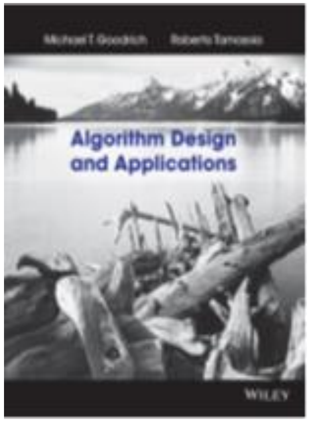
Unidade 9 – Merge Sort e Quick Sort

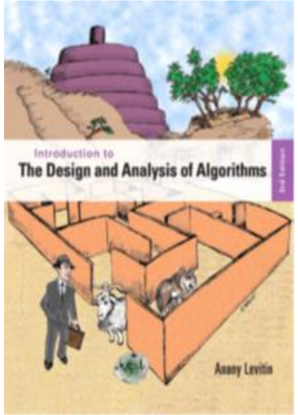
Prof. Aparecido V. de Freitas
Doutor em Engenharia
da Computação pela EPUSP
aparecidovfreitas@gmail.com

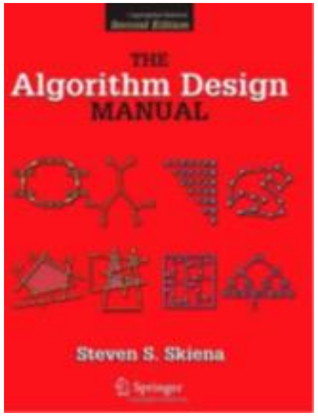
QualitSys

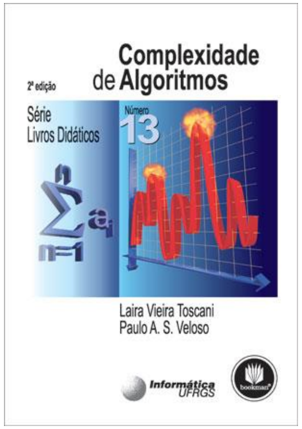
Bibliografia

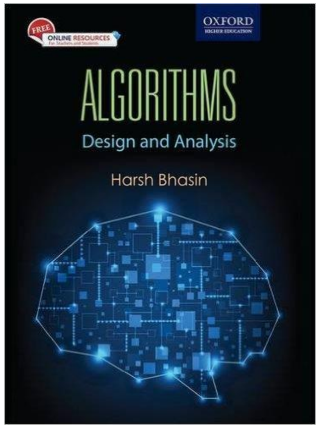
- Algorithm Design and Applications – Michael T. Goodrich, Roberto Tamassia, Wiley, 2015
- Introduction to the Design and Analysis of Algorithms – Anany Levitin, Pearson, 2012
- The Algorithm Design Manual – Steven S. Skiena, Springer, 2008
- Complexidade de Algoritmos – Série Livros Didáticos – UFRGS
- Algorithms – Design and Analysis – Harsh Bhasin – Oxford University Press - 2015





