INSTITUTO FEDERAL
Ceará
Campus Tianguá

MÉTODOS DE ORDENAÇÃO POR DIVISÃO E CONQUISTA

Ciência da Computação

Disciplina: Construção e Análise de Algoritmos

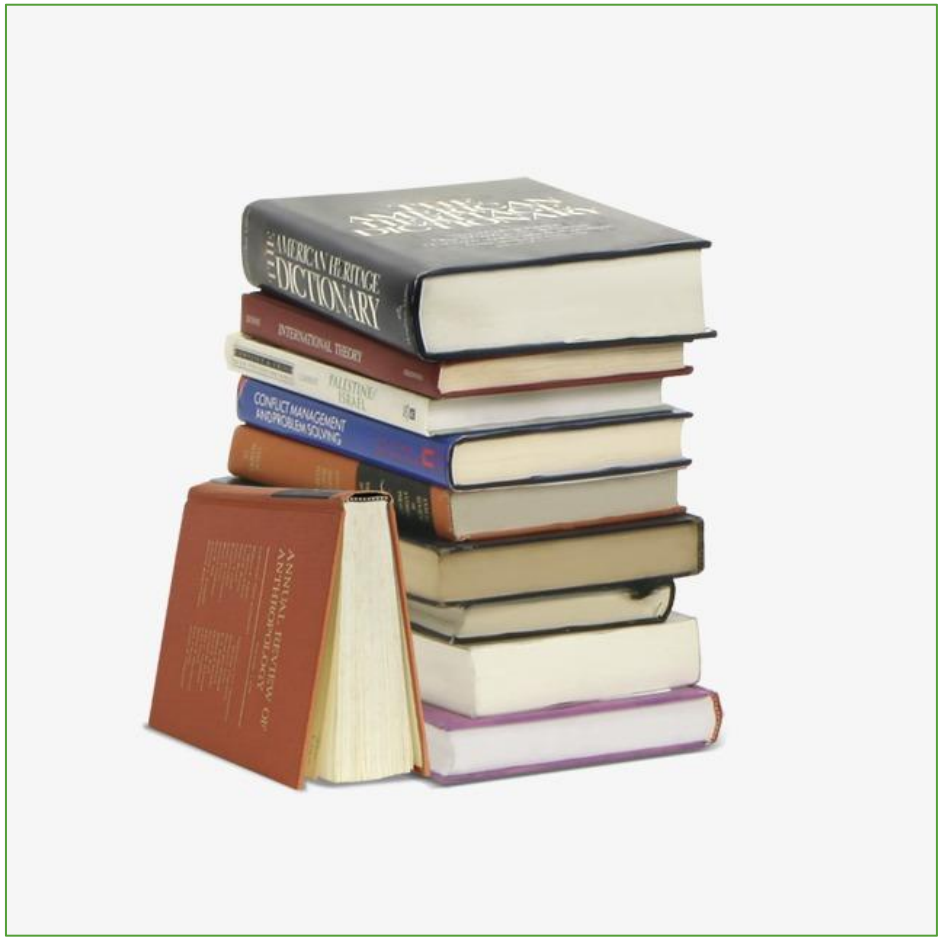
Professor: Adonias Caetano

Objetivos

- Compreender o método de ordenação do MergeSort e Quicksort.
- Ser capaz de implementar esses algoritmos em C.

Conteúdo da aula

- **Divisão & Conquista**
- **MergeSort**
  - Definição
  - Ideia Básica
  - Exemplos
  - Exercício
- **QuickSort**
  - Definição
  - Ideia Básica
  - Exemplos
  - Exercício



DIVIDIR & CONQUISTAR

Introdução

- É provavelmente uma das mais conhecidas estratégias de projeto de algoritmos;

- Os algoritmos **Dividir & Conquistar** funcionam de acordo com um plano geral

Introdução

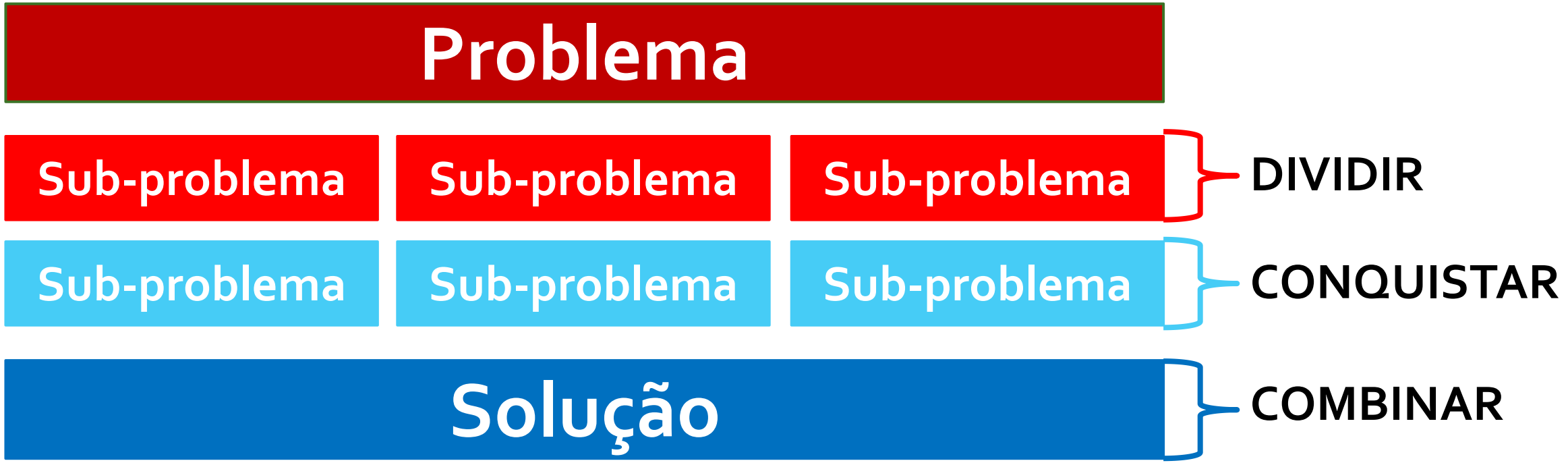
1. Dividir a instância do problema em duas ou mais instâncias menores

