



# Universidade Federal do Maranhão

A Universidade que Cresce com Inovação e Inclusão Social

## Algoritmos de Ordenação cont.....

Estrutura de Dados II

Prof. João Dallyson

Email: [Joao.dallyson@ufma.br](mailto:Joao.dallyson@ufma.br)

# Aula passada.....

- **Introdução**
- **Conceitos básicos**
- **Classificação:**
  - Localização, uso de memória, estabilidade
- **Algoritmos:**
  - SelectionSort ( $O(n^2)$  , Não Estável)
  - InsertionSort ( $O(n^2)$ , Estável)
  - ShellSort (? , Não Estável)

# Divisão e Conquista

- **Dividir** o problema em um certo número de subproblemas
- **Conquistar** os subproblemas solucionando-os recursivamente. Se os tamanhos dos subproblemas são suficientemente pequenos, então, solucionar os subproblemas de forma simples.
- **Combinar** as soluções dos subproblemas na solução de problema original

# Merge Sort

- **Ordenação tipo dividir-para-conquistar**
  - Cria uma sequência ordenada a partir de duas outras também ordenadas
- **Passos:**
  - **Dividir:** dividir a sequência de  $n$  elementos a serem ordenados em duas subsequências de  $n/2$  elementos cada;
  - **Conquistar:** ordenar as duas subsequências recursivamente utilizando a ordenação por intercalação;
  - **Combinar:** intercalar as duas subsequências ordenadas para produzir a solução.