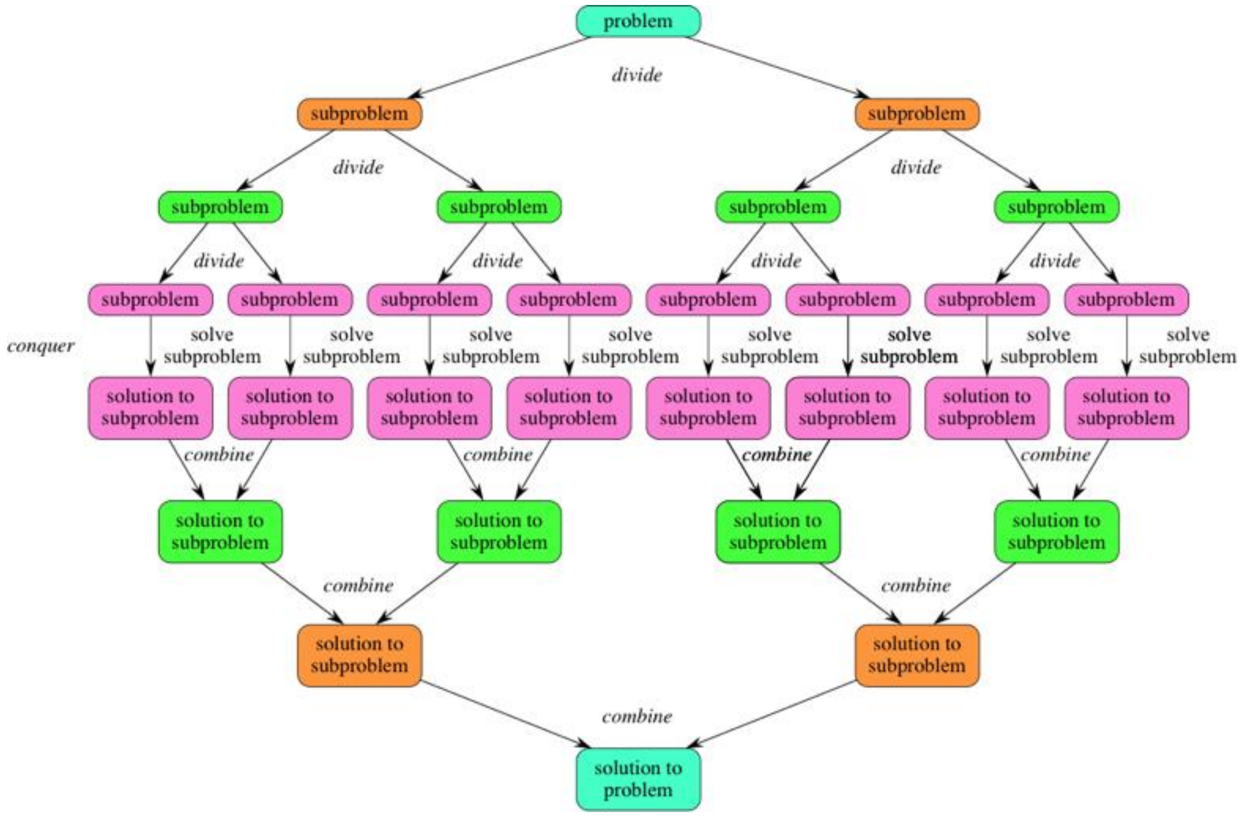






Introdução

- O algoritmo Merge Sort utiliza a técnica de programação Divisão-e-Conquista.

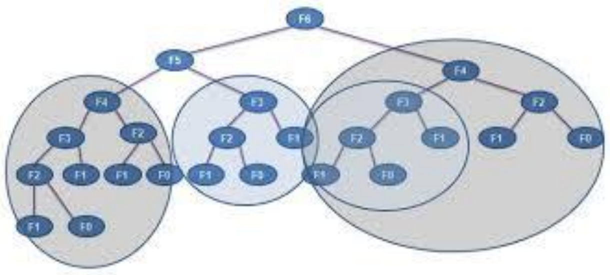


Técnica Divisão e Conquista

- É provavelmente uma das mais conhecidas técnicas de projeto de algoritmos;
- Algoritmos **divide-and-conquer** funcionam de acordo com um plano geral.

Técnica Divisão e Conquista

- Dado um problema com uma entrada grande, quebra-se a entrada em porções menores (**DIVISÃO**);
- Resolve-se cada porção separadamente (**CONQUISTA**);
- Combinam-se os resultados.

