

# Ordenação - Métodos eficientes

## Merge Sort

---

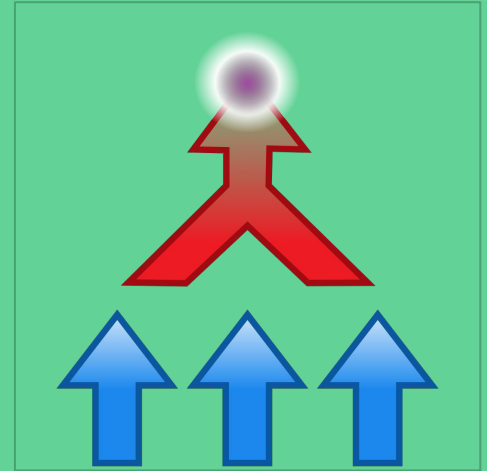
Prof. Joaquim Quinteiro Uchôa  
Profa. Juliana Galvani Gregghi  
Profa. Marluce Rodrigues Pereira  
Profa. Paula Christina Cardoso  
Prof. Renato Ramos da Silva



# Roteiro

- Problema da Intercalação
- Merge Sort
- Merge Sort Iterativo

# Problema da Intercalação



# Merge Sort - Visão Geral

O merge sort é um algoritmo de ordenação eficiente e bastante popular, sendo a implementação preferida em listas encadeadas e como base para ordenação externa.

É um algoritmo recursivo, baseado na ideia de dividir para conquistar, cujas bases estão no problema da intercalação (ou fusão):

*dados dois vetores ordenados  $a$  e  $b$ , de tamanhos  $m$  e  $n$ , respectivamente, intercalá-los de forma a gerar o vetor  $c$ , de tamanho  $m+n$ , contendo os elementos de  $a$  e  $b$  ordenados.*

# Problema da Intercalação - Exemplo

Sejam  $a = [2, 4, 8, 50, 61, 72]$

e  $b = [0, 1, 3, 9, 60, 64, 65, 65, 90, 100]$ .

Então o resultado da intercalação será dado por

$c = [0, 1, 2, 3, 4, 8, 9, 50, 60, 61, 64, 65, 72, 90, 100]$

# Intercalação - Algoritmo - i

```
void intercala_vet(int a[], int b[], int c[], int m, int n){  
    int i = 0; // i percorre a[]  
    int j = 0; // j percorre b[]  
    for (int k = 0; k < m+n; k++) { // k percorre c[]  
        if ((i < m) and (j < n)){ // não terminou a[] e b[]  
            if (a[i] <= b[j]){  
                c[k] = a[i]; // copia elemento de a[] em c[]  
                i++; // avança no vetor a[]  
            } else { // b[j] é menor que a[i]  
                c[k] = b[j];  
                j++;  
            }  
        }  
    }
```



# Intercalação - Algoritmo - i

```
void intercala_vet(int a[], int b[], int c[], int m, int n){  
    int i = 0; // i percorre a[]  
    int j = 0; // j percorre b[]  
    for (int k = 0; k < m+n; k++) { // k percorre c[]  
        if ((i < m) and (j < n)){ // não terminou a[] e b[]  
            if (a[i] <= b[j]){  
                c[k] = a[i]; // copia  
                i++; // avança no vet  
            } else { // b[j] é menor  
                c[k] = b[j];  
                j++;  
            }  
        }  
    }
```

Após comparação, o menor elemento entre  $a[i]$  ou  $b[j]$  é copiado para o vetor  $c$ , avançando o índice de onde foi copiado.



# Intercalação - Algoritmo - ii

```
        c[k] = b[j];  
        j++;  
    }  
} else if (i == m) { // terminou a[], copia b[]  
    c[k] = b[j];  
    j++;  
} else { // terminou b[], copia a[]  
    c[k] = a[i];  
    i++;  
}  
} // fim do for  
}
```





# Intercalação - Algoritmo - ii

```
        c[k] = b[j];  
        j++;  
    }  
} else if (i == m) { // terminou a[], copia b[]  
    c[k] = b[j];  
    j++;  
} else { // terminou b[]  
    c[k] = a[i];  
    i++;  
}  
} // fim do for  
}
```

Caso um vetor termine, basta copiar os dados do outro vetor...



# Intercalação de Trechos de um Vetor

Não é interessante, para o processo de ordenação, ficar copiando de um vetor para outro em várias etapas desse processo. Assim, para o merge sort, é melhor intercalar trechos de um mesmo vetor:

*Dados trechos ordenados  $a[p \dots q-1]$  e  $a[q \dots r]$ , construir  $a[p \dots r]$  também ordenado. Nesse caso  $p$  é o início,  $q$  é o meio e  $r$  é o fim do trecho a ser intercalado.*

# Intercalação de Trechos - Algoritmo - i

```
void intercala(int a[], int inicio, int meio, int fim) {  
    int i = inicio, j = meio + 1;  
    int tamanho = fim - inicio + 1;  
    int aux[tamanho]; // vetor auxiliar  
    for (int k=0; k < tamanho; k++) {  
        if ((i <= meio) and (j <= fim)){  
            if (a[i] <= a[j]){  
                aux[k] = a[i]; // copia trecho1 em aux[]  
                i++; // avança em trecho1  
            } else { //  
                aux[k] = a[j]; // copia trecho2 em aux[]  
                j++; // avança em trecho2  
            }  
        }  
    }  
}
```



# Intercalação de Trechos - Algoritmo - i

```
    } else if (i > meio) { // terminou o trecho1
        aux[k] = a[j];
        j++;
    } else { // terminou o trecho2
        aux[k] = a[i];
        i++;
    }
}
// terminando: copiar de aux[] em a[inicio:fim]
for (int k=0; k < tamanho; k++){
    a[inicio + k] = aux[k];
}
}
```

