# Algoritmo e Estrutura de Dados II
# CTCO02

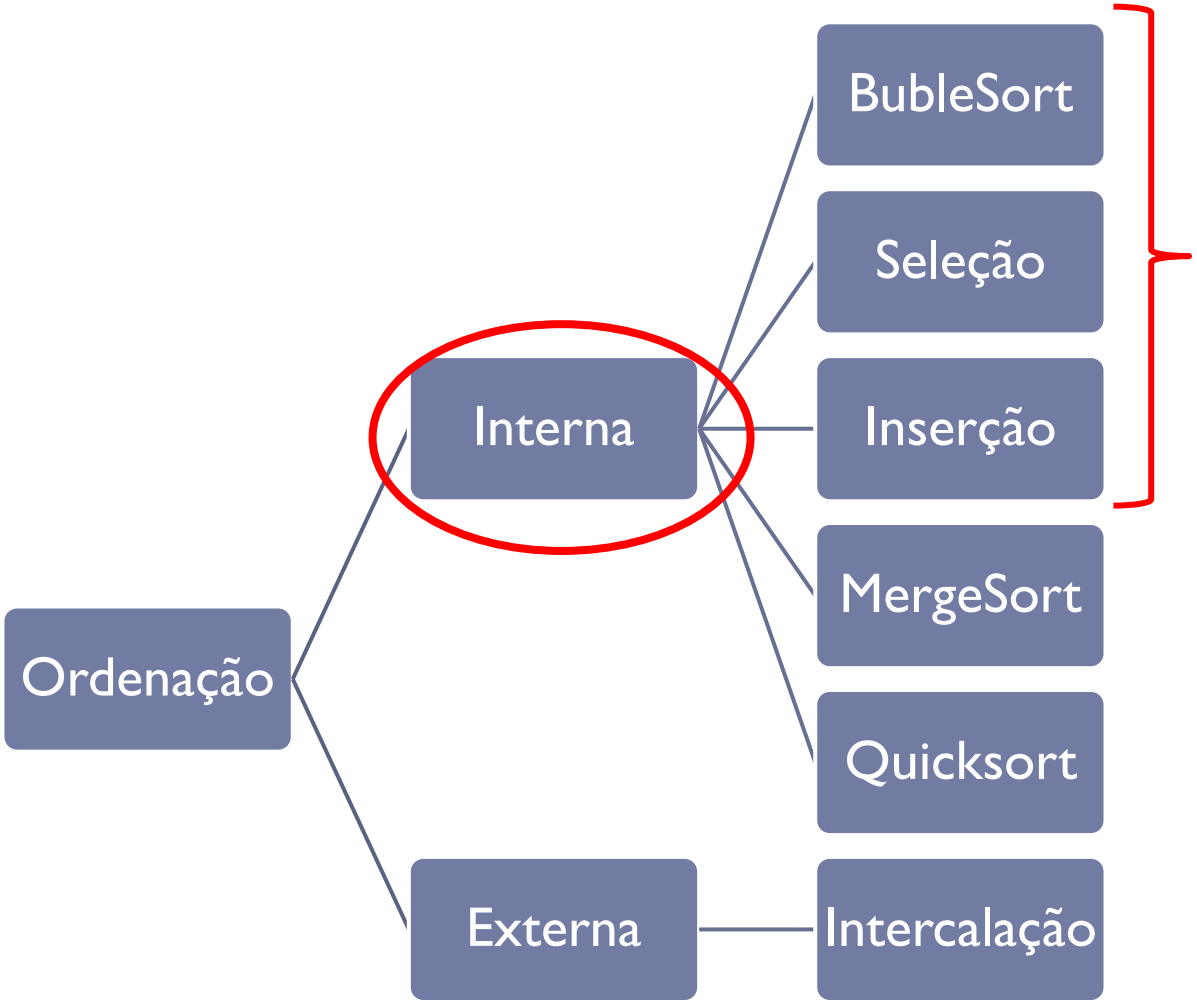
MergeSort

Vanessa Souza

Universidade Federal de Itajubá – Instituto de Matemática e Computação

# Ordenação

# Classificação dos Métodos de Ordenação



# Comparação entre os métodos

- Existe uma outra gama de algoritmos de ordenação mais eficientes ($O(n\log_2 n)$).
  - mergeSort
  - quickSort

- Esses algoritmos baseiam-se na estratégia de DIVIDIR PARA CONQUISTAR

- Implementações mais difíceis – estritamente recursivos

# RECURSÃO

Revisão

FONTE : Ziviani

# Algoritmo Recursivo

- Um método que chama a sim mesmo, direta ou indiretamente, é dito recursivo.