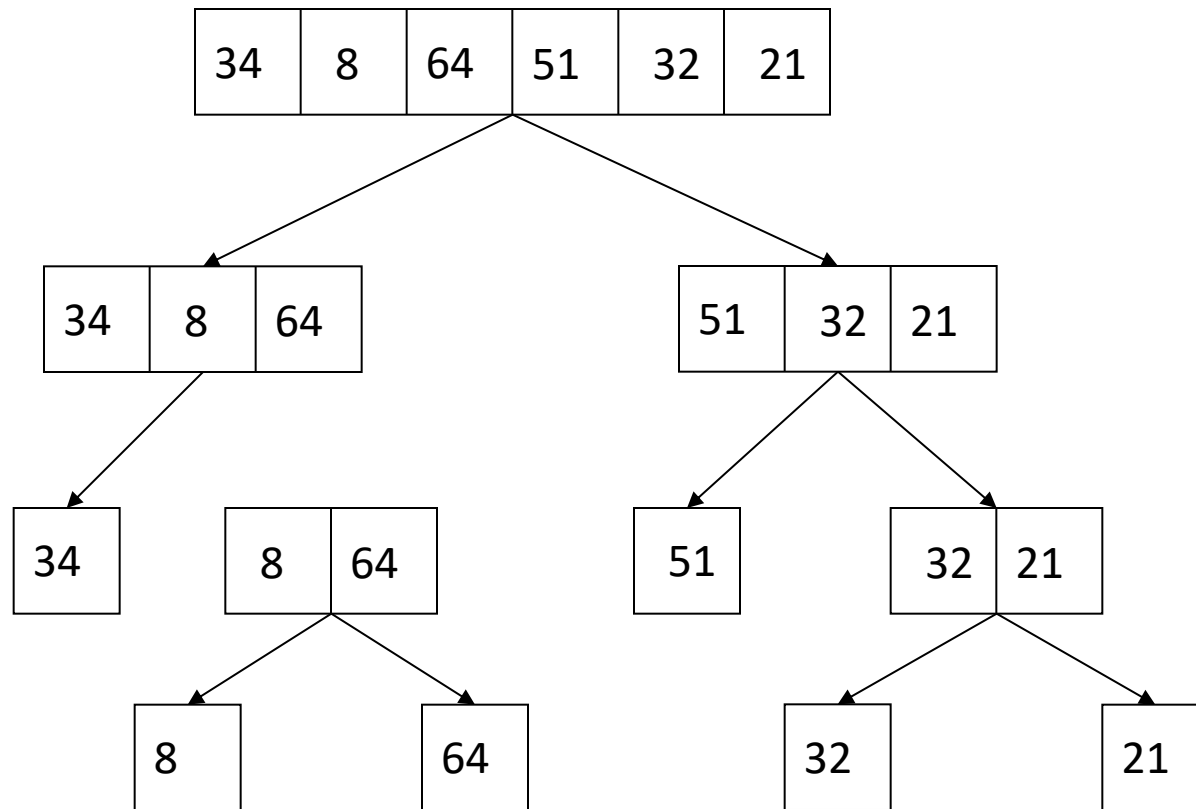
# MergeSort

5	3	7	1	0	8	5
---	---	---	---	---	---	---

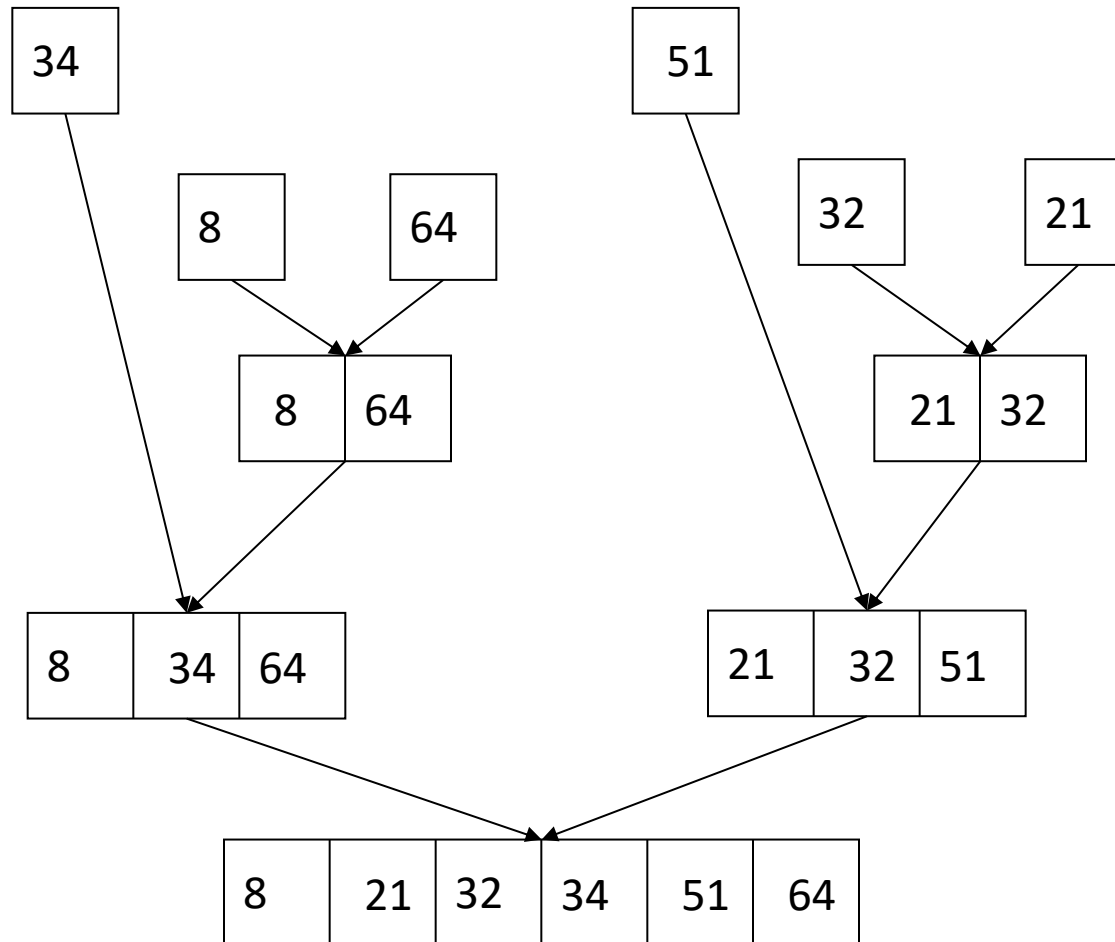
`mergesort([5, 3, 7, 1, 0, 8, 5])`

# Merge Sort

- **Dividir**

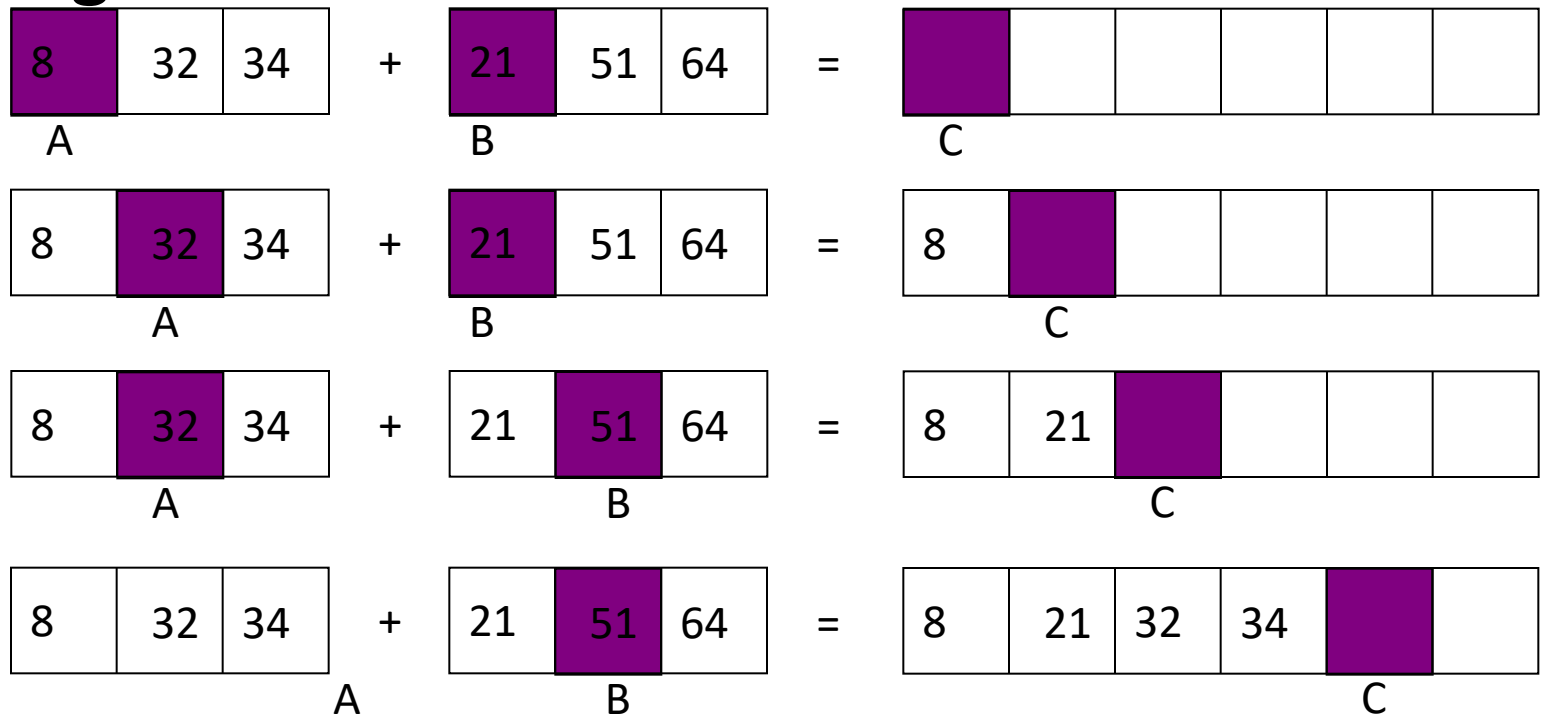


# Merge Sort



# Merge Sort

- Merge



↑  
Obs.: Quando um vetor acabar, copia o restante do outro



# Merge Sort

```
public static int[] MergeSort(int [] A) {  
    int [] Temp = new int[A.length];  
    return MergeMain(A, Temp, 0, A.length-1);  
}  
  
public static int[] MergeMain(int [] A, int [] T, int esq, int dir) {  
    int meio;  
  
    if (esq < dir) {  
        meio = (esq + dir)/2;  
        MergeMain(A, T, esq, meio);  
        MergeMain(A, T, meio + 1, dir);  
        Merge(A, T, esq, meio+1 , dir);  
    }  
  
    return A;  
}
```

# Merge Sort

```
private static void Merge (int [] A, int [] T, int esqPos, int dirPos, int dirFim) {
    int esqFim = dirPos - 1;
    int tempPos = esqPos;
    int numElem = dirFim - esqPos + 1;

    while (esqPos <= esqFim && dirPos <= dirFim) {
        if (A[esqPos] <= A[dirPos])
            T[tempPos++] = A[esqPos++];
        else
            T[tempPos++] = A[dirPos++];
    }
    // copia o resto da primeira parte
    while (esqPos <= esqFim)
        T[tempPos++] = A[esqPos++];
    // copia o resto da primeira parte
    while (dirPos <= dirFim)
        T[tempPos++] = A[dirPos++];

    // Copia vetor