2. Resolver as instâncias menores recursivamente

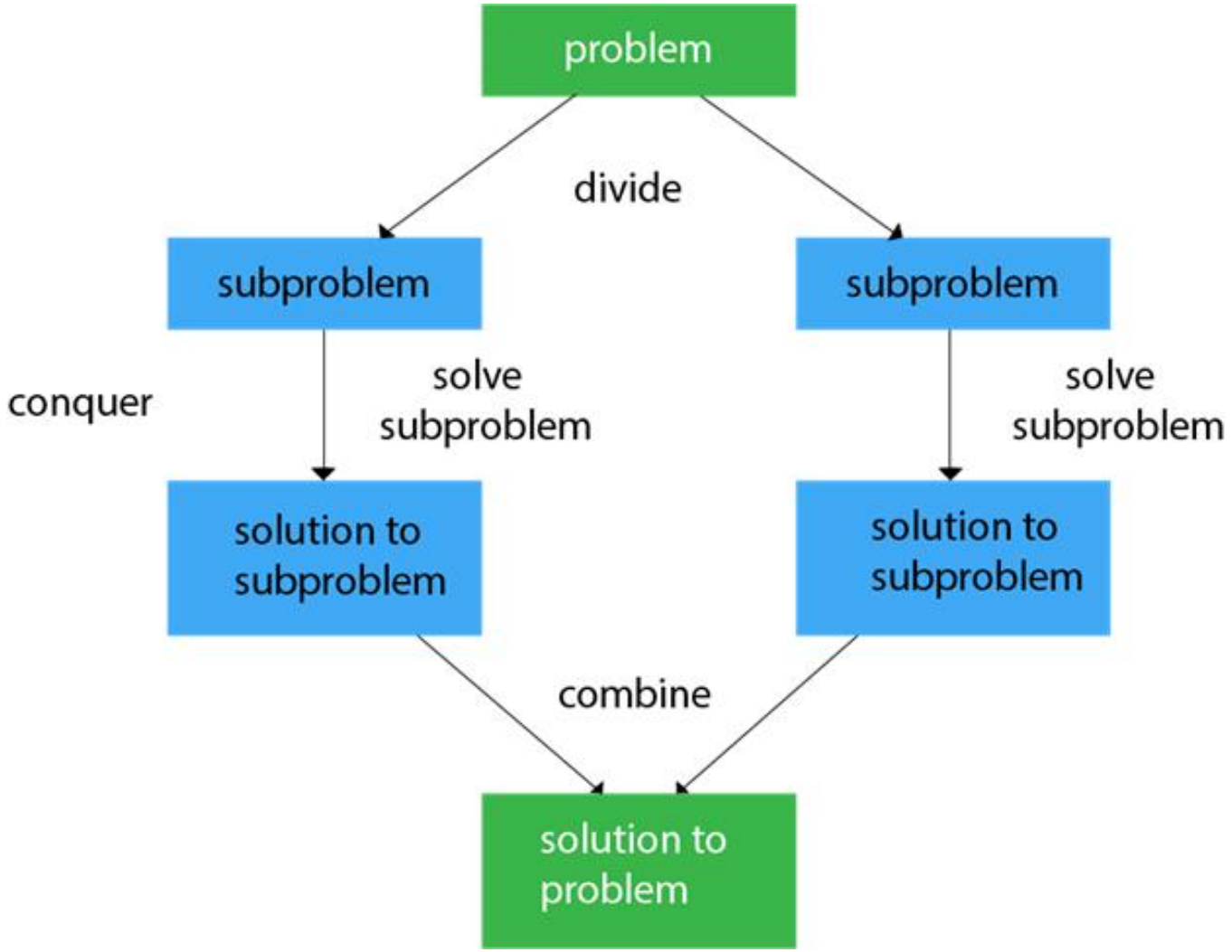
3. Obter a solução para as instâncias originais (maiores) através da combinação destas soluções.

Divisão & Conquista

- O paradigma de dividir e conquistar envolve 3 passos:



Divisão & Conquista

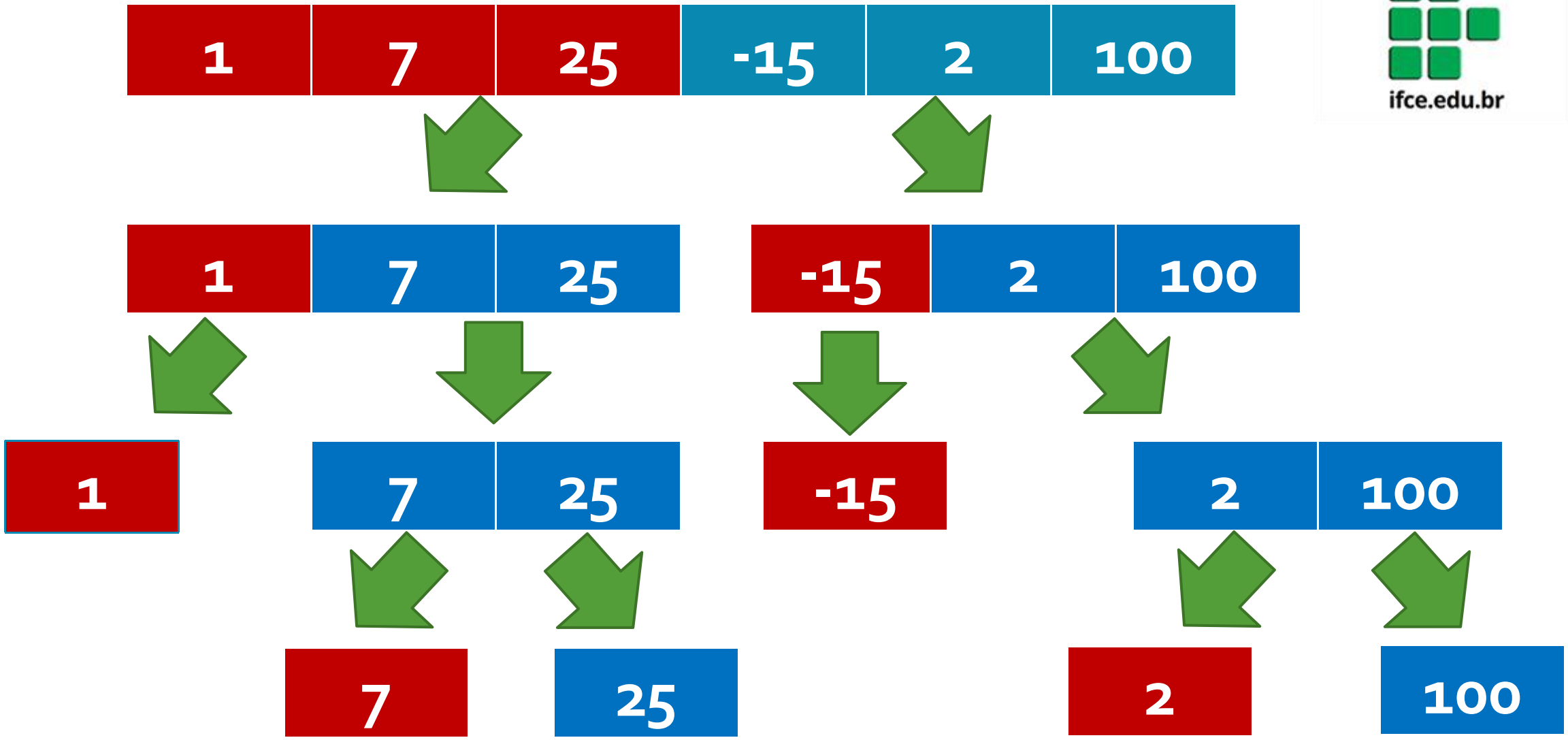


Exemplo

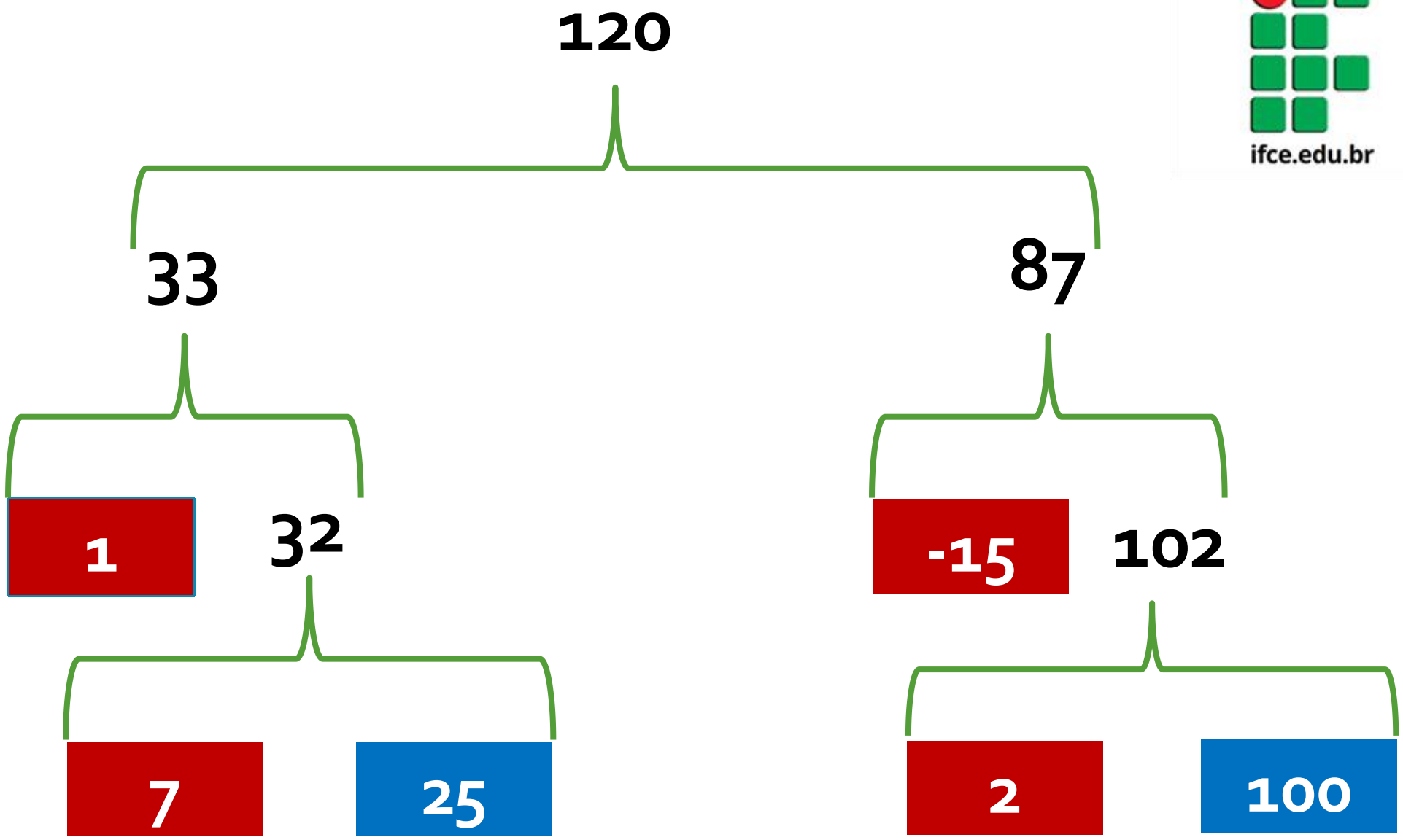
- Computar a soma de $\boldsymbol{n}$ números $a_0, \cdots, a_{n-1}$

- Se $n > 1$, podemos dividir o problema em duas instâncias do mesmo problema:
  - Soma dos primeiros $\lfloor n/2 \rfloor$números
  - Soma dos $\lceil n/2 \rceil$ números restantes.

Exemplo



Exemplo



Exemplo

- Uma vez estas duas somas computadas, adicionamos seus valores para obter o resultado final:

$$a_0, \cdots, a_{n-1} = \left(a_0 + \cdots + a_{\lfloor n/2 \rfloor - 1}\right) + \left(a_{\lceil n/2 \rceil} + \cdots + a_{n-1}\right)$$

Exemplo

Esta é uma maneira eficiente de computar a soma de n números?

É mais eficiente do que uma adição força bruta?

ifce.edu.br

Exemplo

Assim, nem todo algoritmo dividir & conquistar é mais eficiente do que uma solução força bruta.

Por que divisão e conquista?

ifce.edu.br

- Porém, frequentemente, o tempo gasto na execução das três etapas do algoritmo dividir & conquistar é menor do que a resolução por outros métodos.

- A estratégia dividir & conquistar produz os algoritmos mais importantes e eficientes em ciência da computação;

- **Importante**: A estratégia dividir & conquistar é idealmente adaptada a computação paralela.

Análise

- Uma instância de tamanho $\boldsymbol{n}$ pode ser dividida em diversas instâncias de tamanho $\boldsymbol{n/b}$, com $\boldsymbol{a}$ deles devendo ser resolvidos.

- Assim, obtemos a seguinte recorrência:

$$\boldsymbol{T(n) = aT(n/b) + f(n)}$$

- A análise de eficiência é feita através dos métodos vistos anteriormente

ifce.edu.br

MERGE SORT

Definição

- Mergesort é um algoritmo de ordenação e um exemplo clássico do uso da técnica de divisão-e-conquista (to merge = intercalar).

Vantagens

- Algoritmo de ordenação de simples implementação e fácil entendimento utilizando chamadas recursivas.

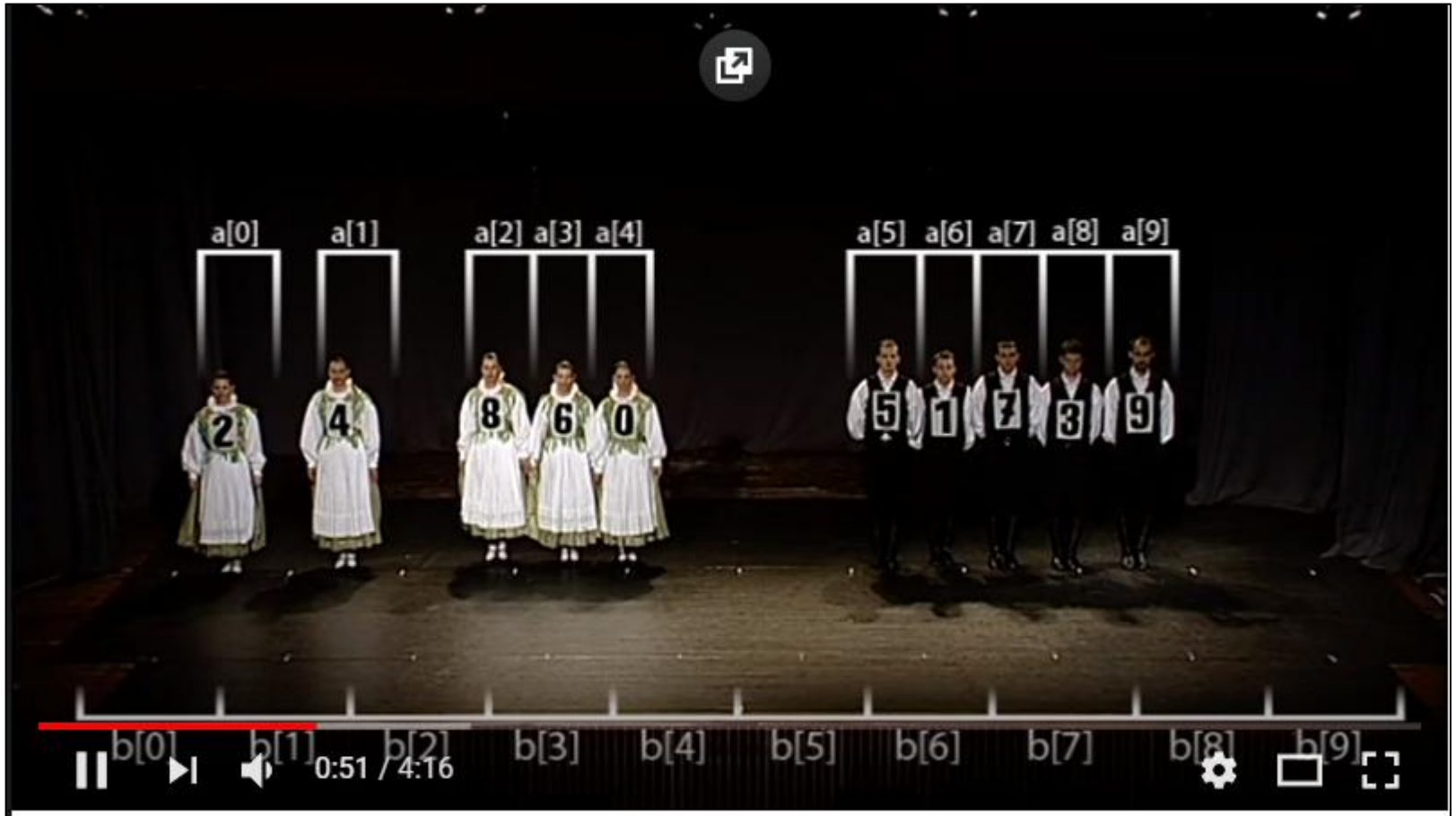
Desvantagens

- **Alto consumo de memoria** devido a serie de chamadas recursivas e uso de array auxiliar.