- O uso da recursividade geralmente permite uma descrição mais clara e concisa dos algoritmos, especialmente quando o problema a ser resolvido é recursivo por natureza ou utiliza estruturas recursivas, tais como as árvores.

# Algoritmo Recursivo

- Um compilador implementa um método recursivo por meio de uma **pilha**, na qual são armazenados os dados usados em cada chamada de um método que ainda não terminou de processar.

- Todos os dados não globais vão para a pilha, pois o estado corrente da computação deve ser registrado para que possa ser recuperado de uma nova ativação de um método recursivo, quando a ativação anterior deverá prosseguir.

# Algoritmo Recursivo

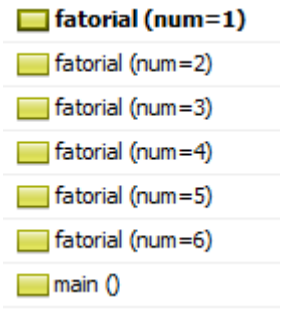
- Quando alcança sua condição de parada, o método retorna para quem chamou, utilizando o endereço de retorno que está no topo da pilha.

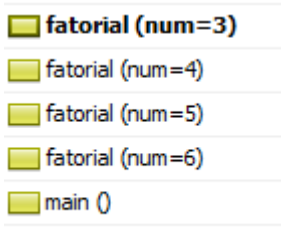
# Algoritmo Recursivo

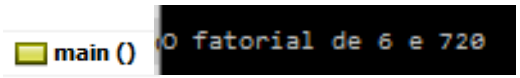
- Exemplo: Cálculo do Fatorial

```
int fatorial(int num)
{
    int fat;
    if (num <= 1)
        return 1;
    else
        fat = num * fatorial (num - 1);
    return fat;
}
```

```
Digite um numero para saber seu fatorial : 6
```




# Recursividade

- Vantagens
  - Simplifica a solução de alguns problemas
  - Algoritmos recursivos são mais compactos para alguns tipos de algoritmo, mais legíveis e mais fáceis de ser compreendidos e implementados.

- Desvantagens
  - Por usarem intensamente a pilha de execução, os algoritmos recursivos tendem a ser mais lentos e a consumir mais memória que os iterativos, porém pode valer a pena sacrificar a eficiência em benefício da clareza.
  - Erros de implementação podem levar a estoure de pilha.