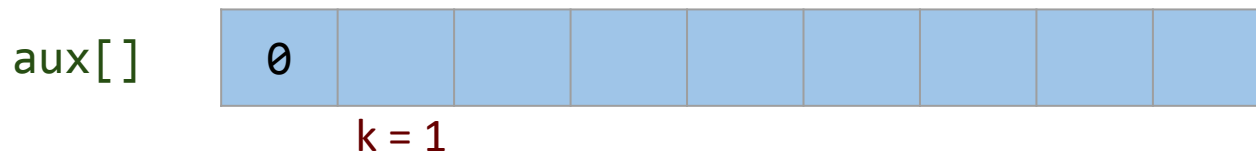
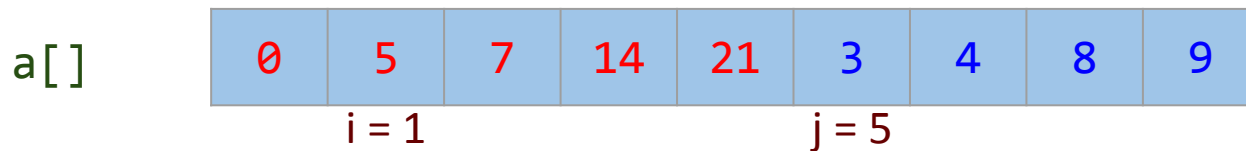
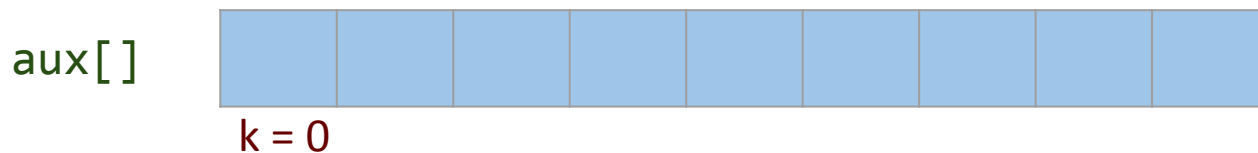
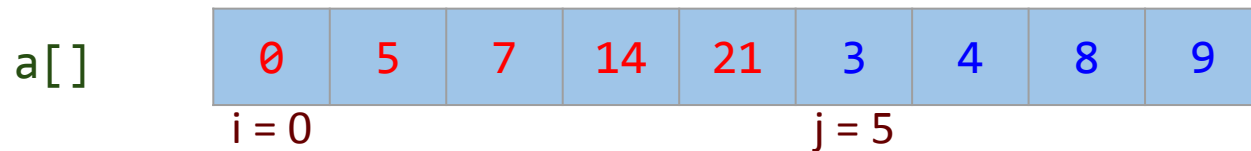


# Intercalação de Trechos - Exemplo - i

Considere o vetor  $a = [0, 5, 7, 14, 21, 3, 4, 8, 9]$ , que possui dois trechos ordenados, o primeiro da posição 0 a 4, e o segundo da posição 5 a 8.

Considere a chamada  
`intercalat(int a[], 0, 4, 8)`

# Intercalação de Trechos - Exemplo - ii



# Intercalação de Trechos - Exemplo - iii

a[ ]

0	5	7	14	21	3	4	8	9
---	---	---	----	----	---	---	---	---

i = 1 j = 6

aux[ ]

0	3							
---	---	--	--	--	--	--	--	--

k = 2

a[ ]

0	5	7	14	21	3	4	8	9
---	---	---	----	----	---	---	---	---

i = 1 j = 7

aux[ ]

0	3	4						
---	---	---	--	--	--	--	--	--

k = 3

# Intercalação de Trechos - Exemplo - iii

a[]

0	5	7	14	21	3	4	8	9
---	---	---	----	----	---	---	---	---

i = 2 j = 7

aux[]

0	3	4	5					
---	---	---	---	--	--	--	--	--

k = 4

a[]

0	5	7	14	21	3	4	8	9
---	---	---	----	----	---	---	---	---

i = 3 j = 7

aux[]

0	3	4	5	7				
---	---	---	---	---	--	--	--	--

k = 5



# Intercalação de Trechos - Exemplo - iv

a[]

0	5	7	14	21	3	4	8	9
---	---	---	----	----	---	---	---	---

i = 3 j = 8

aux[]

0	3	4	5	7	8			
---	---	---	---	---	---	--	--	--

k = 6

a[]

0	5	7	14	21	3	4	8	9
---	---	---	----	----	---	---	---	---

i = 3 j = 9

aux[]

0	3	4	5	7	8	9		
---	---	---	---	---	---	---	--	--

k = 7

# Intercalação de Trechos - Exemplo - v

a[]

0	5	7	14	21	3	4	8	9
---	---	---	----	----	---	---	---	---

i = 4 j = 9

aux[]

0	3	4	5	7	8	9	14	
---	---	---	---	---	---	---	----	--

k = 8

a[]

0	5	7	14	21	3	4	8	9
---	---	---	----	----	---	---	---	---

i = 5 j = 9

aux[]

0	3	4	5	7	8	9	14	21
---	---	---	---	---	---	---	----	----

for encerrado

# Reescrevendo a Intercala()

Existem várias implementações para a função de intercalação, em geral com a mesma eficiência. Deixamos a cargo dos interessados a pesquisa por alternativas.

Entretanto, uma versão reescrita é bastante popular, sendo aqui apresentada. Ela é muito semelhante à versão apresentada, com diferenças pontuais apenas.

# Reescrita da Intercala() - i

```
// intercala v[p..q-1] e v[q..r] em v[p..r]
void intercala(int v[], int p, int q, int r){
    int i = p, j = q;
    int tamanho = r - p + 1;
    int w[tamanho]; // vetor auxiliar
    int k = 0;
    while ((i < q) and (j <= r)) {
        if (v[i] <= v[j]) {
            w[k++] = v[i++]; /* w[k] = v[i]; k++; i++; */
        } else {
            w[k++] = v[j++]; /* w[k] = v[j]; k++; j++; */
        }
    }
}
```



# Reescrita da Intercala() - i

```
// intercala v[p..q-1] e v[q..r] em v[p..r]
void intercala(int v[], int p, int q, int r){
    int i = p, j = q;
    int tamanho = r - p + 1;
    int w[tamanho]; // vetor auxiliar
    int k = 0;
    while ((i < q) and (j <= r)) {
        if (v[i] <= v[j]) {
            w[k++] = v[i++]; /* w[k] = v[i] */
        } else {
            w[k++] = v[j++]; /* w[k] = v[j] */
        }
    }
}
```



Usando while, consegue fazer menos testes. Há um pequeno aumento de eficiência.