    for (int i = 0; i < numElem; i++, dirFim--)
        A[dirFim] = T[dirFim];
}
```

# Merge Sort (Tempo de execução)

Para: (1)  $T(1) = 1$

(2)  $T(N) = 2T(N/2) + N$

\*\* Admitindo que  $N$  é potência de 2

Relação de recorrência

Uma solução

- 
1. Substitui continuamente a relação de recorrência do lado direito da equação.  
Primeiro substitui por  $N/2$

$$2T(N/2) = 2(2(T(N/4) + N/2)) = 4T(N/4) + N$$

Daí

$$T(N) = 4T(N/4) + 2N$$

2. Fazendo sequenciadamente para  $N/4$ ,  $N/8$  temos que:

$$T(N) = 2^k T(N/2^k) + kN$$

3. Sendo  $k = \log N$

$$T(N) = NT(1) + N \log N = N \log N + N$$

# Merge Sort (Tempo de execução)

- A expressão de recorrência é dada por  $T(n) = 2T(n/2) + n$ , sendo que  $2T(n/2)$  corresponde as duas chamadas recursivas e  $n$  ao tempo gasto na intercalação das duas metades do vetor.
- O tempo nas demais linha é  $O(1)$
- A recorrência acima pode ser solucionada pelo método mestre (caso 2) Se  $f(n) = \Theta(n^{\log_b a})$ , então  $T(n) = \Theta(n^{\log_b a} \lg n)$ .
- Sendo  $a=2$ ,  $b=2$ ,  $f(n) = n$

então

$$f(n) = \Theta(n^{\log_b a})$$

$$n = \Theta(n^{\log_2 2})$$

$$n = \Theta(n^1),$$

$$T(n) = \Theta(n \log n)$$

# Merge Sort

- **Tempo de execução médio proporcional a  $N \log N$** 
  - Pior caso igual
- **Prós**
  - Estável
  - Mais fácil de ser paralelizado
  - Implementação não-recursiva simples
- **Contras**
  - Uso de memória auxiliar do tamanho do vetor inicial