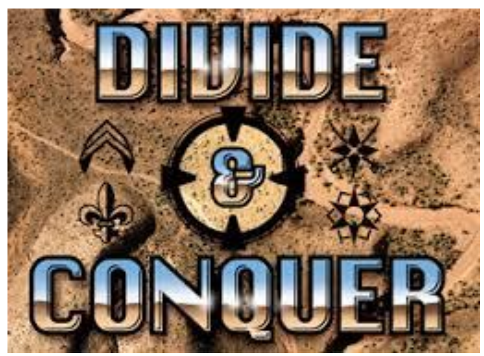


Algoritmos Divisão e Conquista

- São, em geral, recursivos;
- Requer um número menor de acessos à memória;
- São altamente paralelizáveis. Se existirem vários processadores disponíveis, a estratégia propicia eficiência.

Quando empregar Divisão e Conquista ?

- Quando for possível decompor-se a instância em sub-instâncias;
- Problemas onde um grupo de operações são correlacionadas ou repetidas.



Passos do Algoritmo Divisão e Conquista

PROBLEMA

Dividir

Problema | Problema | Problema | Problema | Problema

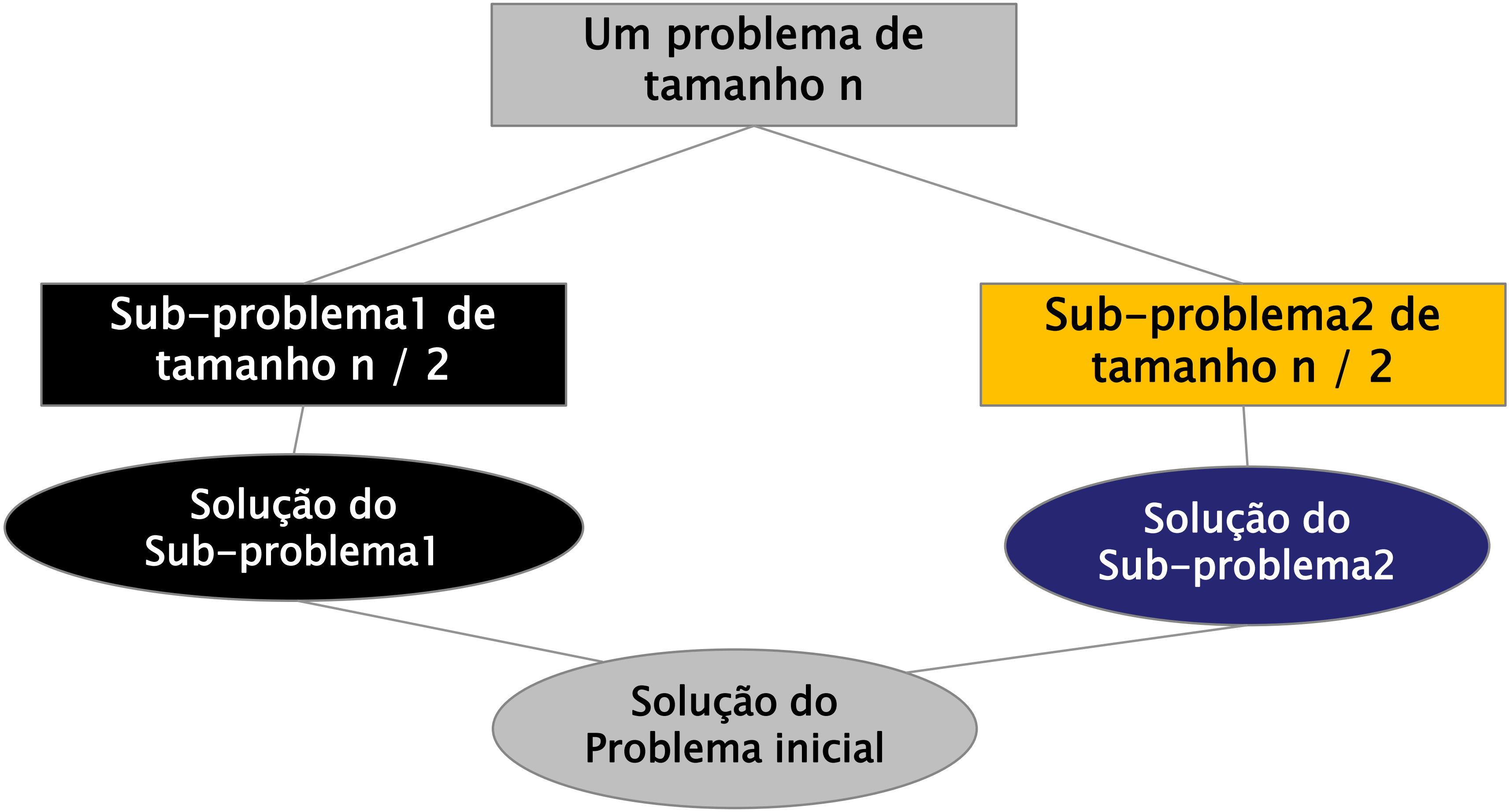
Conquistar

Solução | Solução | Solução | Solução | Solução

Combinar

Solução

Técnica Divisão e Conquista



Técnica Divisão e Conquista
Algoritmo Genérico

DivisãoeConquista(x)

if x é pequeno ou simples do
        return resolver(x)
else
        decompor x em conjuntos menores x_0, x_1, … x_n
        for i $\leftarrow$ 0 to n do
                $y_i \leftarrow$ DivisãoeConquista(x_i)
                $i \leftarrow i + 1$
        combinar y_i’s

return y

Técnica Divisão e Conquista

Exemplo 1- Máximo elemento de uma lista

O problema consiste em encontrar o maior elemento de um array a [1 .. n]

3	8	1	7	6	2	9	4

Técnica Divisão e Conquista

Exemplo 1 - Máximo elemento de uma lista - Pseudocódigo

```
maximo (lista, int inicio, int fim)

        if ( inicio = fim )
           retorna lista(inicio)