Ideia Básica

- **Divisão**: divida o vetor com n elementos em dois subvetores de tamanho $\lfloor n/2 \rfloor$ e $\lceil n/2 \rceil$
- **Conquista**: ordene os dois subvetores recursivamente usando o Mergesort
- **Combinação**: intercale os dois subvetores para obter um vetor ordenado usando o algoritmo Intercala

AlgoRythmics



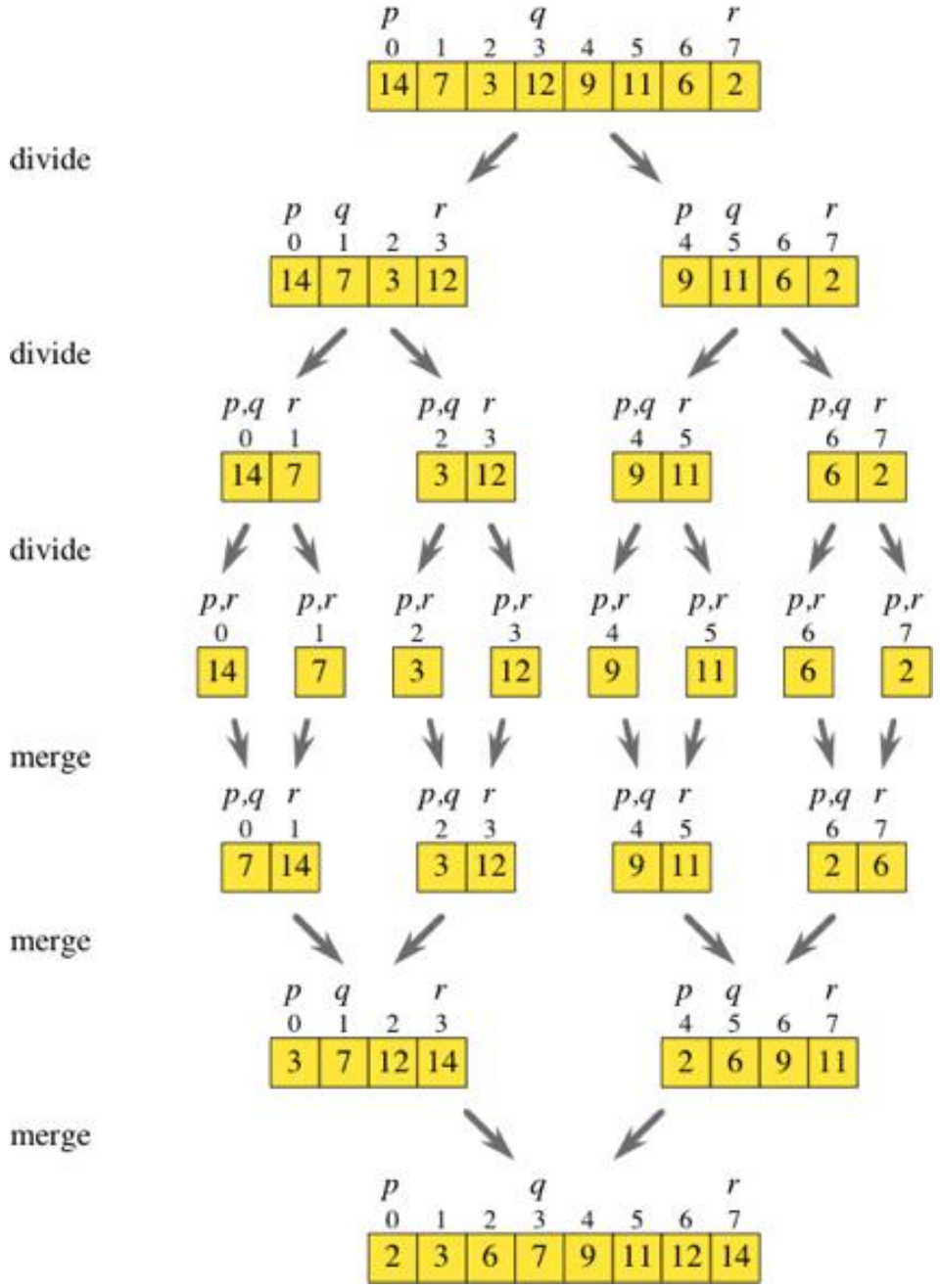
Merge-sort with Transylvanian-saxon (German) folk dance

https://youtu.be/XaqR3G_NVoo

Animação

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

Ilustração


p q r
0 1 2 3 4 5 6 7
14 7 3 12 9 11 6 2
divide
p q r
0 1 2 3
14 7 3 12
p q r
4 5 6 7
9 11 6 2
divide
p,q r
0 1
14 7
p,q r
2 3
3 12
p,q r
4 5
9 11
p,q r
6 7
6 2
divide
p,r
0
14
p,r
1
7
p,r
2
3
p,r
3
12
p,r
4
9
p,r
5
11
p,r
6
6
p,r
7
2
merge
p,q r
0 1
7 14
p,q r
2 3
3 12
p,q r
4 5
9 11
p,q r
6 7
2 6
merge
p q r
0 1 2 3
3 7 12 14
p q r
4 5 6 7
2 6 9 11
merge
p q r
0 1 2 3 4 5 6 7
2 3 6 7 9 11 12 14

Algoritmo

- O algoritmo de ordenação MERGESORT utiliza duas funções ou rotinas:
- **Merge** (chamado de Intercala às vezes)
- **MergeSort**

Algoritmo MergeSort

- Caso o tamanho do vetor seja maior que 1
  1. divida o vetor no meio
  2. ordene a primeira metade recursivamente
  3. ordene a segunda metade recursivamente
  4. intercale as duas metades
- Senão devolva o elemento

Pseudocódigo MergeSort

ifce.edu.br

MergeSort (A, p, r)

1: **if** $p < r$ **then**
2: $q = (p + r)/2$
3: Mergesort(A, p, q)
4: Mergesort$(A, q + 1, r)$
5: Merge(A, p, q, r)

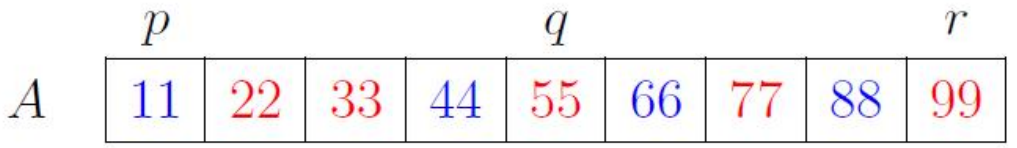
Chamada Inicial: Mergesort$(A, 1, n)$

Algoritmo Merge

- **Problema:** Dados $A[p \cdots q]$ e $A[q+1 \cdots r]$crescentes, rearranjar $A[p \cdots r]$ de modo que ele fique em ordem crescente.

Entrada:

Saída:



Algoritmo Merge

- A intercalação de dois vetores ordenados pode ser feito em tempo linear
  - Uma variável em cada vetor indica o próximo elemento a ser inserido a lista intercalada
  - Enquanto ambos vetores tiverem elementos coloque o menor entre os dois elemento indicados no vetor intercalado e incremente índice respectivo
  - Quando um dois vetores não tiver mais elementos, concatene o outro no final do vetor intercalado.

Algoritmo Merge

```
Merge (A, p, q, r)
  ▷ B[p...r] vetor auxiliar
 1: for i = p to q do
 2:     B[i] = A[i]
 3: for j = q + 1 to r do
 4:     B[r + q + 1 − j] = A[j]
 5: i = p
 6: j = r
 7: for k = p to r do
 8:     if B[i] ≤ B[j] then
 9:         A[k] = B[i]
10:         i = i + 1
11:     else
12:         A[k] = B[j]
13:         j = j − 1
```

ifce.edu.br

Exercício de Fixação

- Implemente na linguagem C o algoritmo de ordenação Merge sort.