# Exercício

- 1) Dada a seguinte lista de números [21, 1, 26, 45, 29, 28, 2, 9, 16, 49, 39, 27, 43, 34, 46, 40]. Que resposta ilustra a lista a ser classificada após 3 chamadas recursivas para mergesort?
  - a) [16, 49, 39, 27, 43, 34, 46, 40]
  - b) [21,1]
  - c) [21, 1, 26, 45]
  - d) [21]
- 2) Considere a lista da questão anterior. Que resposta ilustra as primeiras duas listas a serem juntadas.
  - a) [21, 1] e [26, 45]
  - b) [[1, 2, 9, 21, 26, 28, 29, 45] e [16, 27, 34, 39, 40, 43, 46, 49]
  - c) [21] e [1]
  - d) [9] e [16]

# QuickSort

- Proposto por Hoare em 1960 e publicado em 1962.
- É o algoritmo de ordenação interna mais rápido que se conhece para uma ampla variedade de situações.
- Provavelmente é o mais utilizado.
- A idéia 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.
- <http://visualgo.net/sorting.html>

# QuickSort

- A parte mais delicada do método é relativa ao método particao.
- O vetor  $v[\text{esq}..\text{dir}]$  é rearranjado por meio da escolha arbitrária de um pivô  $x$ .
- O vetor  $v$  é particionado em duas partes:
  - A parte esquerda com chaves menores ou iguais a  $x$ .
  - A parte direita com chaves maiores ou iguais a  $x$ .



# Como escolher o pivô?

- **Primeiro elemento?**
- **Último elemento?**
- **Aleatório?**
- **Elemento Central?**
- **Elemento mediano entre primeiro, central e último?**