        meio = ( inicio + fim ) / 2;

        m1 = maximo(lista, inicio, meio)
        m2 = maximo(lista, meio+1, fim);

        if (m1 >= m2) retorna m1;
           else retorna m2;
```

Técnica Divisão e Conquista

Exemplo 1 - Máximo elemento de uma lista

```
package paa;

public class DivConquista_1 {

 public static void main(String[] args) {
        int[] lista = { 4,133,1,9,5,99,2 } ;
        int indiceInicio = 0;
        int indiceFim = lista.length-1;

        System.out.println("Maximo valor da lista: "
            + maximo(lista, indiceInicio, indiceFim));
 }
```

Técnica Divisão e Conquista

Exemplo 1 – Máximo elemento de uma lista

```
public static Integer maximo (int[] lista, int indiceInicio, int indiceFim) {

  if (  indiceInicio == indiceFim )
         return lista[indiceInicio] ;

  int indiceMetade = ( indiceInicio + indiceFim ) / 2;

  Integer m1 = maximo(lista, indiceInicio, indiceMetade);
  Integer m2 = maximo(lista, indiceMetade+1, indiceFim);

  if (m1 >= m2) return m1;
         else return m2;
 }
}
```

Técnica Divisão e Conquista

Exemplo 2 – Soma dos valores de uma lista

```
package paa;

public class DivConquista_2 {

  public static void main(String[] args) {

     int[] lista = { 4,6,1,9,5,100 } ;

     int indiceInicio = 0;
     int indiceFim = lista.length-1;

     System.out.println("Soma dos valores da lista: " +
        soma(lista, indiceInicio, indiceFim));
  }
```

