  - **Enquanto** esq < dir **faça**:
    - **Enquanto** v[esq] <= pivo && esq <= final **faça**:
      - incrementa esq
    - **Enquanto** v[dir] > pivo && dir >= inicio **faça**:
      - decrementa esq
    - **Se** esq < dir **então:**
      - troca v[esq] e v[dir]
  - troca v[dir] com v[inicio]
  - retorna dir

QuickSort - Complexidade

- Considerando um vetor com **n** elementos, o tempo de execução é:
  - **O(n log n)**, melhor caso e caso médio;
  - **O(n²)**, pior caso.
- Em geral, é algoritmo muito rápido.
  - Porém, é um algoritmo lento em alguns casos especiais
  - Por exemplo: quando o particionamento não é balanceado

QuickSort

- Desvantagens
  - Não é um algoritmo estável
- Como escolher o pivô?
  - Existem várias abordagens diferentes
  - No pior caso, o pivô divide o vetor de **n** 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 **$O(n^2)$**

QuickSort

- Desvantagens
  - No caso de um particionamento não balanceado, o InsertionSort acaba sendo mais eficiente que o QuickSort
    - O pior caso do QuickSort ocorre quando o vetor já está ordenado, uma situação onde a complexidade é O(N) no InsertionSort
- Vantagem
  - Apesar de seu pior caso ser quadrático, costuma ser a melhor opção prática para ordenação de grandes conjuntos de dados

Exercício

- Implemente o **QuickSort** em C considerando as seguintes assinaturas de função
  - **Função 1: quickSort**

```
/**
 * \brief Ordena o vetor usando QuickSort
 *
 * \param v vetor a ser ordenado
 * \param inicio índice do primeiro elemento do vetor
 * \param fim índice do último elemento do vetor
 *
 * Ordena o vetor usando o método QuickSort
 */
void quickSort(int *v, int inicio, int fim);
```

Exercício

- **Função 2: particiona**

```
/**
 * \brief Particiona um vetor
 *
 * \param v vetor a ser particionado
 * \param inicio índice do primeiro elemento do vetor
 * \param fim índice do último elemento do vetor
 *
 * \return índice do elemento pivô
 *
 * Particiona um vetor escolhendo o primeiro elemento (inicio)
 * como pivô. Ao final da função, todos os elementos menores
 * que o pivô devem estar antes dele e todos os elementos
 * maiores que o pivô devem estar após ele.
 */
int particiona(int *v, int inicio, int fim);
```