# Reescrita da Intercala() - ii

```
// terminou um dos vetores, agora copia o outro
```

```
while (i < q) {  
    w[k++] = v[i++];  
}
```

```
while (j <= r) {  
    w[k++] = v[j++];  
}
```

```
// agora copiamos do vetor auxiliar aux[] em v[p:r]
```

```
for (int m = 0; m < tamanho; m++){  
    v[p + m] = w[m];  
}
```

```
}
```



# Reescrita da Intercala() - ii

```
// terminou um dos vetores, agora copia o outro
```

```
while (i < q) {  
    w[k++] = v[i++];  
}
```

```
while (j <= r) {  
    w[k++] = v[j++];  
}
```

```
// agora copiamos do vetor auxiliar aux[] em v[p:r]
```

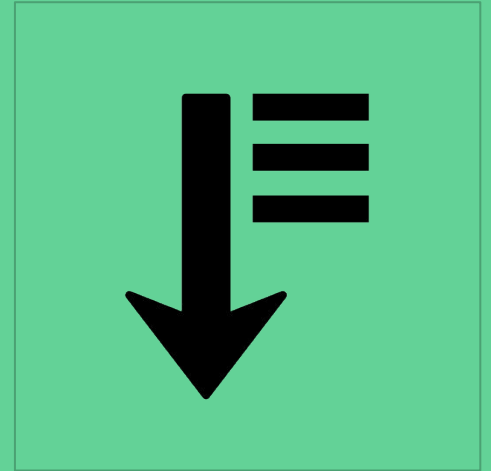
```
for (int m = 0; m < tamanho; m++){  
    v[p + m] = w[m];  
}
```

```
}
```

Usando while, passar todos os elementos já com o teste. Há um pequeno aumento de eficiência.



# Merge Sort





# Merge Sort

O merge sort consiste em aplicar, recursivamente, o processo de divisão do vetor original em metades, até que cada metade tenha um único elemento, intercalando as metades ordenadas após isso:

```
void mergesort(int a[], int inicio, int fim){  
    int meio;  
    if (inicio < fim) {  
        meio = (inicio + fim)/2;  
        mergesort(a, inicio, meio);  
        mergesort(a, meio+1, fim);  
        intercala(a, inicio, meio, fim);  
    }  
}
```

# Merge Sort

O merge sort consiste em aplicar, recursivamente, o processo de divisão do vetor original em metades, até que cada metade tenha um único elemento, intercalando as metades ordenadas após isso:

```
void mergesort(int a[], int inicio, int fim){  
    int meio;  
    if (inicio < fim) {  
        meio =  
        mergeso  
        mergeso  
        interca  
    }  
}
```

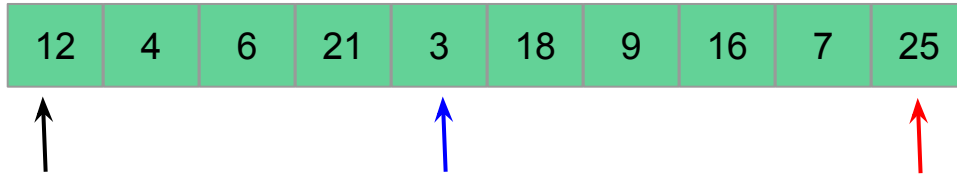
Procedimento é chamado inicialmente passando o vetor, sua posição inicial e sua posição final.

**inicio** = 0

**fim** = 9

**meio** =  $(0 + 9) / 2 = 4$

**mergesort** (a, 0, 9)



**inicio < fim ? Sim!**

### **Chamadas Recursivas**

**mergesort** (a, 0, 9)  $\Leftarrow$

**mergesort** (a, 0, 4)

**mergesort** (a, 5, 9)

**intercala** (a, 0, 4, 9)

inicio = 0

fim = 4

meio =  $(0 + 4) / 2 = 2$

mergesort (a, 0, 4)

12	4	6	21	3	18	9	16	7	25
----	---	---	----	---	----	---	----	---	----



inicio < fim ? Sim!

### Chamadas Recursivas

mergesort (a, 0, 9)

mergesort (a, 0, 4) ←

mergesort (a, 0, 2)

mergesort (a, 3, 4)

intercala (a, 0, 2, 4)

mergesort (a, 5, 9)

intercala (a, 0, 4, 9)

**inicio** = 0

**fim** = 2

**meio** =  $(0 + 2) / 2 = 1$

**mergesort** (a, 0, 2)

12	4	6	21	3	18	9	16	7	25
----	---	---	----	---	----	---	----	---	----



**inicio < fim ? Sim!**

### **Chamadas Recursivas**

**mergesort** (a, 0, 9)

**mergesort** (a, 0, 4)

**mergesort** (a, 0, 2)  $\Leftarrow$

**mergesort** (a, 0, 1)

**mergesort** (a, 2, 2)

**intercala** (a, 0, 2, 2)

**mergesort** (a, 3, 4)

**intercala** (a, 0, 2, 4)

**mergesort** (a, 5, 9)

**intercala** (a, 0, 4, 9)

inicio = 0

fim = 1

meio =  $(0 + 1) / 2 = 0$

mergesort (a, 0, 1)

12	4	6	21	3	18	9	16	7	25
----	---	---	----	---	----	---	----	---	----



inicio < fim ? Sim!

## Chamadas Recursivas

mergesort (a, 0, 9)

mergesort (a, 0, 4)

mergesort (a, 0, 2)

mergesort (a, 0, 1) ←

mergesort (a, 0, 0)

mergesort (a, 1, 1)

intercala (a, 0, 0, 1)

mergesort (a, 2, 2)

intercala (a, 0, 2, 2)

mergesort (a, 3, 4)

intercala (a, 0, 2, 4)

mergesort (a, 5, 9)

intercala (a, 0, 4, 9)

**inicio** = 0

**fim** = 0

mergesort (a, 0, 0)

12	4	6	21	3	18	9	16	7	25
----	---	---	----	---	----	---	----	---	----



**inicio < fim ? Não!**

## Chamadas Recursivas

mergesort (a, 0, 9)

mergesort (a, 0, 4)

mergesort (a, 0, 2)

mergesort (a, 0, 1)

mergesort (a, 0, 0) ⇐

mergesort (a, 1, 1)

intercala (a, 0, 0, 1)

mergesort (a, 2, 2)

intercala (a, 0, 2, 2)

mergesort (a, 3, 4)

intercala (a, 0, 2, 4)

mergesort (a, 5, 9)

intercala (a, 0, 4, 9)

**inicio** = 0

**fim** = 1

mergesort (a, 1, 1)

12	4	6	21	3	18	9	16	7	25
----	---	---	----	---	----	---	----	---	----



**inicio < fim ? Não!**

## Chamadas Recursivas

mergesort (a, 0, 9)

mergesort (a, 0, 4)

mergesort (a, 0, 2)

mergesort (a, 0, 1)

~~mergesort (a, 0, 0)~~

mergesort (a, 1, 1) ⇐

intercala (a, 0, 0, 1)

mergesort (a, 2, 2)

intercala (a, 0, 2, 2)

mergesort (a, 3, 4)