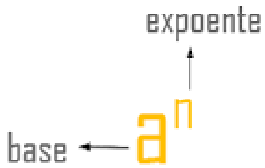
Técnica Divisão e Conquista

Exemplo 2 – Soma dos valores de uma lista

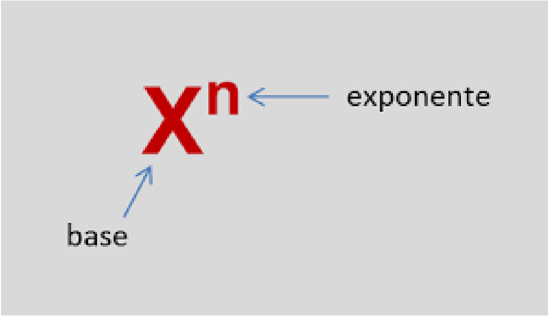
```
public static Integer soma (int[] lista, int indiceInicio, int indiceFim) {

    if (indiceInicio == indiceFim )
     return lista[indiceInicio];

    int indiceMetade = (indiceInicio + indiceFim)/2;

    int s1 = soma(lista, indiceInicio, indiceMetade);

    int s2 = soma(lista, indiceMetade+1, indiceFim);

    return (s1 + s2);
   }
}
```

Técnica Divisão e Conquista

Exemplo 3 - Exponenciação



O problema consiste em computar x^n.



Solução Divisão e Conquista

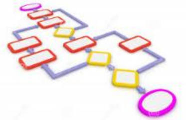
Exemplo 3 - Exponenciação

```
Pow(a, n)

    if n=0 then
        return 1
    if n é par then
        return Pow(a, n/2) × Pow(a, n/2)
    else
        return Pow(a,(n-1)/2) × Pow(a, (n-1)/2) × a
```

Exemplo 3

Solução Divisão e Conquista - Exponenciação

```java
package paa;

public class DivConquista_3 {

  public static void main(String[] args) {

  int  x = 2;
  int  n = 5;

  System.out.println("Potencia de " + x + " elevado a " +
                n + " :  " +  pot(x,n));

  }
```