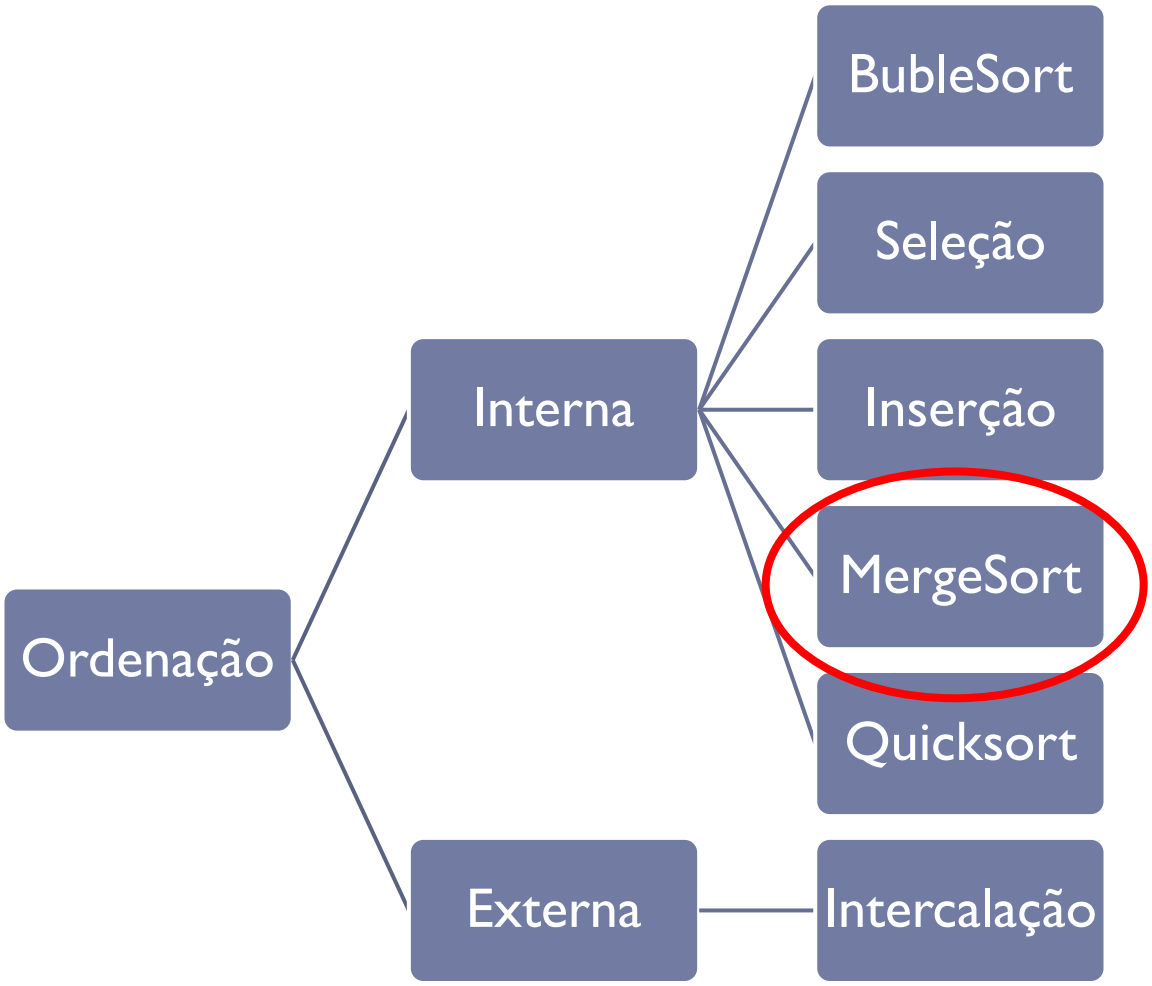
# Recursividade

- Quadro comparativo da execução de algoritmos para o cálculo do Fibonacci.

| n | 10 | 20 | 30 | 50 | 100 |
|---|---|---|---|---|---|
| Recursivo | 8 ms | 1 s | 2 min | 21 dias | $10^9$ anos |
| Iterativo | 0.17 ms | 0.33 ms | 0.5 ms | 0.75 ms | 1.5 ms |

- Evitar uso de recursividade quando existe uma solução óbvia por iteração.

# Classificação dos Métodos de Ordenação



# Ordenação por Fusão ou Intercalação

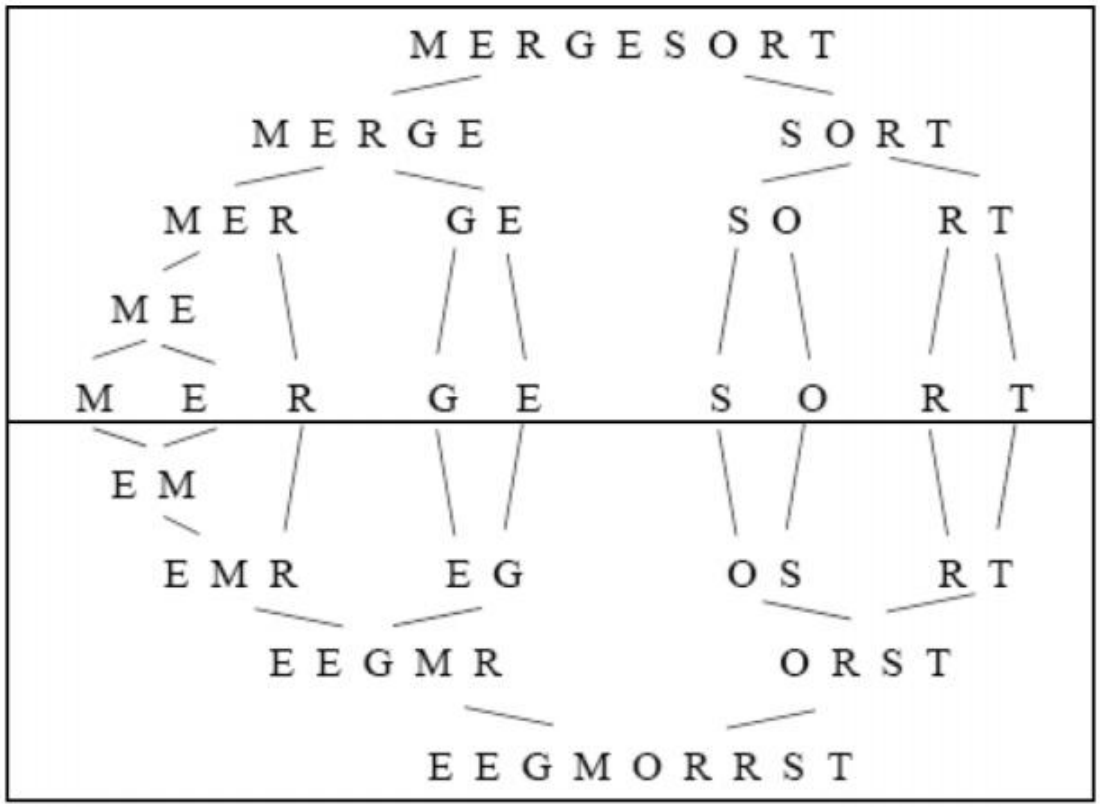
MergeSort

# MergeSort

- **Ideia :** reduzir um problema em problemas menores, resolver cada um destes subproblemas e combinar as soluções parciais para obter a solução do problema original.