Solução

```c
# include <stdio.h>
# include <stdlib.h>
# define MAX 6

int A[] = {5, 6, -9, 9, 0, 4 }, B[MAX];

void Merge(int *A, int e, int q, int d);
void MergeSort(int *A, int e, int d);


int main()
{
    int i;

    MergeSort(A, 0, MAX - 1);

    printf("\n\nVetor ordenado:\n");
    for(i = 0; i < MAX; i++)
        printf("%d\t", A[i]);
    printf("\n");
    return 0;
}
```

ifce.edu.br

Solução

ifce.edu.br

```
57  void MergeSort(int *A, int e, int d)
58  {
59      int q;
60      if (e < d)
61      {
62          q = floor((e + d)/2); // Determina a metade do vetor
63          MergeSort(A, e, q); // Primeira metade
64          MergeSort(A,q + 1, d); // Segunda metade
65          Merge(A, e, q, d); // Combina as metades já ordenadas
66      }
67  }
```

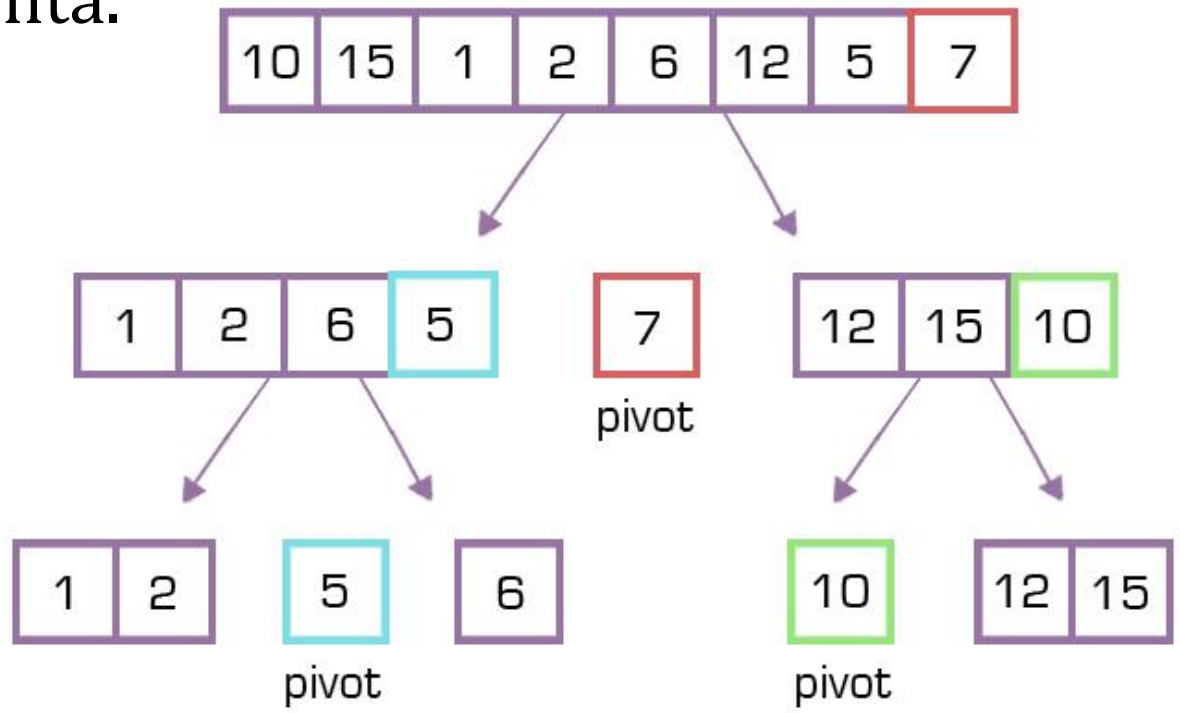
Solução

```
24  void Merge(int *A, int e, int q, int d)
25  {
26      int *temp, p1, p2, tamanho;
27      int i, j, k, fim1 = 0, fim2 = 0;
28      tamanho = d- e + 1;
29      p1 = e; // Ponteiro do vetor 1
30      p2 = q + 1; // Ponteiro do vetor 2
31
32      if ((temp = (int *) malloc(tamanho*sizeof(int))) != NULL)
33      {
34          for(i = 0; i<tamanho; i++)
35          {
36              if(!fim1 && !fim2) // Nenhum dos vetores chegou ao fim
37              {
38                  if(A[p1] < A[p2]) temp[i] = A[p1++]; //Seleciona o menor
39                  else temp[i]=A[p2++];
40
41                  if(p1 > q) fim1 = 1;
42
43                  if(p2 > d) fim2 = 1;
44              }
45              else
46              {
47                  if(!fim1) temp[i] = A[p1++];
48                  else temp[i] = A[p2++];
49              }
50          }
51          for(j = 0, k = e; j < tamanho; j++, k++) //Copia do vetor temporário p/ o vetor a ser retornado
52          A[k] = temp[j];
53      }
54      free(temp);
55  }
```

QUICK SORT

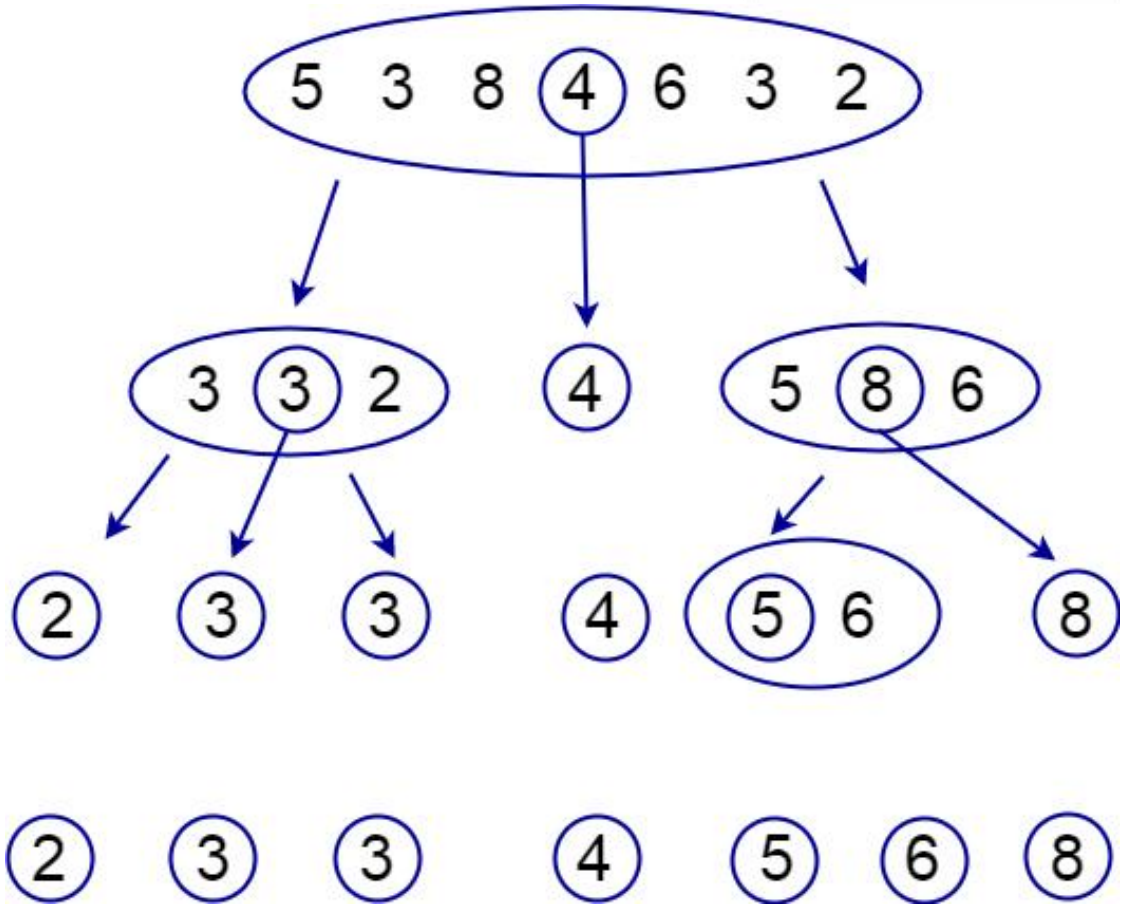
Definição

- O **Quick Sort** é um dos método mais rápidos de ordenação, apesar de apresentar às vezes partições desequilibradas que o podem conduzir a uma ordenação lenta.