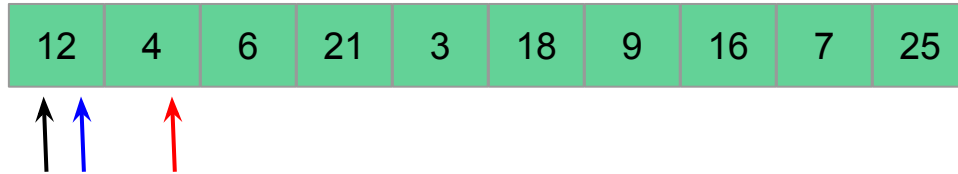
intercala (a, 0, 2, 4)

mergesort (a, 5, 9)

intercala (a, 0, 4, 9)



intercala (a, 0, 0, 1)



## Chamadas Recursivas

mergesort (a, 0, 9)

mergesort (a, 0, 4)

mergesort (a, 0, 2)

mergesort (a, 0, 1)

~~mergesort (a, 0, 0)~~

~~mergesort (a, 1, 1)~~

intercala (a, 0, 0, 1) ←

mergesort (a, 2, 2)

intercala (a, 0, 2, 2)

mergesort (a, 3, 4)

intercala (a, 0, 2, 4)

mergesort (a, 5, 9)

intercala (a, 0, 4, 9)

mergesort (a, 0, 1)

4	12	6	21	3	18	9	16	7	25
---	----	---	----	---	----	---	----	---	----

## Chamadas Recursivas

mergesort (a, 0, 9)

mergesort (a, 0, 4)

mergesort (a, 0, 2)

mergesort (a, 0, 1) ←

~~mergesort (a, 0, 0)~~

~~mergesort (a, 1, 1)~~

~~intercala (a, 0, 0, 1)~~

mergesort (a, 2, 2)

intercala (a, 0, 2, 2)

mergesort (a, 3, 4)

intercala (a, 0, 2, 4)

mergesort (a, 5, 9)

intercala (a, 0, 4, 9)

inicio = 0

fim = 2

mergesort (a, 2, 2)

4	12	6	21	3	18	9	16	7	25
---	----	---	----	---	----	---	----	---	----

inicio < fim ? Não!

### Chamadas Recursivas

mergesort (a, 0, 9)

mergesort (a, 0, 4)

mergesort (a, 0, 2)

~~mergesort (a, 0, 1)~~

mergesort (a, 2, 2) ←

intercala (a, 0, 1, 2)

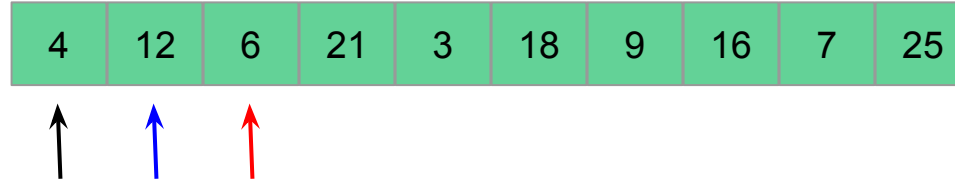
mergesort (a, 3, 4)

intercala (a, 0, 2, 4)

mergesort (a, 5, 9)

intercala (a, 0, 4, 9)

intercala (a, 0, 1, 2)



## Chamadas Recursivas

mergesort (a, 0, 9)

mergesort (a, 0, 4)

mergesort (a, 0, 2)

~~mergesort (a, 0, 1)~~

~~mergesort (a, 2, 2)~~

intercala (a, 0, 1, 2) ←

mergesort (a, 3, 4)

intercala (a, 0, 2, 4)

mergesort (a, 5, 9)

intercala (a, 0, 4, 9)

mergesort (a, 0, 2)

4	6	12	21	3	18	9	16	7	25
---	---	----	----	---	----	---	----	---	----

### Chamadas Recursivas

mergesort (a, 0, 9)

mergesort (a, 0, 4)

mergesort (a, 0, 2)  $\Leftarrow$

~~mergesort (a, 0, 1)~~

~~mergesort (a, 2, 2)~~

~~intercala (a, 0, 1, 2)~~

mergesort (a, 3, 4)

intercala (a, 0, 2, 4)

mergesort (a, 5, 9)

intercala (a, 0, 4, 9)

**inicio** = 3

**fim** = 4

**meio** =  $(3 + 4) / 2 = 3$

**mergesort (a, 3, 4)**

4	6	12	21	3	18	9	16	7	25
---	---	----	----	---	----	---	----	---	----

**inicio < fim ? Sim!**

### **Chamadas Recursivas**

**mergesort (a, 0, 9)**

**mergesort (a, 0, 4)**

~~mergesort (a, 0, 2)~~

~~mergesort (a, 3, 4)~~ ←

**mergesort (a, 3, 3)**

**mergesort (a, 4, 4)**

**intercala (a, 3, 3, 4)**

~~intercala (a, 0, 2, 4)~~

**mergesort (a, 5, 9)**

**intercala (a, 0, 4, 9)**

inicio = 3

**fim** = 3

mergesort (a, 3, 3)

4	6	12	21	3	18	9	16	7	25
---	---	----	----	---	----	---	----	---	----

inicio < fim ? Não!

### Chamadas Recursivas

mergesort (a, 0, 9)

mergesort (a, 0, 4)

~~mergesort (a, 0, 2)~~

mergesort (a, 3, 4)

mergesort (a, 3, 3) ←

mergesort (a, 4, 4)

intercala (a, 3, 3, 4)

intercala (a, 0, 2, 4)

mergesort (a, 5, 9)

intercala (a, 0, 4, 9)

inicio = 4

fim = 4

mergesort (a, 4, 4)

4	6	12	21	3	18	9	16	7	25
---	---	----	----	---	----	---	----	---	----

inicio < fim ? Não!