- É composto de duas fases:
  - Divisão
    - Divide o vetor original em dois outros de tamanhos menores, recursivamente, até obter vetores de tamanho 1
  - Junção ou Merge
    - Intercala os elementos dos dois vetores ordenados para obter a ordenação total.

# MergeSort




Fase de divisão

Fase de intercalação (merge)

# MergeSort

- O mergeSort não é paralelo!

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| input | M | E | R | G | E | S | O | R | T | E | X | A | M | P | L | E |
| sort left half | E | E | G | M | O | R | R | S | T | E | X | A | M | P | L | E |
| sort right half | E | E | G | M | O | R | R | S | A | E | E | L | M | P | T | X |
| merge results | A | E | E | E | E | G | L | M | M | O | P | R | R | S | T | X |

# MergeSort

**Algorithm 1** MergeSort

```
procedure MergeSort(V, inicio, fim)                ▷ inicio e fim sao indices do vetor
    meio ← ⌊(inicio+fim)/2⌋
    if inicio < fim then
        MergeSort(V, inicio, meio)
        MergeSort(V, meio+1, fim)
        Merge(V, inicio, meio, fim)
    end if
end procedure

procedure Merge(V, inicio, meio, fim)
    v1 ← V[inicio, meio]
    v2 ← V[meio+1, fim]
    vAux                                           ▷ vetor auxiliar com tamanho igual a (fim-inicio)+1
    while v1.size > 0 E v2.size > 0 do
        Compara os elementos de v1 e v2 e ordena-os no vetor auxiliar
    end while
    Copia o resto de v1 ou o resto de v2 para o vetor auxiliar
    Copia o vetor auxiliar para o vetor original
end procedure
```

# MergeSort

- Ordenar o vetor abaixo com o algoritmo MergeSort

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 5 | 3 | 1 | 2 | 4 |

```
procedure MERGESORT(V, inicio, fim)
    meio ← ⌊(inicio+fim)/2⌋
    if inicio < fim then
        MergeSort(V, inicio, meio)
        MergeSort(V, meio+1, fim)
        Merge(V, inicio, meio, fim)
    end if
end procedure
```

# MergeSort

- Chamada 1
  - Início = 0
  - Fim = 4

➡ mergeSort(v, 0, 4) ➡ meio = 2

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 5 | 3 | 1 | 2 | 4 |

# MergeSort

- Chamada 2
  - Início = 0
  - Fim = 2

➡ mergeSort(v, 0, 2) ➡ meio = 1

mergeSort(v, 0, 4) ➡ meio = 2

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 5 | 3 | 1 | 2 | 4 |

# MergeSort

- Chamada 3
  - Início = 0
  - Fim = 1

➡ mergeSort(v, 0, 1) ➡ meio = 0
mergeSort(v, 0, 2) ➡ meio = 1
mergeSort(v, 0, 4) ➡ meio = 2

# MergeSort

- Chamada 4
  - Início = 0
  - Fim = 0

➡ mergeSort(v, 0, 0) ➡ *ordenado*
mergeSort(v, 0, 1) ➡ meio = 0
mergeSort(v, 0, 2) ➡ meio = 1
mergeSort(v, 0, 4) ➡ meio = 2

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 5 | 3 | 1 | 2 | 4 |

- Nesse ponto, a função Merge encontra uma condição de parada da recursão.
- Semanticamente, o vetor é unitário e, portanto, está ordenado.
- A chamada é retirada da pilha de execução.

# MergeSort

- Chamada 5
  - Início = 1
  - Fim = 1

~~mergeSort(v, 0, 0)~~ ➡ ~~*ordenado*~~

➡ mergeSort(v, 0, 1) ➡ meio = 0

mergeSort(v, 0, 2) ➡ meio = 1

mergeSort(v, 0, 4) ➡ meio = 2


- A chamada que está no topo da pilha continua de onde ela parou.

# MergeSort

- Chamada 5
  - Início = 1
  - Fim = 1

mergeSort(v, 1, 1) ➡ *ordenado*

~~mergeSort(v, 0, 0) ➡ *ordenado*~~

mergeSort(v, 0, 1) ➡ meio = 0

mergeSort(v, 0, 2) ➡ meio = 1

mergeSort(v, 0, 4) ➡ meio = 2


- A chamada que está no topo da pilha continua de onde ela parou.