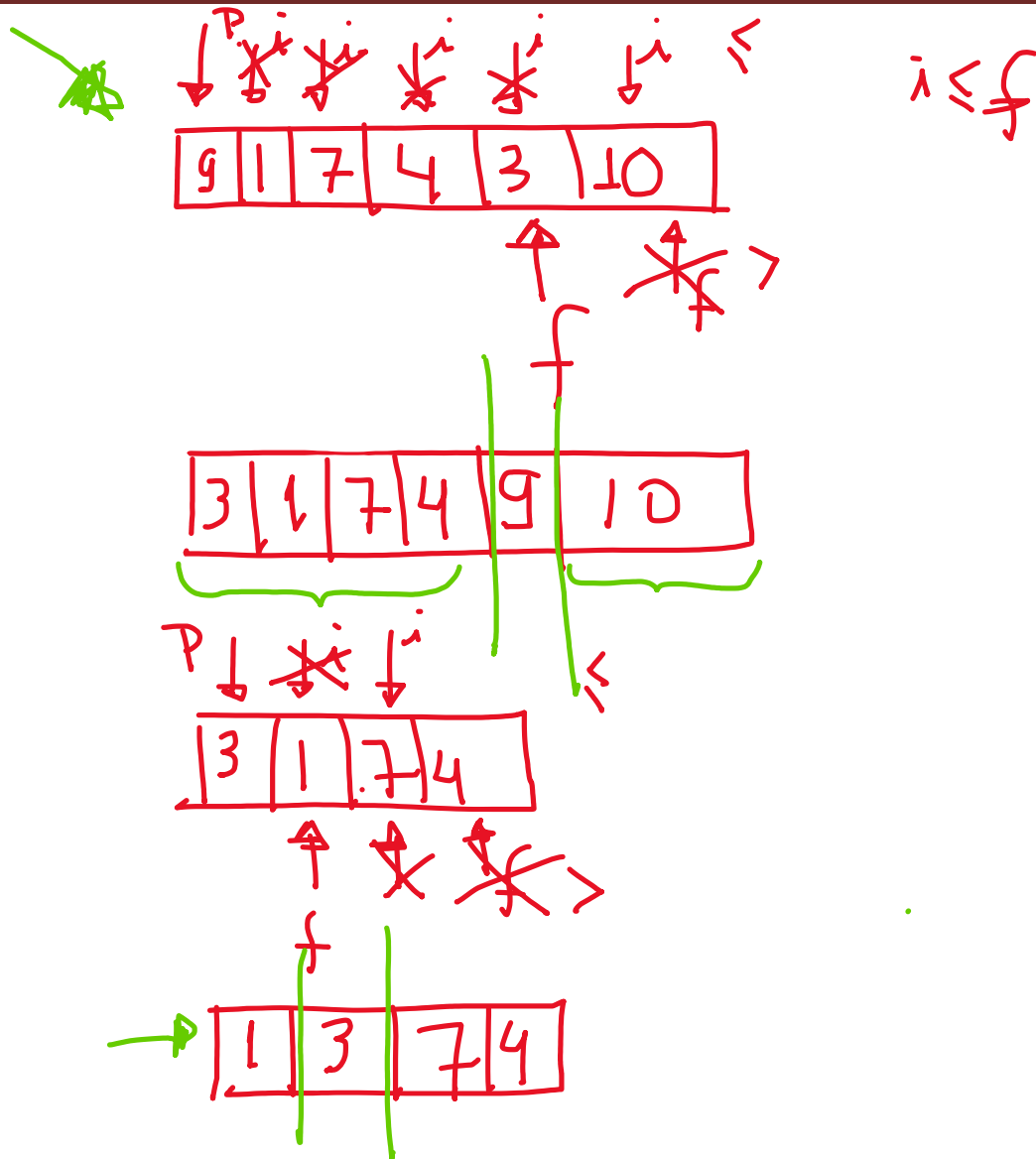
# QuickSort

- **Algoritmo para o particionamento:**
  - Escolha arbitrariamente um **pivô**  $x$ .
  - Percorra o vetor a partir da esquerda até que  $v[i] \geq x$ .
  - Percorra o vetor a partir da direita até que  $v[f] < x$ .
  - Troque  $v[i]$  com  $v[f]$ .
  - Continue este processo até os ponteiros  $i$  e  $f$  se cruzarem.
- **Ao final, o vetor  $v[\text{esq}..\text{dir}]$  está particionado de tal forma que:**
  - Os itens em  $v[\text{esq}], v[\text{esq} + 1], \dots, v[f]$  são menores ou iguais a  $x$ .
  - Os itens em  $v[i], v[i + 1], \dots, v[\text{dir}]$  são maiores ou iguais a  $x$ .