### Chamadas Recursivas

mergesort (a, 0, 9)

mergesort (a, 0, 4)

~~mergesort (a, 0, 2)~~

mergesort (a, 3, 4)

~~mergesort (a, 3, 3)~~

mergesort (a, 4, 4) ⇐

intercala (a, 3, 3, 4)

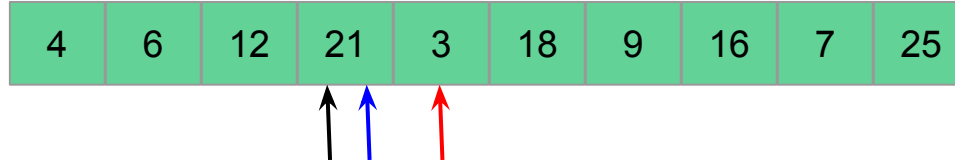
intercala (a, 0, 2, 4)

mergesort (a, 5, 9)

intercala (a, 0, 4, 9)



intercala (a, 3, 3, 4)



### Chamadas Recursivas

mergesort (a, 0, 9)

mergesort (a, 0, 4)

~~mergesort (a, 0, 2)~~

mergesort (a, 3, 4)

~~mergesort (a, 3, 3)~~

~~mergesort (a, 4, 4)~~

intercala (a, 3, 3, 4) ←

intercala (a, 0, 2, 4)

mergesort (a, 5, 9)

mergesort (a, 0, 4, 9)

mergesort (a, 3, 4)

4	6	12	3	21	18	9	16	7	25
---	---	----	---	----	----	---	----	---	----

### Chamadas Recursivas

mergesort (a, 0, 9)

mergesort (a, 0, 4)

~~mergesort (a, 0, 2)~~

mergesort (a, 3, 4)  $\Leftarrow$

~~mergesort (a, 3, 3)~~

~~mergesort (a, 4, 4)~~

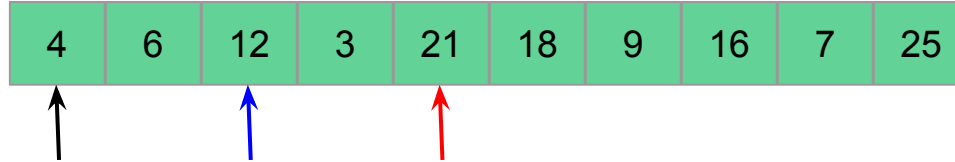
~~intercala (a, 3, 3, 4)~~

intercala (a, 0, 2, 4)

mergesort (a, 5, 9)

intercala (a, 0, 4, 9)

intercala (a, 0, 2, 4)



### Chamadas Recursivas

mergesort (a, 0, 9)

mergesort (a, 0, 4)

~~mergesort (a, 0, 2)~~

~~mergesort (a, 3, 4)~~

intercala (a, 0, 2, 4) ⇐

mergesort (a, 5, 9)

intercala (a, 0, 4, 9)

mergesort (a, 0, 4)

3	4	6	12	21	18	9	16	7	25
---	---	---	----	----	----	---	----	---	----

### Chamadas Recursivas

mergesort (a, 0, 9)

mergesort (a, 0, 4) ←

~~mergesort (a, 0, 2)~~

~~mergesort (a, 3, 4)~~

~~intercala (a, 0, 2, 4)~~

mergesort (a, 5, 9)

intercala (a, 0, 4, 9)

**inicio** = 5

**fim** = 9

**meio** =  $(5 + 9) / 2 = 7$

**mergesort (a, 5, 9)**

3	4	6	12	21	18	9	16	7	25
---	---	---	----	----	----	---	----	---	----

**inicio < fim ? Sim!**

### **Chamadas Recursivas**

**mergesort (a, 0, 9)**

~~**mergesort (a, 0, 4)**~~

**mergesort (a, 5, 9) ←**

**mergesort (a, 5, 7)**

**mergesort (a, 8, 9)**

**intercala (a, 5, 7, 9)**

**intercala (a, 0, 4, 9)**

**inicio** = 5

**fim** = 7

**meio** =  $(5 + 7) / 2 = 6$

**mergesort (a, 5, 7)**

3	4	6	12	21	18	9	16	7	25
---	---	---	----	----	----	---	----	---	----

**inicio < fim ? Sim!**

### **Chamadas Recursivas**

**mergesort (a, 0, 9)**

~~**mergesort (a, 0, 4)**~~

**mergesort (a, 5, 9)**

**mergesort (a, 5, 7) ←**

**mergesort (a, 5, 6)**

**mergesort (a, 7, 7)**

**intercala (a, 5, 6, 7)**

**mergesort (a, 8, 9)**

**intercala (a, 5, 7, 9)**

**intercala (a, 0, 4, 9)**

inicio = 5

fim = 6

meio =  $(5 + 6) / 2 = 5$

mergesort (a, 5, 6)

3	4	6	12	21	18	9	16	7	25
---	---	---	----	----	----	---	----	---	----

inicio < fim ? Sim!

## Chamadas Recursivas

mergesort (a, 0, 9)

~~mergesort (a, 0, 4)~~

mergesort (a, 5, 9)

mergesort (a, 5, 7)

mergesort (a, 5, 6) ⇐

mergesort (a, 5, 5)

mergesort (a, 6, 6)

intercala (a, 5, 5, 6)

mergesort (a, 7, 7)

intercala (a, 5, 6, 7)

mergesort (a, 8, 9)

intercala (a, 5, 7, 9)

intercala (a, 0, 4, 9)

inicio = 5

fim = 5

mergesort(a, 5, 5)

3	4	6	12	21	18	9	16	7	25
---	---	---	----	----	----	---	----	---	----

inicio < fim ? Não!

## Chamadas Recursivas

mergesort(a, 0, 9)

~~mergesort(a, 0, 4)~~

mergesort(a, 5, 9)

mergesort(a, 5, 7)

mergesort(a, 5, 6)

mergesort(a, 5, 5) ⇐

mergesort(a, 6, 6)

intercala(a, 5, 5, 6)

mergesort(a, 7, 7)

intercala(a, 5, 6, 7)

mergesort(a, 8, 9)

intercala(a, 5, 7, 9)

intercala(a, 0, 4, 9)



inicio = 6

fim = 6

mergesort(a, 6, 6)

3	4	6	12	21	18	9	16	7	25
---	---	---	----	----	----	---	----	---	----

inicio < fim ? Não!

## Chamadas Recursivas

mergesort(a, 0, 9)

~~mergesort(a, 0, 4)~~

mergesort(a, 5, 9)

mergesort(a, 5, 7)

mergesort(a, 5, 6)

~~mergesort(a, 5, 5)~~

mergesort(a, 6, 6) ⇐

intercala(a, 5, 5, 6)

mergesort(a, 7, 7)

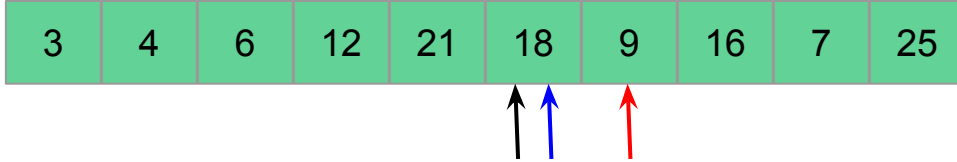
intercala(a, 5, 6, 7)

mergesort(a, 8, 9)

intercala(a, 5, 7, 9)

intercala(a, 0, 4, 9)

intercala (a, 5, 5, 6)