# MergeSort

- Chamada 5
  - Início = 1
  - Fim = 1

~~mergeSort(v, 1, 1)~~ ➡ ~~*ordenado*~~

~~mergeSort(v, 0, 0)~~ ➡ ~~*ordenado*~~

mergeSort(v, 0, 1) ➡ meio = 0

mergeSort(v, 0, 2) ➡ meio = 1

mergeSort(v, 0, 4) ➡ meio = 2


- O vetor é unitário e, portanto, a chamada sai da pilha de execução.
- A chamada que está no topo da pilha continua de onde ela parou.

# MergeSort

merge(v, 0, 0, 1)

~~mergeSort(v, 1, 1)~~ ➡ *~~ordenado~~*

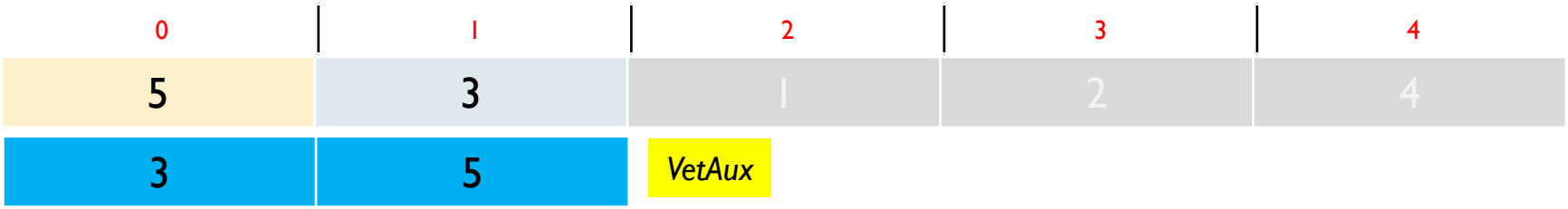
~~mergeSort(v, 0, 0)~~ ➡ *~~ordenado~~*

- Chamada 6
  - Início = 0
  - Meio = 0
  - Fim = 1

mergeSort(v, 0, 1) ➡ meio = 0

mergeSort(v, 0, 2) ➡ meio = 1

mergeSort(v, 0, 4) ➡ meio = 2




- Semanticamente, há dois vetores previamente ordenados
- Nesse caso, cada vetor tem tamanho 1
- Um novo vetor de tamanho 2 é alocado
- Os vetores previamente ordenados são 'juntados' nesse novo vetor

# MergeSort

~~merge(v, 0, 0, 1)~~

~~mergeSort(v, 1, 1)~~ ➡ *~~ordenado~~*

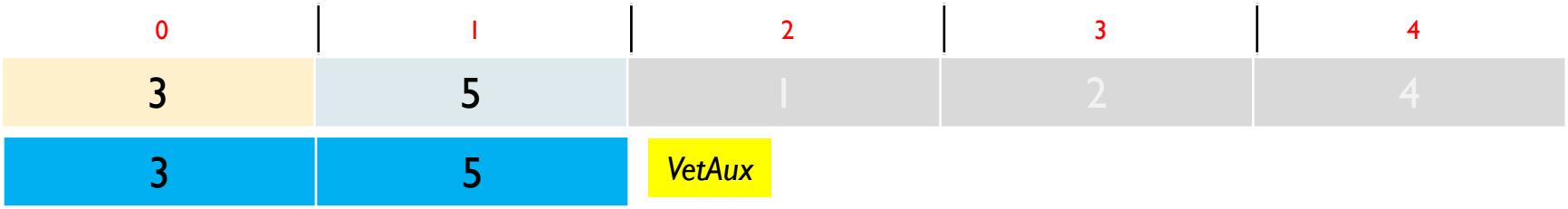
~~mergeSort(v, 0, 0)~~ ➡ *~~ordenado~~*

~~mergeSort(v, 0, 1)~~ ➡ ~~meio = 0~~

mergeSort(v, 0, 2) ➡ meio = 1

mergeSort(v, 0, 4) ➡ meio = 2

- Chamada 6
  - Início = 0
  - Meio = 0
  - Fim = 1




- Posteriormente, o vetor grande ordenado é copiado para o vetor original
- A função merge chega ao fim e é retirada da pilha de execução
- A função que chamou o merge também chega ao fim e é retirada da pilha de execução

# MergeSort

- Chamada 7
  - Início = 0
  - Fim = 2

➡ mergeSort(v, 0, 2) ➡ meio = 1
mergeSort(v, 0, 4) ➡ meio = 2


- A função que está no topo da pilha continua de onde parou

# MergeSort

- Chamada 7
  - Início = 0
  - Fim = 2

➡ mergeSort(v, 2, 2) ➡ *ordenado*

mergeSort(v, 0, 2) ➡ meio = 1

mergeSort(v, 0, 4) ➡ meio = 2


- O vetor é unitário, por isso está ordenado. A função é retirada da pilha.