Ideia Básica

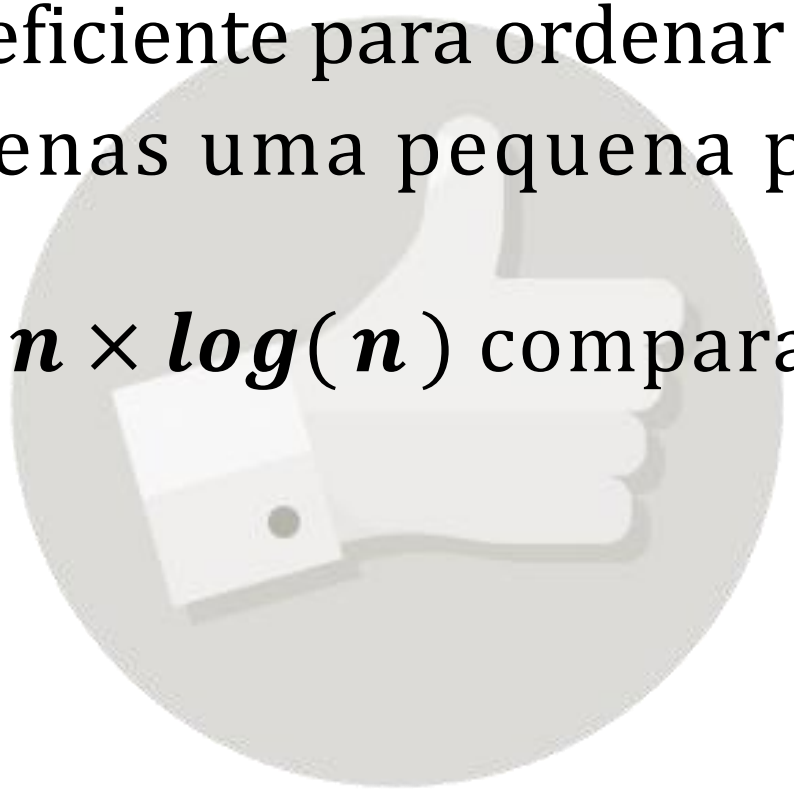
1. Divide o vetor em dois subvetores de acordo com pivô. Por exemplo, o 1º elemento;
2. O pivô é colocado entre ambos, ficando na posição correta.
3. Os dois subvetores são ordenadas de forma idêntica, até que se chegue ao vetor com um só elemento.



Vantagens vs. Desvantagens

ifce.edu.br

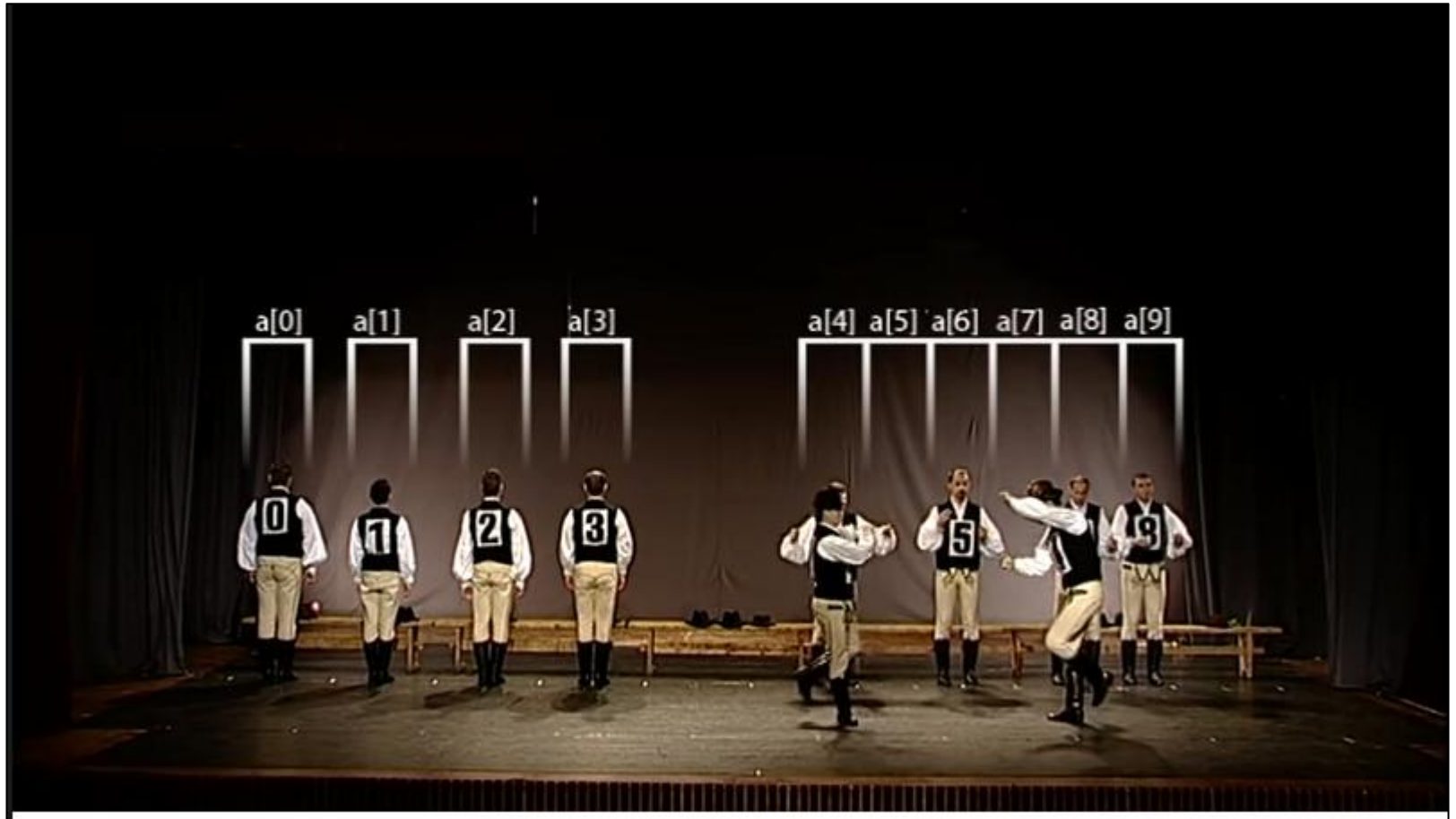
- **Vantagens:**
  - É extremamente eficiente para ordenar arquivos de dados.
  - Necessita de apenas uma pequena pilha como memória auxiliar.
  - Requer cerca de $\boldsymbol{n} \times \boldsymbol{log}(\boldsymbol{n})$ comparações em média para ordenar $\boldsymbol{n}$ itens.



Vantagens vs. Desvantagens

- **Desvantagens:**
  - Tem um pior caso $\boldsymbol{O(n^2)}$ 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**.

AlgoRythmics



Quick-sort with Hungarian (Küküllőmenti legényes) folk dance

https://youtu.be/ywWBy6J5gz8

Animação

6 5 3 1 8 7 2 4

Baseado na escolha aleatória do pivô

Algoritmo

1.Escolha um elemento da lista, denominado *pivô*;

Rearranje a lista de forma que todos os elementos anteriores ao pivô sejam menores que ele, e todos os elementos posteriores ao pivô sejam maiores que ele.