## Chamadas Recursivas

mergesort (a, 0, 9)

~~mergesort (a, 0, 4)~~

~~mergesort (a, 5, 9)~~

mergesort (a, 5, 7)

mergesort (a, 5, 6)

~~mergesort (a, 5, 5)~~

~~mergesort (a, 6, 6)~~

intercala (a, 5, 5, 6) ←

mergesort (a, 7, 7)

intercala (a, 5, 6, 7)

mergesort (a, 8, 9)

intercala (a, 5, 7, 9)

intercala (a, 0, 4, 9)

mergesort (a, 5, 6)

3	4	6	12	21	9	18	16	7	25
---	---	---	----	----	---	----	----	---	----

## Chamadas Recursivas

mergesort (a, 0, 9)

~~mergesort (a, 0, 4)~~

~~mergesort (a, 5, 9)~~

mergesort (a, 5, 7)

~~mergesort (a, 5, 6) ⇐~~

~~mergesort (a, 5, 5)~~

~~mergesort (a, 6, 6)~~

~~intercala (a, 5, 5, 6)~~

~~mergesort (a, 7, 7)~~

~~intercala (a, 5, 6, 7)~~

mergesort (a, 8, 9)

intercala (a, 5, 7, 9)

intercala (a, 0, 4, 9)

inicio = 7

**fim** = 7

mergesort (a, 7, 7)

3	4	6	12	21	9	18	16	7	25
---	---	---	----	----	---	----	----	---	----

inicio < fim ? Não!

### Chamadas Recursivas

mergesort (a, 0, 9)

~~mergesort (a, 0, 4)~~

mergesort (a, 5, 9)

mergesort (a, 5, 7)

~~mergesort (a, 5, 6)~~

mergesort (a, 7, 7) ⇐

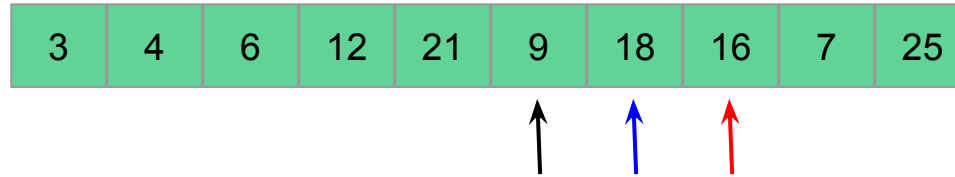
intercala (a, 5, 6, 7)

mergesort (a, 8, 9)

intercala (a, 5, 7, 9)

intercala (a, 0, 4, 9)

intercala (a, 5, 6, 7)



### Chamadas Recursivas

mergesort (a, 0, 9)

~~mergesort (a, 0, 4)~~

mergesort (a, 5, 9)

mergesort (a, 5, 7)

~~mergesort (a, 5, 6)~~

~~mergesort (a, 7, 7)~~

intercala (a, 5, 6, 7) ⇐

mergesort (a, 8, 9)

intercala (a, 5, 7, 9)

intercala (a, 0, 4, 9)

**mergesort (a, 5, 7)**

3	4	6	12	21	9	16	18	7	25
---	---	---	----	----	---	----	----	---	----

### **Chamadas Recursivas**

**mergesort (a, 0, 9)**

~~**mergesort (a, 0, 4)**~~

~~**mergesort (a, 5, 9)**~~

**mergesort (a, 5, 7) ⇐**

~~**mergesort (a, 5, 6)**~~

~~**mergesort (a, 7, 7)**~~

~~**intercala (a, 5, 6, 7)**~~

**mergesort (a, 8, 9)**

**intercala (a, 5, 7, 9)**

**intercala (a, 0, 4, 9)**

**inicio** = 8

**fim** = 9

**meio** =  $(8 + 9) / 2 = 8$

**mergesort** (a, 8, 9)

3	4	6	12	21	9	16	18	7	25
---	---	---	----	----	---	----	----	---	----

**inicio < fim ? Sim!**

### **Chamadas Recursivas**

**mergesort** (a, 0, 9)

~~**mergesort** (a, 0, 4)~~

**mergesort** (a, 5, 9)

~~**mergesort** (a, 5, 7)~~

**mergesort** (a, 8, 9) ←

**mergesort** (a, 8, 8)

**mergesort** (a, 9, 9)

**intercala** (a, 8, 8, 9)

**intercala** (a, 5, 7, 9)

**intercala** (a, 0, 4, 9)

inicio = 8

fim = 8

mergesort (a, 8, 8)

3	4	6	12	21	9	16	18	7	25
---	---	---	----	----	---	----	----	---	----

inicio < fim ? Não!

### Chamadas Recursivas

mergesort (a, 0, 9)

~~mergesort (a, 0, 4)~~

mergesort (a, 5, 9)

~~mergesort (a, 5, 7)~~

mergesort (a, 8, 9)

mergesort (a, 8, 8) ←

mergesort (a, 9, 9)

intercala (a, 8, 8, 9)

intercala (a, 5, 7, 9)

intercala (a, 0, 4, 9)



inicio = 9

fim = 9

mergesort (a, 9, 9)

3	4	6	12	21	9	16	18	7	25
---	---	---	----	----	---	----	----	---	----

inicio < fim ? Não!

### Chamadas Recursivas

mergesort (a, 0, 9)

~~mergesort (a, 0, 4)~~

mergesort (a, 5, 9)

~~mergesort (a, 5, 7)~~

mergesort (a, 8, 9)

~~mergesort (a, 8, 8)~~

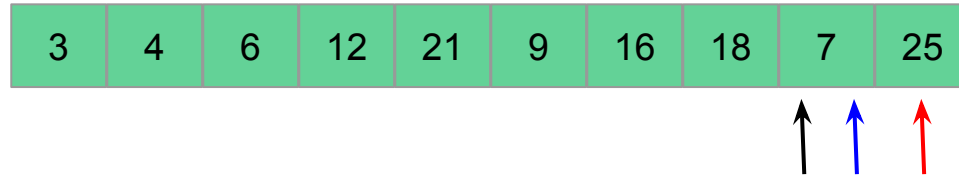
mergesort (a, 9, 9) ⇐

intercala (a, 8, 8, 9)

intercala (a, 5, 7, 9)

intercala (a, 0, 4, 9)

intercala (a, 8, 8, 9)



### Chamadas Recursivas

mergesort (a, 0, 9)

~~mergesort (a, 0, 4)~~

mergesort (a, 5, 9)

~~mergesort (a, 5, 7)~~

mergesort (a, 8, 9)