# MergeSort

- Chamada 7
  - Início = 0
  - Fim = 2

~~mergeSort(v, 2, 2)~~ ➡ ~~*ordenado*~~

➡ mergeSort(v, 0, 2) ➡ meio = 1

mergeSort(v, 0, 4) ➡ meio = 2


- A função que está no topo da pilha continua de onde parou

# MergeSort

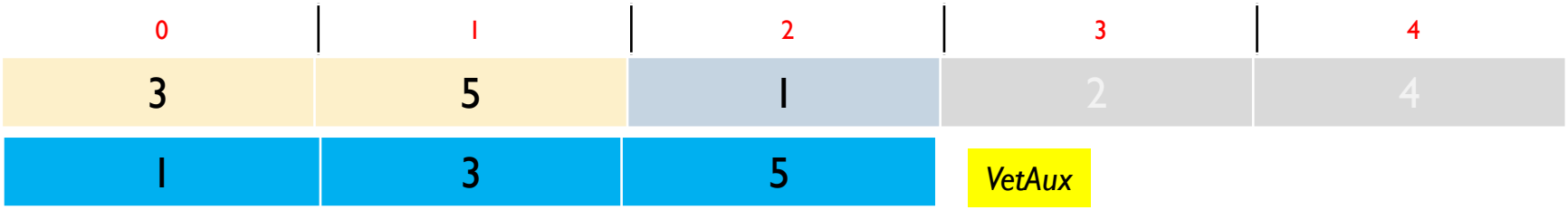
- Chamada 7
  - Início = 0
  - Meio = 1
  - Fim = 2

merge(v, 0, 1, 2)

~~mergeSort(v, 2, 2)~~ ➡ ~~*ordenado*~~

mergeSort(v, 0, 2) ➡ meio = 1

mergeSort(v, 0, 4) ➡ meio = 2




- Semanticamente, há dois vetores previamente ordenados
- Vetor 1: posições 0 e 1
- Vetor 2: posição 2
- Um novo vetor de tamanho 3 é alocado
- Os vetores previamente ordenados são 'juntados' nesse novo vetor

# MergeSort

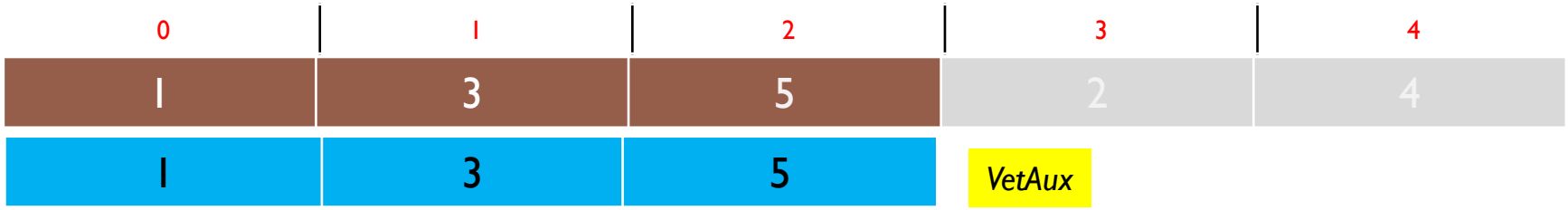
- Chamada 7
  - Início = 0
  - Meio = 0
  - Fim = 1

~~merge(v, 0, 1, 2)~~

~~mergeSort(v, 2, 2)~~ ➡ ~~*ordenado*~~

~~mergeSort(v, 0, 2)~~ ➡ ~~meio = 1~~

➡ mergeSort(v, 0, 4) ➡ meio = 2




- Posteriormente, o vetor grande ordenado é copiado para o vetor original
- A função merge chega ao fim e é retirada da pilha de execução
- A função que chamou o merge também chega ao fim e é retirada da pilha de execução

# MergeSort

- Chamada 7
  - Início = 0
  - Meio = 0
  - Fim = 1

~~merge(v, 0, 1, 2)~~

~~mergeSort(v, 2, 2)~~ ➡ ~~*ordenado*~~

~~mergeSort(v, 0, 2)~~ ➡ ~~meio = 1~~

➡ mergeSort(v, 0, 4) ➡ meio = 2


- Lado esquerdo do vetor já foi todo ordenado.
- O processo se repete no lado direito