Ao fim do processo o pivô estará em sua posição final e haverá duas sublistas não ordenadas. Essa operação é denominada *partição*;

Recursivamente ordene a sublista dos elementos menores e a sublista dos elementos maiores;

Pseudocódigo

```
QuickSort(A, p, r)
        if p < r
                q = Partition(A, p, r)
                QuickSort(A, p, q)
                QuickSort(A, q+1, r)
```

```
Partition(A, p, r)
        x = A[p]
        i = p - 1
        j = r + 1
        while (TRUE)
                repeat  j = j - 1
                until A[j] ≤ x
                repeat i = i + 1
                until A[i] ≥ x
                if i < j
                        swap(A[i], A[j])
                else
                        return j
```

Exercício de Fixação

- Implemente na linguagem C o algoritmo de ordenação Quicksort.

Solução

```c
#include <stdio.h>

void swap(int* a, int* b);
int partition(int vec[], int left, int right) ;
void quickSort(int vec[], int left, int right);

int main()
{
    int v[] = {5, 6, -9, 9, 0, 4 };
    int n = 6, i;

    quickSort(v, 0, n - 1);

    printf("\n\nVetor ordenado:\n");
    for(i = 0; i < n; i++)
        printf("%d\t", v[i]);
    printf("\n");
    return 0;
}

void swap(int* a, int* b)
{
    int tmp;
    tmp = *a;
    *a = *b;
    *b = tmp;
}
```

ifce.edu.br

Solução

```
29  void quickSort(int vec[], int left, int right) {
30      int r;
31      if (right > left) {
32          r = partition(vec, left, right);
33          quickSort(vec, left, r - 1);
34          quickSort(vec, r + 1, right);
35      }
36  }
37
38  int partition(int vec[], int left, int right) {
39      int i, j;
40      i = left;
41
42      for (j = left + 1; j <= right; ++j) {
43          if (vec[j] < vec[left]) {
44              ++i;
45              swap(&vec[i], &vec[j]);
46          }
47      }
48      swap(&vec[left], &vec[i]);
49
50      return i;
51  }
```

ifce.edu.br

Quicksort não recursivo

```
void QuickSortNaoRec (Vetor A, Indice n)
{
     TipoPilha pilha; TipoItem item;
     int esq, dir, i, j;

     FPVazia(&pilha);
     esq = 0;
     dir = n-1;
     item.dir = dir;
     item.esq = esq;
     Empilha(item,&pilha);
     do
      if (dir > esq)  {
          Particao(A,esq,dir,&i, &j);
          item.dir = j;
          item.esq = esq;
          Empilha(item,&pilha);
          esq = i;
       }
       else {
            Desempilha(&pilha,&item);
            dir = item.dir;
            esq = item.esq;
        }
   while (!Vazia(pilha));
}
```

```
void QuickSortNaoRec (Vetor A, Indice n)
{
     TipoPilha pilha; TipoItem item;
     int esq, dir, i, j;

     FPVazia(&pilha);
     esq = 0;
     dir = n-1;
     item.dir = dir;
     item.esq = esq;
     Empilha(item, &pilha);
     do
       if (dir > esq)  {
          Particao(A,esq,dir,&i, &j);
          if ((j-esq)>(dir-i))  {
             item.dir = j;
             item.esq = esq;
             Empilha(item, &pilha);
             esq = i;
          }
          else {
              item.esq = i;
              item.dir = dir;
              Empilha(item, &pilha);
              dir = j;
          }
       }
       else {
          Desempilha(&pilha,&item);
          dir = item.dir;
          esq = item.esq;
       }
     while (!Vazia(pilha));
  }
```

ifce.edu.br

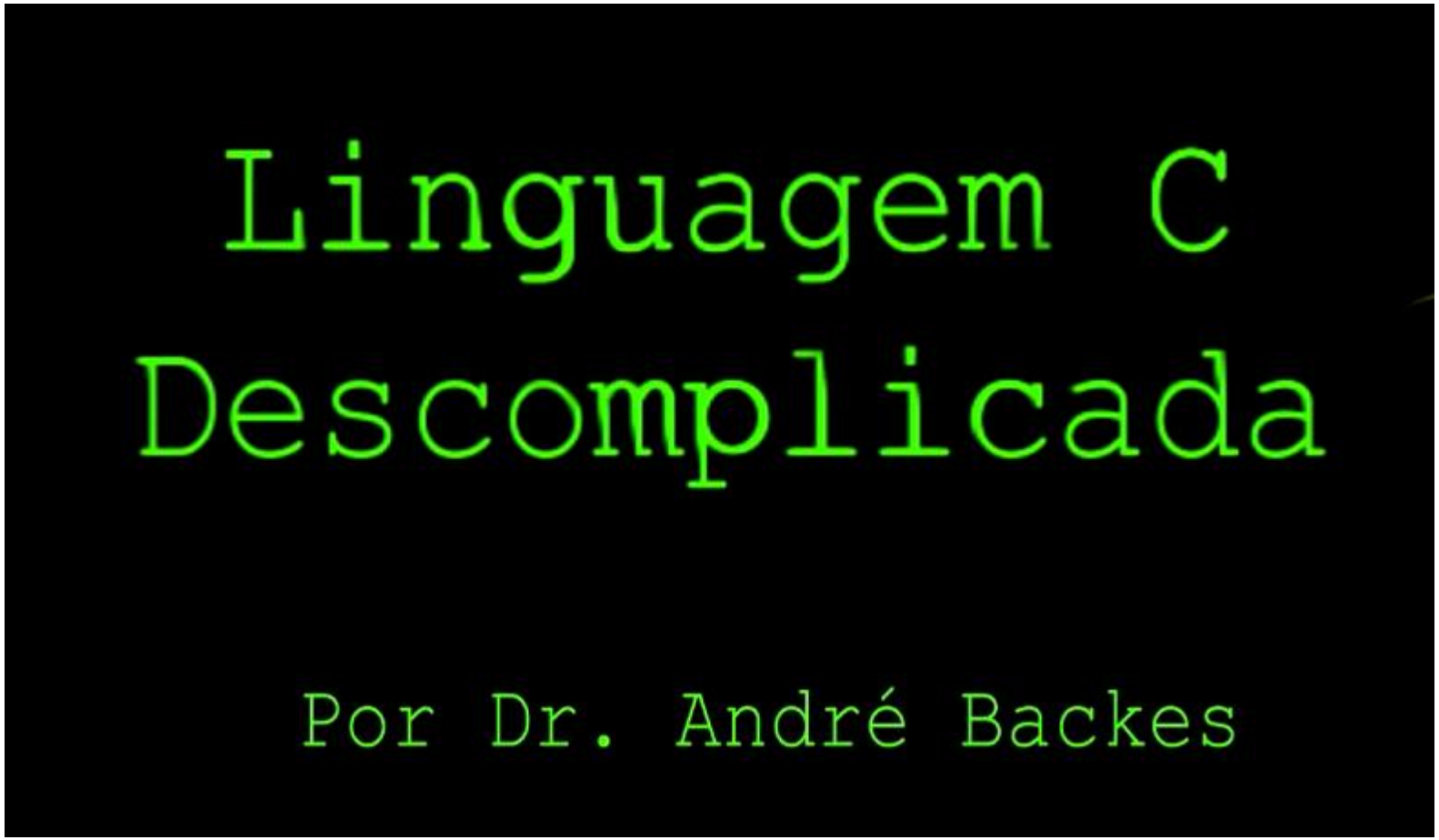
Melhorias no Quicksort

- Pivô – Mediana de três
- Não empilhar quando tem um só
- Melhor ainda: usar algoritmo de inserção para vetores pequenos (menos de 10 ou 20 elementos)
- Escolha do lado a ser empilhado
- Melhoria no tempo de execução de 25% a 30%

Dúvidas?