~~mergesort (a, 8, 8)~~

~~mergesort (a, 9, 9)~~

intercala (a, 8, 8, 9) ⇐

intercala (a, 5, 7, 9)

intercala (a, 0, 4, 9)

mergesort (a, 8, 9)

3	4	6	12	21	9	16	18	7	25
---	---	---	----	----	---	----	----	---	----

### Chamadas Recursivas

mergesort (a, 0, 9)

~~mergesort (a, 0, 4)~~

mergesort (a, 5, 9)

~~mergesort (a, 5, 7)~~

mergesort (a, 8, 9) ←

~~mergesort (a, 8, 8)~~

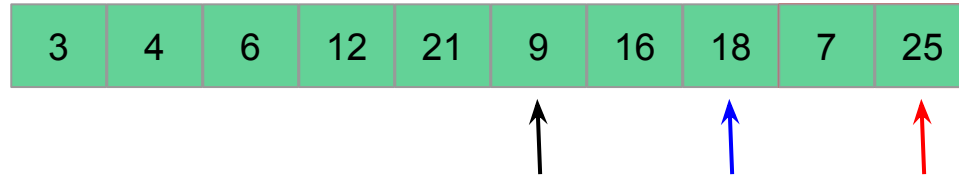
~~mergesort (a, 9, 9)~~

~~intercala (a, 8, 8, 9)~~

intercala (a, 5, 7, 9)

intercala (a, 0, 4, 9)

intercala (a, 5, 7, 9)



### Chamadas Recursivas

mergesort (a, 0, 9)

~~mergesort (a, 0, 4)~~

mergesort (a, 5, 9)

~~mergesort (a, 5, 7)~~

~~mergesort (a, 8, 9)~~

intercala (a, 5, 7, 9) ⇐

intercala (a, 0, 4, 9)

**mergesort (a, 5, 9)**

3	4	6	12	21	7	9	16	18	25
---	---	---	----	----	---	---	----	----	----

### **Chamadas Recursivas**

**mergesort (a, 0, 9)**

~~**mergesort (a, 0, 4)**~~

**mergesort (a, 5, 9) ←**

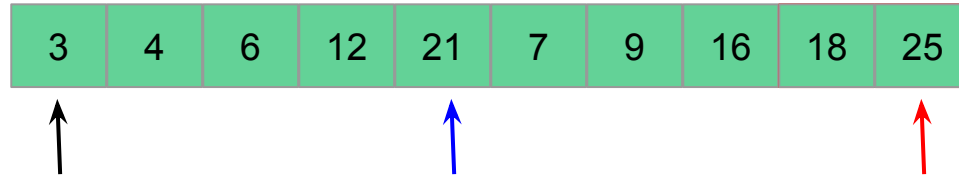
~~**mergesort (a, 5, 7)**~~

~~**mergesort (a, 8, 9)**~~

~~**intercala (a, 5, 7, 9)**~~

**intercala (a, 0, 4, 9)**

intercala (a, 0, 4, 9)



### Chamadas Recursivas

mergesort (a, 0, 9)

~~mergesort (a, 0, 4)~~

~~mergesort (a, 5, 9)~~

intercala (a, 0, 4, 9) ←

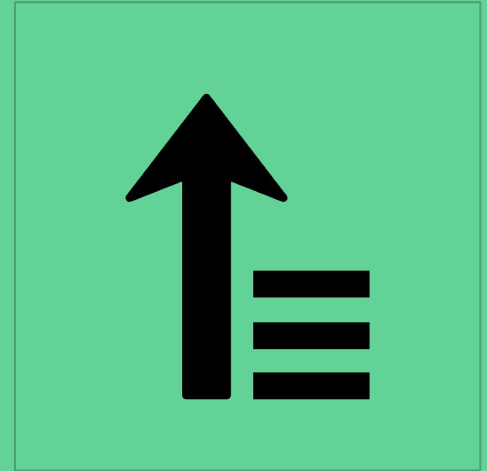
Esta é a última etapa de intercalação: são intercalados os dois lados do vetor.

Note que os subvetores [3, 4, 6, 12, 21] e [ 7, 9, 16, 18, 25] já estão ordenados. A próxima etapa de intercalação resultará no vetor ordenado.

3	4	6	7	9	12	16	18	21	25
---	---	---	---	---	----	----	----	----	----

# Merge Sort

## Iterativo





# Merge Sort Top-Down

A versão tradicional do merge sort utiliza uma abordagem inicialmente *top-down* (de cima para baixo) ao sair dividindo os vetores e depois volta intercalando, em passagens *bottom-up* (de baixo para cima). Por conta disso, o método é chamado de implementação ***top-down***.

É possível pular a etapa *top-down*, indo direto às etapas *bottom-up*, em uma versão iterativa do merge sort.

# Merge Sort Bottom-Up Iterativo

O raciocínio por trás de uma versão **bottom-up** iterativa do merge sort é considerar que o vetor já está inicialmente partido em pedaços de tamanho um e ir aumentando o tamanho dos pedaços.

Em cada etapa, o pedaço dobra de tamanho, até que os pedaços sejam de tamanho igual ou maior ao próprio vetor.

Essa versão do algoritmo é base para o algoritmos de ordenação externa (em disco).

# Merge Sort Iterativo - Algoritmo


```
void mergeiterativo (int v[], int tam) {  
    int p, r, b = 1;  
    while (b < tam) {  
        p = 0;  
        while (p + b < tam) {  
            r = p + 2*b - 1;  
            if (r >= tam) r = tam - 1;  
            intercala(v, p, p+b, r);  
            p = p + 2*b;  
        }  
        b = 2*b; // dobra o tamanho do bloco  
    }  
}
```



# Merge Sort Iterativo - Algoritmo

```
void mergeiterativo (int v[], int tam) {  
    int p, r, b = 1;  
    while (b < tam) {  
        p = 0;  
        while (p + b < tam) {  
            r = p + 2*b - 1;  
            if (r >= tam) r = tam - 1;  
            intercala(v, p, p+b, r);  
            p = p + 2*b;  
        }  
        b = 2*b; // dobra o tamanho do bloco  
    }  
}
```

b é o tamanho do bloco, inicialmente consideramos blocos de tamanho 1



# Merge Sort Iterativo - Algoritmo

```
void mergeiterativo (int v[], int tam) {  
    int p, r, b = 1;  
    while (b < tam) {  
        p = 0;  
        while (p + b < tam) {  
            r = p + 2*b - 1;  
            if (r >= tam) r = tam - 1;  
            intercala(v, p, p+b, r);  
            p = p + 2*b;  
        }  
        b = 2*b; // dobra o tamanho do bloco  
    }  
}
```

O bloco dobra de tamanho a cada iteração externa, que termina quando o tamanho do bloco é maior que o tamanho do vetor.