# MergeSort

- Chamada 8
  - Início = 3
  - Fim = 4

➡ mergeSort(v, 3, 4) ➡ meio = 3

mergeSort(v, 0, 4) ➡ meio = 2


- Chamada do topo da pilha continua de onde parou

# MergeSort

- Chamada 8
  - Início = 3
  - Fim = 4

~~mergeSort(v, 3, 3)~~ ➡ ~~*ordenado*~~

➡ mergeSort(v, 3, 4) ➡ meio = 3

mergeSort(v, 0, 4) ➡ meio = 2

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 3 | 5 | 2 | 4 |

- Vetor unitário. Chamada sai do topo da pilha
- Função do topo da pilha continua de onde parou

# MergeSort

- Chamada 9
  - Início = 4
  - Fim = 4

➡ mergeSort(v, 4, 4) ➡ *ordenado*

~~mergeSort(v, 3, 3) ➡ *ordenado*~~

mergeSort(v, 3, 4) ➡ meio = 3

mergeSort(v, 0, 4) ➡ meio = 2

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 3 | 5 | 2 | 4 |

- Vetor unitário. Chamada sai do topo da pilha

# MergeSort

- Chamada 10
  - Início = 3
  - Meio = 3
  - Fim = 4

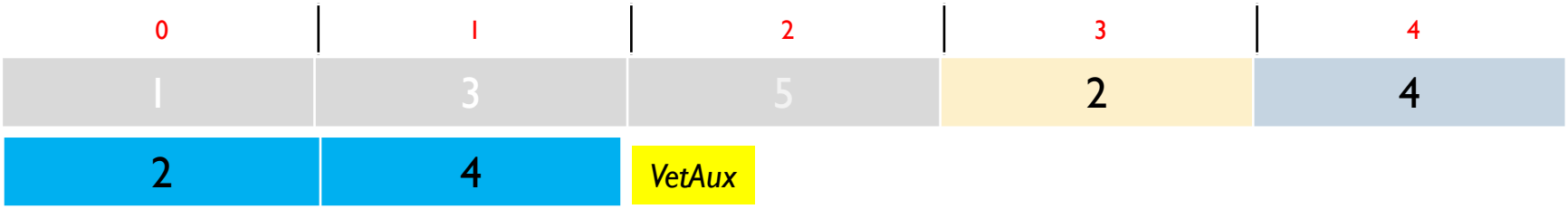
merge(v, 3, 3, 4)

~~mergeSort(v, 4, 4)~~ ➡ *~~ordenado~~*

~~mergeSort(v, 3, 3)~~ ➡ *~~ordenado~~*

mergeSort(v, 3, 4) ➡ meio = 3

mergeSort(v, 0, 4) ➡ meio = 2




- Semanticamente, há dois vetores previamente ordenados
- Vetor 1: posição 3
- Vetor 2: posição 4
- Um novo vetor de tamanho 2 é alocado
- Os vetores previamente ordenados são 'juntados' nesse novo vetor