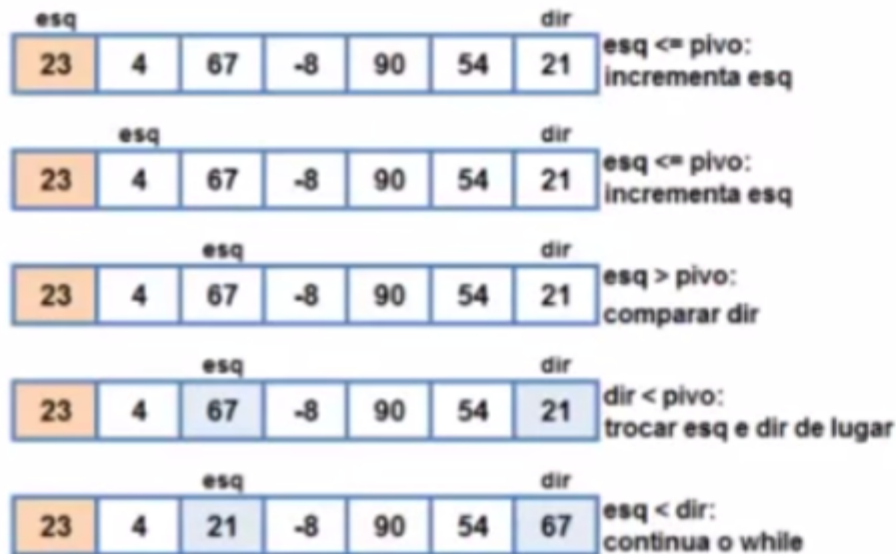
# QuickSort

**Unsorted Array**

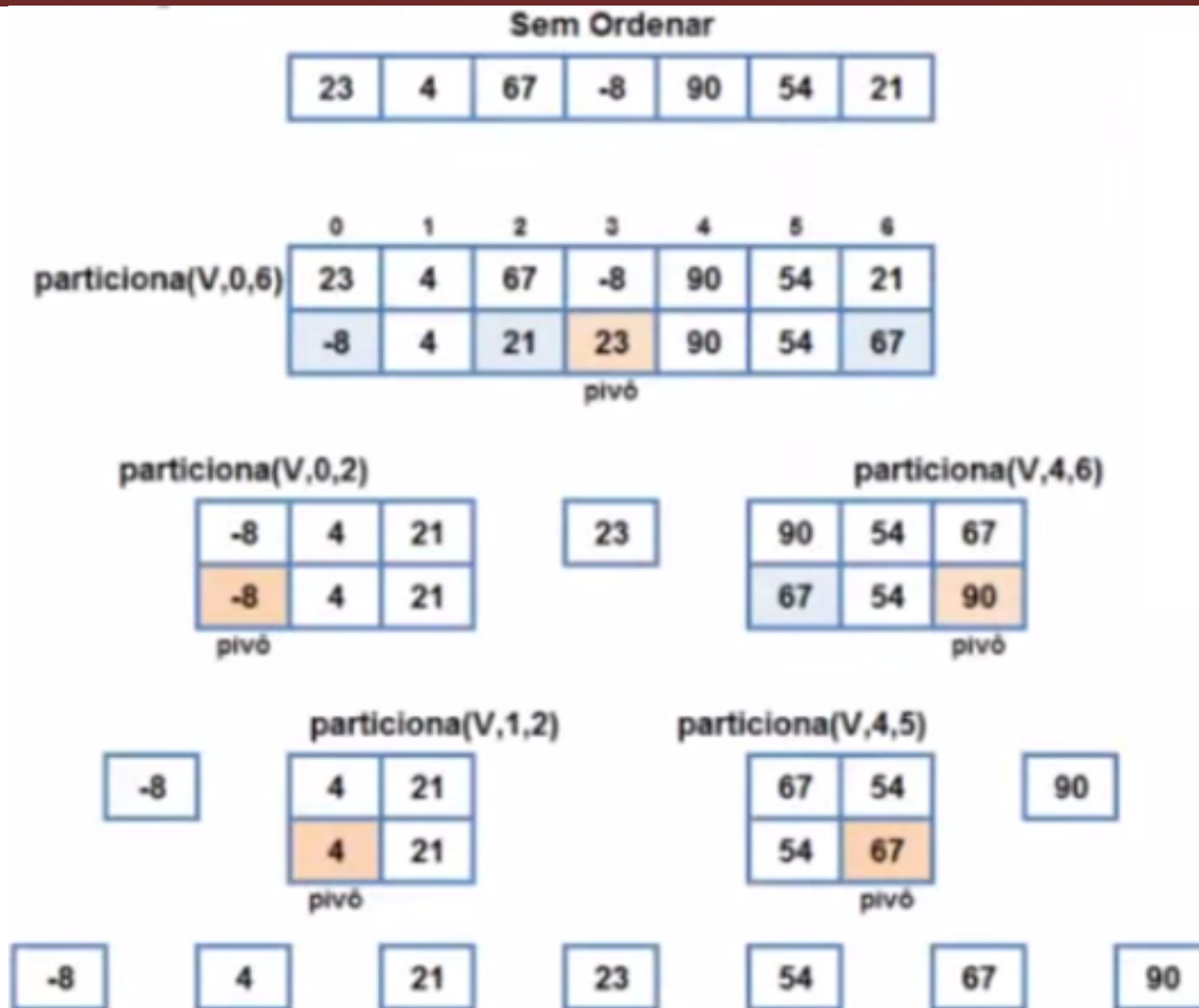




# QuickSort



# QuickSort



# QuickSort

QUICKSORT( $A, p, r$ )

**if**  $p < r$

**then**  $q \leftarrow \text{PARTITION}(A, p, r)$

        QUICKSORT( $A, p, q - 1$ )

        QUICKSORT( $A, q + 1, r$ )

PARTITION( $A, p, r$ )