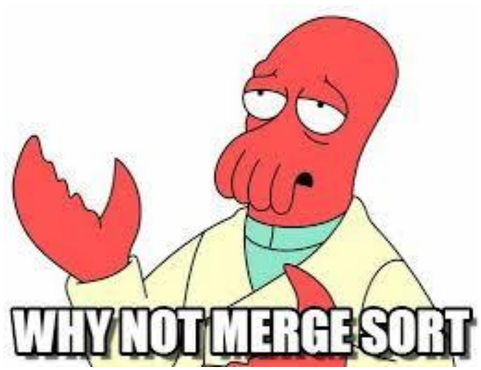
Exemplo 3

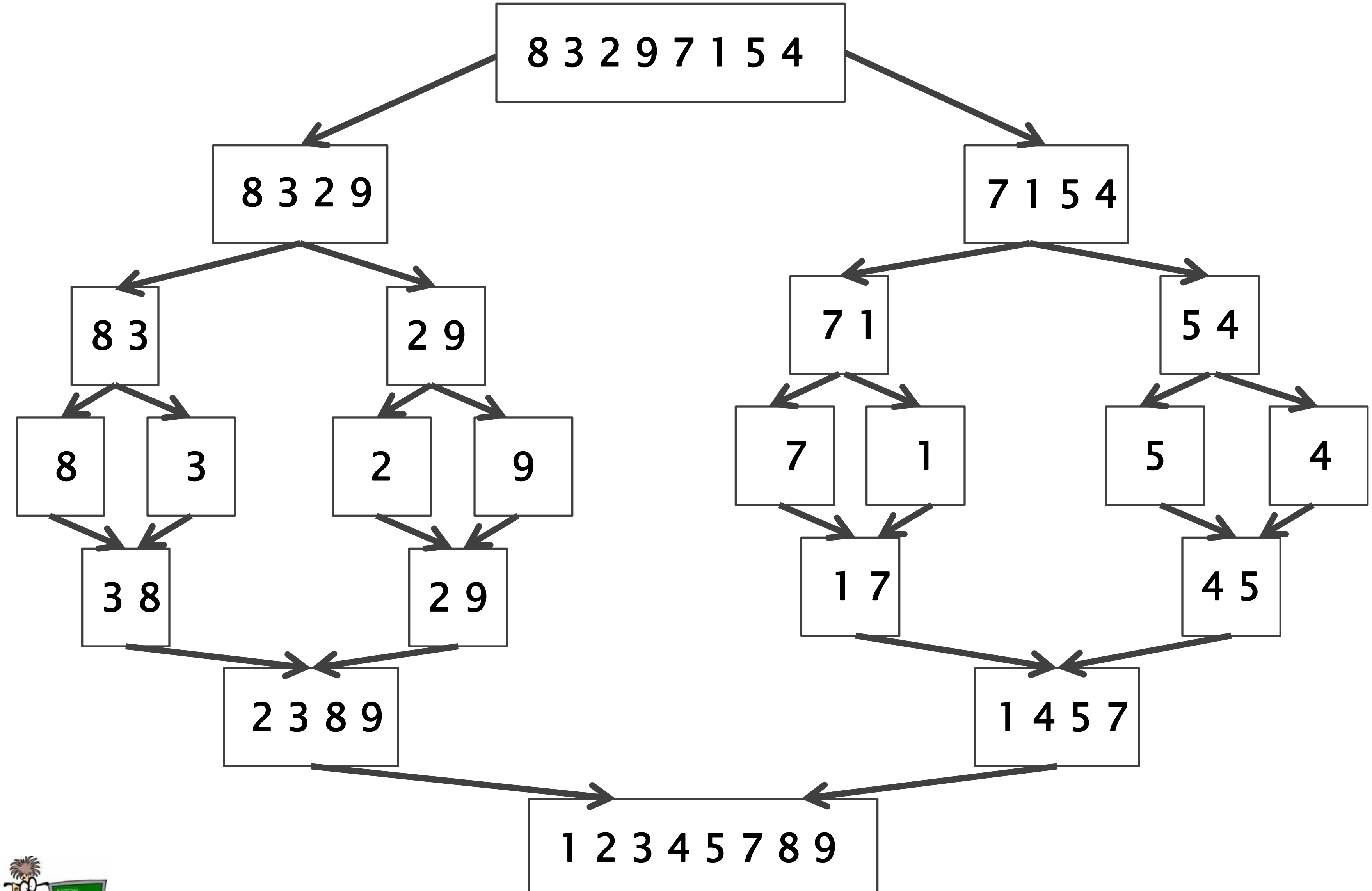
Solução Divisão e Conquista - Exponenciação

```
public static Long pot (int x , int n) {

 if (n == 0) return 1L;

 if (n%2 == 0)

  return (pot (x, n/2 ) * pot(x,n/2) );

 else

  return (pot (x, (n-1)/2 )  * pot(x, (n-1)/2) * x ) ;
  }
}
```

Mergesort

- É um perfeito exemplo de uma aplicação da técnica **Divide-and-Conquer**.
- O algoritmo classifica um dado array A[0..n-1] dividindo-o em duas metades $A[0 \,..\, \lfloor n/2 \rfloor - 1]$ e $A[\lfloor n/2 \rfloor \,..\, n-1]$, sorteando cada uma delas de forma recursiva, e em seguida efetua um merge das partes sorteadas.