Vídeo-Aulas

- Aulas 51 e 52



Vídeo-Aulas

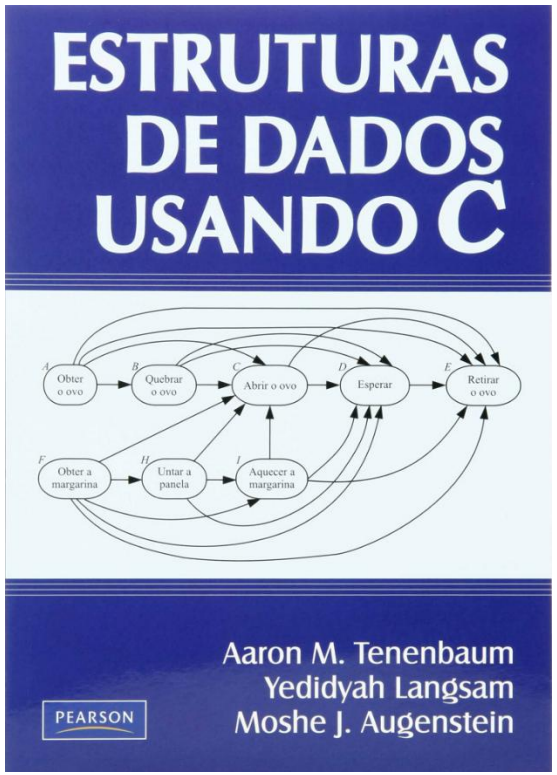
- Como fazer Merge Sort em Java - Canal do Código

Vídeo-Aulas

- Quicksort simples e sem complicações - Canal do Código

Bibliografia

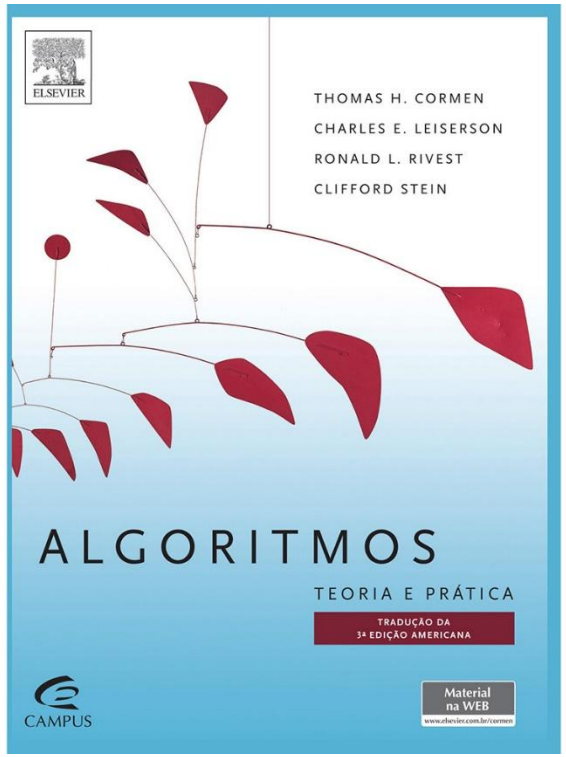
- SHILDT, Herbert. **C, Completo e Total.** 3ª edição. São Paulo: Makron Books, 1996.
- **Capítulo 19**

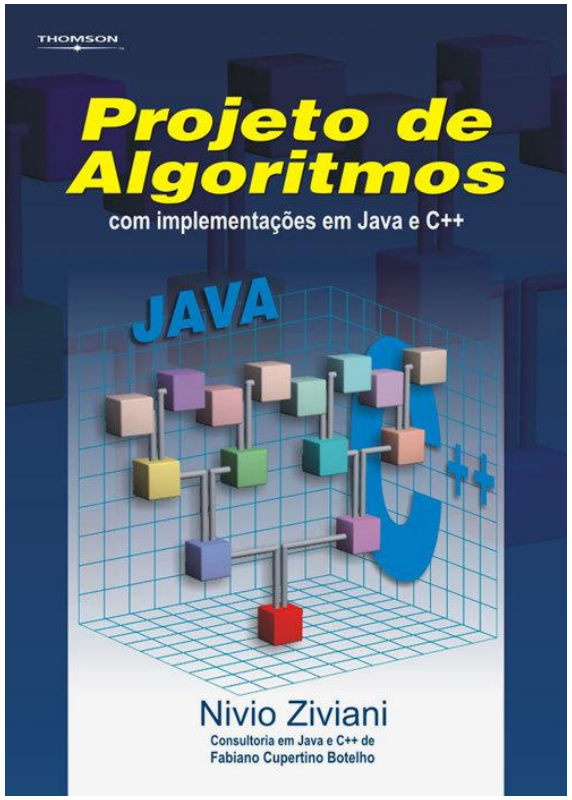


- TENENBAUM, Aaron M.; LANGSAM, Yedidyah; AUGENSTEIN, Moshe J. **Estruturas de dados usando C.** São Paulo : MAKRON *Books,* 1995.
- **Capítulo 06**

Bibliografia

- CORMEN, Thomas *et al*. Algoritmos: teoria e prática. Rio de Janeiro: Elsevier, 2002.
- **Capítulo 07**

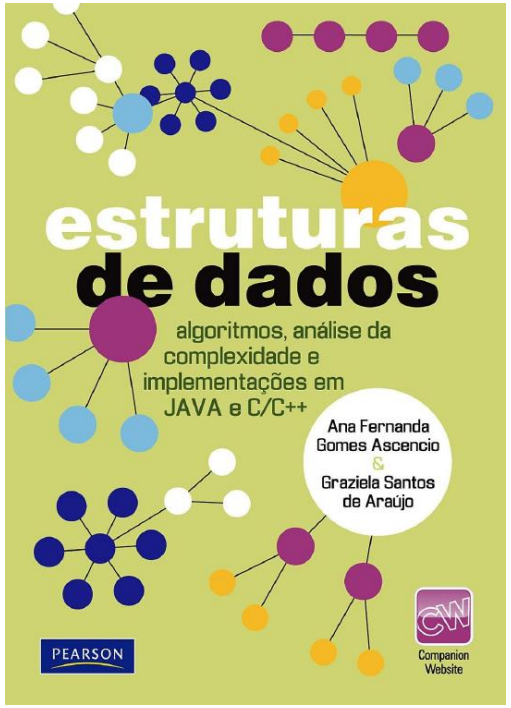


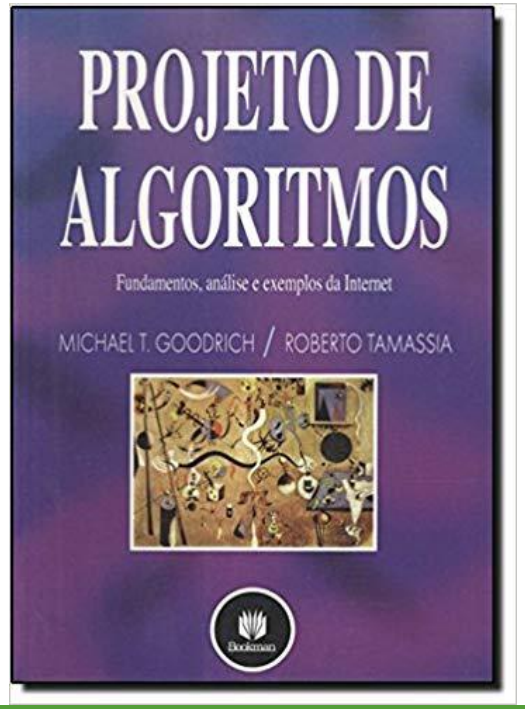


- ZIVIANI, Nivio. **Projeto de Algoritmos: com implementações em Java e C++.** 1ª edição. São Paulo: Cengage Learning, 2013.
- **Capítulo 03**

Bibliografia

- ASCENCIO, Ana Fernanda Gomes. Estruturas de dados: análise da complexidade e implementações em Java e C/C++ . São Paulo, Pearson Prentice Hall, 2010.
- **Capítulo 04**





- GOODRICH, Michael T.; TAMASSIA, Roberto. **Projeto de algoritmos: fundamentos, análise e exemplos da internet**. Bookman Editora, 2009.
- **Capítulo 04**