$x \leftarrow A[r]$

$i \leftarrow p - 1$

**for**  $j \leftarrow p$  **to**  $r - 1$

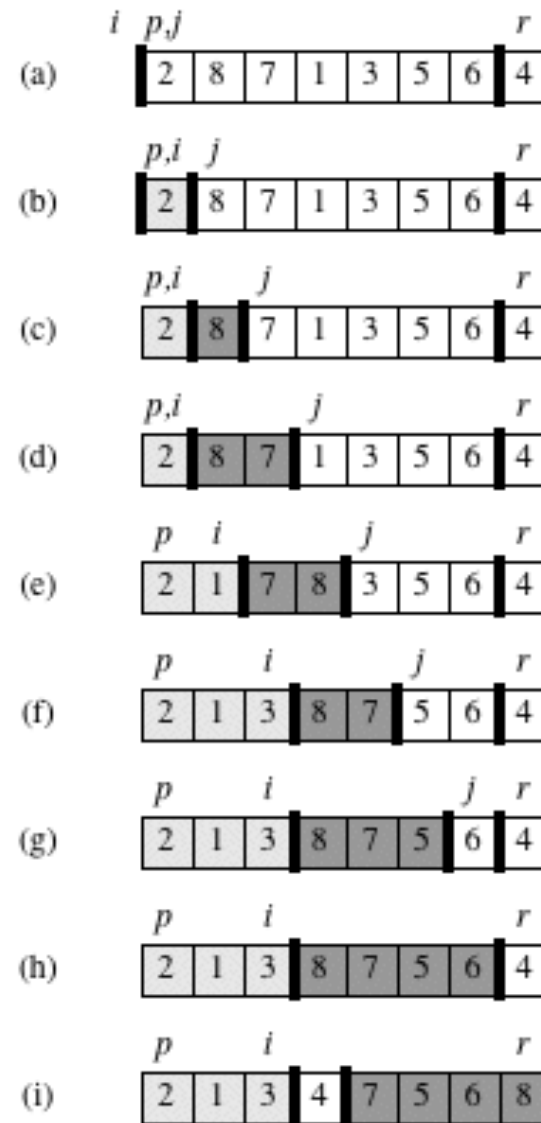
**do if**  $A[j] \leq x$

**then**  $i \leftarrow i + 1$

            exchange  $A[i] \leftrightarrow A[j]$

exchange  $A[i + 1] \leftrightarrow A[r]$

**return**  $i + 1$



# QuickSort

```
private static void quicksort(int[] vetor, int inicio, int fim)
{
    if (inicio < fim)
    {
        int posicaoPivo = particiona(vetor, inicio, fim);

        quicksort(vetor, inicio, posicaoPivo - 1);
        quicksort(vetor, posicaoPivo + 1, fim);
    }
}
```



# QuickSort

```
private static int particiona(int[] vetor, int inicio, int fim)
{
    int pivo = vetor[inicio];
    int i = inicio + 1, f = fim;

    System.out.println("-----");
    while (i <= f)
    {
        if (vetor[i] <= pivo)
            i++;
        else if (pivo < vetor[f])
            f--;
        else
        {
            int troca = vetor[i];
            vetor[i] = vetor[f];
            vetor[f] = troca;
            i++;
            f--;
        }
    }
    vetor[inicio] = vetor[f];
    vetor[f] = pivo;
    return f;
}
```

# Quicksort – Análise no pior caso

- **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.**
- **O pior caso pode ser evitado empregando pequenas modificações no algoritmo.**
- **Para isso basta escolher três itens quaisquer do vetor e usar a mediana dos três como pivô.**

# QuickSort - otimização

- **Mediana de três:**

- Para evitar o pior caso do quicksort, podemos escolher o pivô como a mediana de três elementos

T	V	Y	Z	S	X	U
T	V	Y	Z	S	X	U
T	S	U	Z	Y	X	V

- Aumentar o número de elementos considerados na mediana, ex: 5 ou 9, não ajuda muito.

# QuickSort – Análise no melhor caso

- **Análise**

- Melhor caso:

$$T(n) = n + T(n / 2) + T(n / 2)$$

$$T(n) = 2T(n / 2) + n.$$

- Esta situação ocorre quando cada partição divide o arquivo em duas partes iguais. Essa mesma recorrência já foi calculada para o MergeSort

$$T(n) = \Theta (n \cdot \log n).$$

- 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 (Random)

- Na análise do comportamento do caso médio fizemos uma suposição de que todas as permutações dos números de entrada são igualmente prováveis.
- Isso nem sempre é verdade
- Para corrigir este problema, vamos construir uma versão randomica do quicksort.
  - Nós temos que randomicamente particionar o vetor de entrada
  - Para isso, escolha o pivo randomicamente dentro dos elementos a serem ordenados
- <http://visualgo.net/sorting.html>

# QuickSort (Random)

RANDOMIZED-PARTITION( $A, p, r$ )

$i \leftarrow \text{RANDOM}(p, r)$

exchange  $A[r] \leftrightarrow A[i]$

**return** PARTITION( $A, p, r$ )

RANDOMIZED-QUICKSORT( $A, p, r$ )

**if**  $p < r$

**then**  $q \leftarrow \text{RANDOMIZED-PARTITION}(A, p, r)$

        RANDOMIZED-QUICKSORT( $A, p, q - 1$ )

        RANDOMIZED-QUICKSORT( $A, q + 1, r$ )

# 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**.

# Questão ENADE 2018 (Eng. Comp)

## QUESTAO 09

O MergeSort é um método de ordenação que combina dois vetores ordenados e cria um terceiro vetor maior também ordenado. O algoritmo abaixo apresenta essa ideia e combina os vetores `a[lo..mid]` e `a[mid+1..hi]` no vetor `a[lo..hi]`.

```
public class MergeSort {  
    private static Comparable[] aux;  
  
    public static void merge(Comparable[] a, int lo, int mid, int hi) {  
        int i = lo, j = mid+1;  
        for (int k = lo; k <= hi; k++)  
            aux[k] = a[k];  
        for (int k = lo; k <= hi; k++) {  
            if (i > mid)  
                a[k] = aux[j++];  
            else if (j > hi )  
                a[k] = aux[i++];  
            else if (aux[j].compareTo(aux[i]) < 0)  
                a[k] = aux[j++];  
            else  
                a[k] = aux[i++];  
        }  
    }  
  
    public static void sort(Comparable[] a) {  
        aux = new Comparable[a.length];  
        sort(a, 0, a.length - 1);  
    }  
  
    private static void sort(Comparable[] a, int lo, int hi) {  
        //implementação  
    }  
}
```

SEDGEWICK, R.; WAYNE, K. Algorithms. 4. ed. Boston: Addison-Wesley, 2011 (adaptado).