8 3 2 9 7 1 5 4
8 3 2 9
7 1 5 4
8 3
2 9
7 1
5 4
8
3
2
9
7
1
5
4
3 8
2 9
1 7
4 5
2 3 8 9
1 4 5 7
1 2 3 4 5 7 8 9

Mergesort - Implementação

```
package uscs;

public class MergeSort {

  public static int[] vetor = {8,3,2,9,7,1,5,4};

  static int n = vetor.length;

  public static void main(String[] args) {

          Sort(0,n-1);
          for (int i=0;i<n;i++ )
                  System.out.println("vetor[" + i + "] = " + vetor[i]);
}
```

Mergesort - Implementação

```
public static void Sort(int inicio, int fim) {

        if (inicio < fim) {

                int meio = (inicio + fim) / 2;

                Sort(inicio, meio);
                Sort(meio + 1, fim);
                Merge(inicio, meio, fim);
        }
}
```

Mergesort - Implementação

```
public static void Merge(int inicio, int meio, int fim) {

        int n = fim - inicio + 1;

        int[] temp = new int[n];
        int tamanho = temp.length;

        for (int posicao = 0; posicao < tamanho; posicao++) {
                temp[posicao] = vetor[inicio + posicao];
        }
```

Mergesort - Implementação

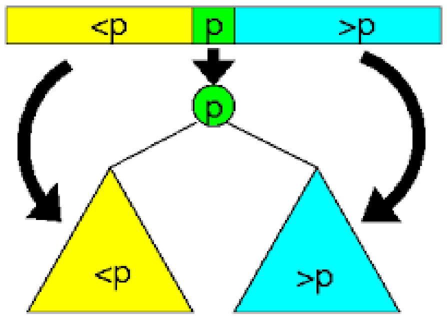
```
int i = 0;
int j = meio - inicio + 1;

 for (int posicao = 0; posicao < tamanho; posicao++) {

   if (j <= n- 1 )
    if ( i <= meio - inicio)
               if (temp[i] < temp[j] )
                          vetor[inicio + posicao] = temp[i++];
               else vetor[inicio + posicao] = temp[j++];
    else vetor[inicio + posicao] = temp[j++];
   else vetor[inicio + posicao] = temp[i++];
  }

 }

}
```

Mergesort - Eficiência

- Prova-se que a ordem de complexidade do algoritmo MergeSort é:

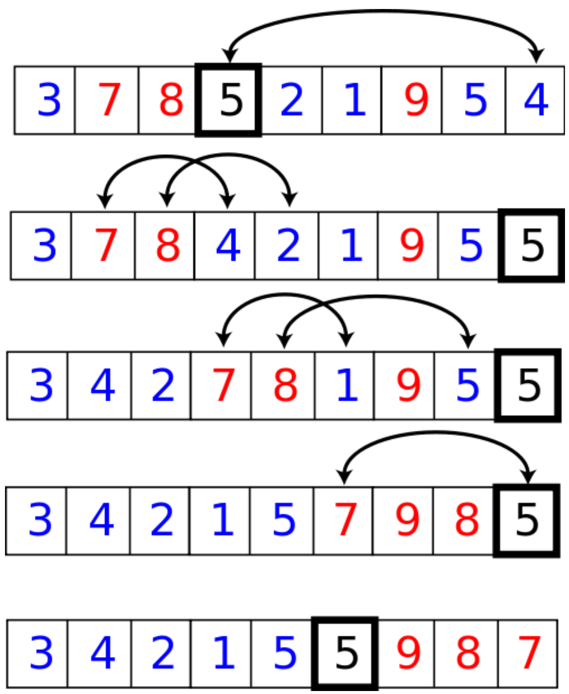
$$O(n) \ = \ n * \log(n)$$

Quicksort



Quicksort

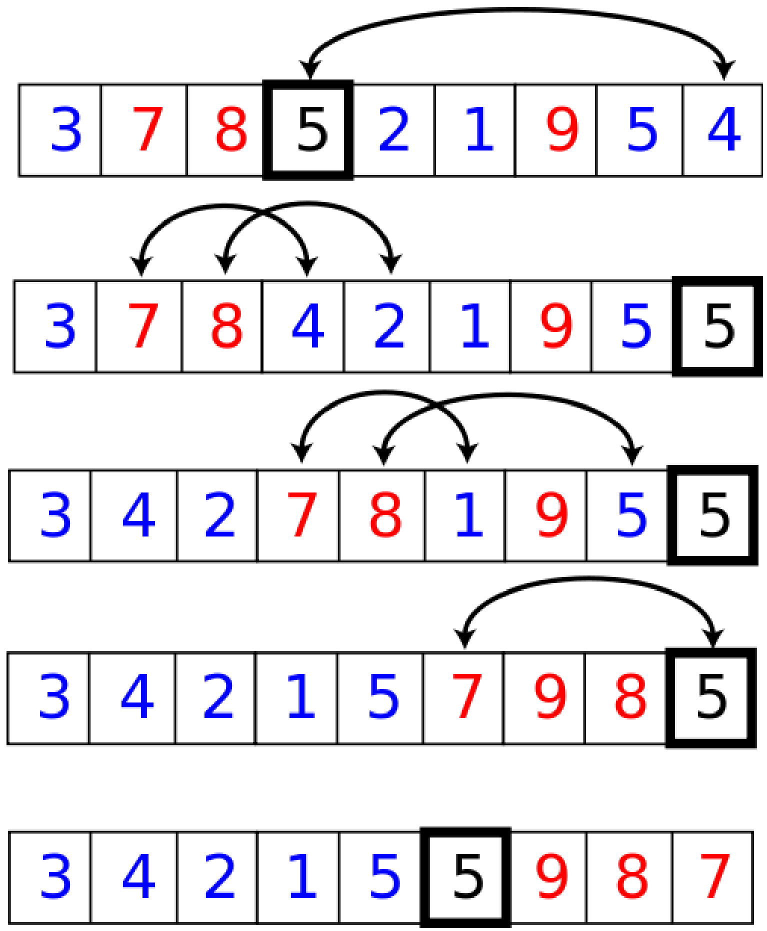
- É um algoritmo com boa eficiência;
- Emprega a estratégia **Divisão e Conquista**;
- Inicialmente divide uma lista em duas menores sub-listas, e recursivamente ordena estas sub-listas;



Quicksort - Algoritmo

- Escolha um elemento da lista, chamado **Pivot**;
- Geralmente o **Pivot** pode estar no meio da lista;
- Reordene a lista de modo que todos os elementos menores que o **Pivot** estejam antes dele e todos os elementos maiores que o **Pivot** estejam após ele;
- Após este particionamento, o **Pivot** está na sua posição final;
- Este procedimento inicial é chamado Operação de Partição;
- Recursivamente, aplique os passos acima para as sub-lista com elementos menores e separadamente para a sub-lista com maiores elementos.

Quicksort - Algoritmo



Quicksort - Algoritmo

```
package uscs;

import java.util.Arrays;

public class QuickSort {

        public static int[] lista = {24,2,45,20,56,75,2,56,99,53,12};

        public static void main(String[] args) {

                int n = lista.length;

                Quicksort(0 , n-1);

                System.out.println(Arrays.toString(lista));

}
```

Quicksort - Algoritmo

```
public static void Quicksort(int inicio, int fim)  {

        int i = inicio;
        int j = fim;

        int pivot = lista[inicio+(fim-inicio)/2];

        while (i <= j) {
          while (lista[i] < pivot)
                i++;

           while (lista[j] > pivot)
                j--;

            if (i <= j) {
                troca(i, j);
                i++;
                j--;
            }
         }
```

Quicksort - Algoritmo

```
         if (inicio < j)
            Quicksort(inicio, j);
        if (i < fim)
            Quicksort(i, fim);
    }

    public static void troca(int i, int j) {
        int temp = lista[i];
        lista[i] = lista[j];
        lista[j] = temp;
    }
}
```

Quicksort- Eficiência

- Prova-se que a ordem de complexidade do algoritmo **QuickSort** é:

  - **O(n) = n * log (n)** (melhor caso)

  - **O(n) = n^2** (no pior caso)