# MergeSort

- Chamada 10
  - Início = 3
  - Meio = 3
  - Fim = 4

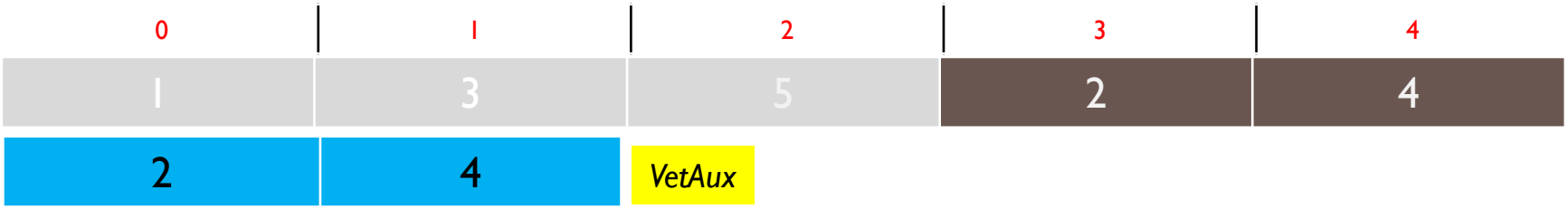
~~merge(v, 3, 3, 4)~~

~~mergeSort(v, 4, 4)~~ ➡ ~~ordenado~~

~~mergeSort(v, 3, 3)~~ ➡ ~~ordenado~~

~~mergeSort(v, 3, 4)~~ ➡ ~~meio = 3~~

mergeSort(v, 0, 4) ➡ meio = 2



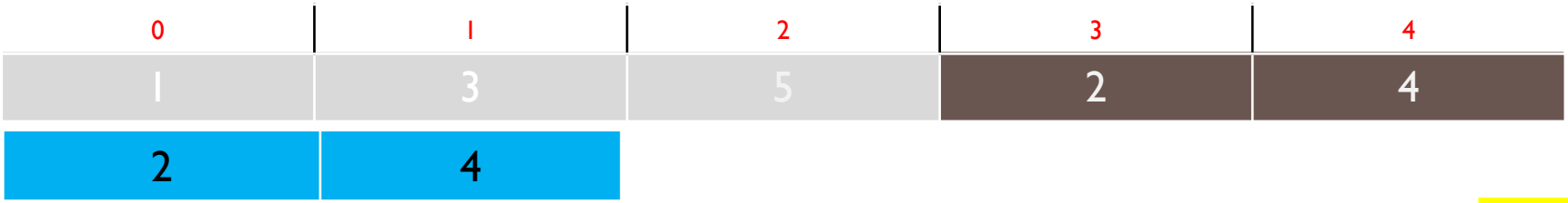

- Posteriormente, o vetor grande ordenado é copiado para o vetor original
- A função merge chega ao fim e é retirada da pilha de execução
- A função que chamou o merge também chega ao fim e é retirada da pilha de execução

# MergeSort

- Chamada 10
  - Início = 3
  - Meio = 3
  - Fim = 4

merge(v, 0, 2, 4)

mergeSort(v, 0, 4) ➡ meio = 2




- Nesse momento, o lado direito do vetor foi ordenado
- Agora é necessário juntar os dois lados previamente ordenados
- Isso acontece quando a primeira chamada chama o merge

# MergeSort

- Chamada 11
  - Início = 0
  - Meio = 2
  - Fim = 4

➡ merge(v, 0, 2, 4)

mergeSort(v, 0, 4) ➡ meio = 2

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 3 | 5 | 2 | 4 |
| 1 | 2 | 3 | 4 | 5 |

*VetAux*

- Semanticamente, há dois vetores previamente ordenados
- Vetor 1: posições 0 a 2
- Vetor 2: posições 3 e 4
- Um novo vetor de tamanho 5 é alocado
- Os vetores previamente ordenados são 'juntados' nesse novo vetor

# MergeSort

- Chamada 11
  - Início = 0
  - Meio = 2
  - Fim = 4

~~merge(v, 0, 2, 4)~~

➡ ~~mergeSort(v, 0, 4) ➡ meio = 2~~


- Posteriormente, o vetor grande ordenado é copiado para o vetor original
- A função merge chega ao fim e é retirada da pilha de execução
- A função que chamou o merge também chega ao fim e é retirada da pilha de execução
- Pilha vazia = Vetor Ordenado!

# MergeSort

- Exercício
  - Fazer o teste de mesa com o seguinte vetor:

| 18 | 7 | 1 | 3 | 8 | 6 |
|---|---|---|---|---|---|

# MergeSort

- Exercício
  - Fazer o teste de mesa com o seguinte vetor:

| 22 | 33 | 55 | 77 | 99 | 11 | 44 | 66 | 88 |
|---|---|---|---|---|---|---|---|---|

# MergeSort

- A eficiência do algoritmo depende de quão eficientemente será a intercalação dos dois vetores (ordenados) em um único vetor ordenado.

- A intercalação pode ser feita fazendo-se, no máximo, (n – 1) comparações, onde *n* é o número total de elementos dos dois vetores originais, ou seja, o algoritmo de intercalação é O(n).

- Como o número de elementos do vetor é reduzido à metade em cada chamada do mergesort, o número total de "rodadas" é $\log_2 n$.

- Assim, a complexidade assintótica do MergeSort é O(nlogn)

# MergeSort

- O MergeSort é considerado um algoritmo ótimo, uma vez que ele tem sempre a mesma complexidade.

- No entanto, o MergeSort ocupa mais espaço na memória para fazer a intercalação dos sub-vetores.

# Exercício

- Pseudo-Código

**Algorithm 1** MergeSort

```
procedure MergeSort(V, inicio, fim)          ▷ inicio e fim sao indices do vetor
    meio ← ⌊(inicio+fim)/2⌋
    if inicio < fim then
        MergeSort(V, inicio, meio)
        MergeSort(V, meio+1, fim)
        Merge(V, inicio, meio, fim)
    end if
end procedure

procedure Merge(V, inicio, meio, fim)
    v1 ← V[inicio, meio]
    v2 ← V[meio+1, fim]
    vAux                                     ▷ vetor auxiliar com tamanho igual a (fim-inicio)+1
    while v1.size > 0 E v2.size > 0 do
        Compara os elementos de v1 e v2 e ordena-os no vetor auxiliar
    end while
    Copia o resto de v1 ou o resto de v2 para o vetor auxiliar
    Copia o vetor auxiliar para o vetor original
end procedure
```

# Trabalho Prático

- Comunicar a formação dos grupos pelo fórum no SIGAA.