# Questão ENADE 2018 (Eng. Comp) cont...

Considerando o código apresentado, a implementação do protótipo do método sort da classe MergeSort é

**A**

```
if (hi == lo)
    return;
int mid = lo + (hi - lo)/2;
sort(a, lo, mid);
sort(a, mid, hi);
merge(a, lo, mid, hi);
```

**B**

```
if (hi > lo)
    return;
int mid = lo + (hi - lo)/2;
sort(a, lo, mid);
sort(a, mid, hi);
merge(a, lo, mid, hi);
```

**C**

```
if (hi <= lo)
    return;
int mid = lo + (hi - lo)/2;
sort(a, lo, mid);
sort(a, mid, hi);
merge(a, lo, mid, hi);
```

**D**

```
if (hi > lo)
    return;
int mid = lo + (hi - lo)/2;
sort(a, lo, mid);
sort(a, mid+1, hi);
merge(a, lo, mid, hi);
```

**E**

```
if (hi <= lo)
    return;
int mid = lo + (hi - lo)/2;
sort(a, lo, mid);
sort(a, mid+1, hi);
merge(a, lo, mid, hi);
```

# Questão ENADE 2021

## QUESTÃO 32

Existe um grande número de implementações para algoritmos de ordenação. Um dos fatores a serem considerados, por exemplo, é o número máximo e médio de comparações que são necessárias para ordenar um vetor com  $n$  elementos. Diz-se também que um algoritmo de ordenação é estável se ele preserva a ordem de elementos que são iguais. Isto é, se tais elementos aparecem na sequência ordenada na mesma ordem em que estão na sequência inicial. Analise o algoritmo abaixo, onde **A** é um vetor e “**i**, **j**, **lo** e **hi**” são índices do vetor:

```
algoritmo ordena(A, lo, hi)
    se lo < hi então
        p := particao(A, lo, hi)
        ordena(A, lo, p - 1)
        ordena(A, p + 1, hi)

algoritmo particao(A, lo, hi)
    pivot := A[hi]
    i := lo
    repita para j := lo até hi
        se A[j] < pivot entao
            troca A[i] com A[j]
            i := i + 1
    troca A[i] com A[hi]
    return i
```

Com relação ao algoritmo apresentado, avalie as afirmações a seguir.

- I. O algoritmo precisa de um espaço adicional  $O(n)$  para a pilha de recursão.
- II. O algoritmo apresentado é um algoritmo de ordenação recursivo e estável.
- III. O algoritmo precisa, em média, de  $O(n \log n)$  comparações para ordenar  $n$  itens.
- IV. O uso do primeiro elemento do vetor como “*pivot*” é mais eficiente que usar o último.

É correto apenas o que se afirma em

- A** I e III.
- B** II e IV.
- C** III e IV.
- D** I, II e III.
- E** I, II e IV.

# Exercício 1

**1) Dada a seguinte lista de números [14, 17, 13, 15, 19, 10, 3, 16, 9, 12] qual será o conteúdo da lista após a segunda partição do algoritmo quicksort.**

a) [9, 3, 10, 13, 12]

b) [9, 3, 10, 13, 12, 14]

c) [9, 3, 10, 13, 12, 14, 17, 16, 15, 19]

d) [9, 3, 10, 13, 12, 14, 19, 16, 15, 17]

# Exercício 2

**2) Dada a seguinte lista de números [1, 20, 11, 5, 9, 16, 14, 13, 19] qual deve ser o valor do pivô usando o método da mediana de 3.**

- a) 1
- b) 9
- c) 16
- d) 19
- e) 11

# Exercício 3

3) Qual dos seguintes algoritmos sempre garantem no pior caso  $O(n \log n)$

- a) Shell Sort
- b) Quick Sort
- c) Merge Sort
- d) Insertion Sort

# Referências

## Básica

- CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Algoritmos: Teoria e Prática. Editora Campus, 2002
- Ziviani, N. Projeto de Algoritmos Com Implementações em Pascal e C, Cengage Learning, 2004.

## Complementar

- TENENBAUM, Aaron; LANGSAM, Yedidiah; AUGENSTEIN, Moshe J. *Estruturas de dados usando C. São Paulo: Makron Books, 1995. ISBN: 9788534603485*
- ASCENCIO, Ana Fernanda Gomes; ARAUJO, Graziela Santos. Estruturas de Dados: Algoritmos, análise da complexidade e implementações em Java e C/C++. Pearson Prentice Hall, 2010
- DROZDEK, Adam. Adam Drozdek. Data Structures and Algorithms in Java. 2. Cengage Learning. 2004. 2. Cengage Learning. 2004
- GOODRICH, Michael T. Estruturas de dados e algoritmos em java. 4 ED. Porto Alegre: Bookman, 2007. 600.
- SKIENA, Steven S.. **The Algorithm Design Manual**. 2. Springer-Verlag. 2008
- Notas de aula: prof. Ítalo Cunha – UFMG. 2012.

# Perguntas....