# Merge Sort Iterativo - Algoritmo

```
void mergeiterativo (int v[], int tam) {  
    int p, r, b = 1;  
    while (b < tam) {  
        p = 0;  
        while (p + b < tam) {  
            r = p + 2*b - 1;  
            if (r >= tam) r = tam - 1;  
            intercala(v, p, p+b, r);  
            p = p + 2*b;  
        }  
        b = 2*b; // dobra o tamanho do bloco  
    }  
}
```

p é a posição inicial a ser usada na intercalação. Ela pula dois blocos a cada iteração interna.

# Merge Sort Iterativo - Algoritmo

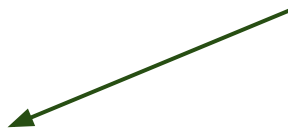
```
void mergeiterativo (int v[], int tam) {  
    int p, r, b = 1;  
    while (b < tam) {  
        p = 0;  
        while (p + b < tam) {  
            r = p + 2*b - 1;  
            if (r >= tam) r = tam - 1;  
            intercala(v, p, p+b, r);  
            p = p + 2*b;  
        }  
        b = 2*b; // dobra o tamanho do bloco  
    }  
}
```

r é a posição final a ser usada na intercalação. Ela pula dois blocos a partir da posição p.

# Merge Sort Iterativo - Algoritmo

```
void mergeiterativo (int v[], int tam) {  
    int p, r, b = 1;  
    while (b < tam) {  
        p = 0;  
        while (p + b < tam) {  
            r = p + 2*b - 1;  
            if (r >= tam) r = tam - 1;  
            intercala(v, p, p+b, r);  
            p = p + 2*b;  
        }  
        b = 2*b; // dobra o tamanho do bloco  
    }  
}
```

Caso o valor de  $r$  ultrapasse a posição final do vetor, então  $r$  é ajustado para indicar essa última posição.





# Merge Sort Iterativo - Algoritmo

```
void mergeiterativo (int v[], int tam) {  
    int p, r, b = 1;  
    while (b < tam) {  
        p = 0;  
        while (p + b < tam) {  
            r = p + 2*b - 1;  
            if (r >= tam) r = tam - 1;  
            intercala(v, p, p+b, r);  
            p = p + 2*b;  
        }  
        b = 2*b; // dobra o tamanho do bloco  
    }  
}
```

Tendo-se p e r, utiliza-se um bloco a partir de p como posição do meio na intercalação.

# Merge Sort Iterativo - Exemplo - i

4	12	6	21	18	3	9	16	25	7
---	----	---	----	----	---	---	----	----	---

tam = 10

p = 0      r = 1      b = 1

**intercala (v, 0, 1, 1)**

12	4	6	21	18	3	9	16	25	7
----	---	---	----	----	---	---	----	----	---

tam = 10

p = 2      r = 3      b = 1

**intercala (v, 2, 3, 3)**

4	12	6	21	18	3	9	16	25	7
---	----	---	----	----	---	---	----	----	---

# Merge Sort Iterativo - Exemplo - ii

tam = 10      p = 4      r = 5      b = 1

**intercala (v, 4, 5, 5)**

4	12	6	21	18	3	9	16	25	7
---	----	---	----	----	---	---	----	----	---

tam = 10      p = 6      r = 7      b = 1

**intercala (v, 6, 7, 7)**

4	12	6	21	3	18	9	16	25	7
---	----	---	----	---	----	---	----	----	---

# Merge Sort Iterativo - Exemplo - iii

tam = 10      p = 8      r = 9      b = 1

**intercala (v, 8, 9, 9)**

4	12	6	21	3	18	9	16	25	7
---	----	---	----	---	----	---	----	----	---

tam = 10      p = 10      r = 9      b = 1

4	12	6	21	3	18	9	16	7	25
---	----	---	----	---	----	---	----	---	----

**$p + b > \text{tam} \Rightarrow \text{dobra tamanho de } b$**

# Merge Sort Iterativo - Exemplo - iv

tam = 10      p = 0      r = 3      b = 2

**intercala (v, 0, 2, 3)**

4	12	6	21	3	18	9	16	7	25
---	----	---	----	---	----	---	----	---	----

tam = 10      p = 4      r = 7      b = 2

**intercala (v, 4, 6, 7)**

4	6	12	21	3	18	9	16	7	25
---	---	----	----	---	----	---	----	---	----

# Merge Sort Iterativo - Exemplo - v

tam = 10

p = 8

r = 9

b = 2

4	6	12	21	3	9	16	18	7	25
---	---	----	----	---	---	----	----	---	----

$p + b > \text{tam} \Rightarrow \text{dobra tamanho de } b$

tam = 10

p = 0

r = 7

b = 4

intercala (v, 0, 4, 7)

4	6	12	21	3	9	16	18	7	25
---	---	----	----	---	---	----	----	---	----

# Merge Sort Iterativo - Exemplo - vi

tam = 10

p = 8

r = 7

b = 4

3	4	6	9	12	16	18	21	7	25
---	---	---	---	----	----	----	----	---	----

**p + b > tam  $\Rightarrow$  dobra tamanho de b**

tam = 10

p = 0

r = 9

b = 8

**intercala (v, 0, 8, 9)**

3	4	6	9	12	16	18	21	7	25
---	---	---	---	----	----	----	----	---	----

# Merge Sort Iterativo - Exemplo - vii

tam = 10

p = 8

r = 9

b = 8

3	4	6	7	9	12	16	18	21	25
---	---	---	---	---	----	----	----	----	----

**$p + b > \text{tam} \Rightarrow \text{dobra tamanho de } b$**

tam = 10

p = 0

r = 0

b = 16

**$b > \text{tam} \Rightarrow \text{procedimento encerrado!}$**



# Análise do Merge Sort

Merge sort e heap sort possuem geralmente o mesmo nível de eficiência, possuindo ambos melhores tempos nos piores casos do quick sort. Entretanto, em memória, o quick sort é preferido, por conta do caso médio bem melhor que esses dois métodos.

Por outro lado, o merge sort é mais eficiente e adequado para uso em memória secundária (discos, etc.), sendo também o método preferido em listas encadeadas.

# Sobre o Material



# Sobre este material

Material produzido coletivamente, principalmente pelos seguintes professores do DCC/UFLA:

- Janderson Rodrigo de Oliveira
- Joaquim Quinteiro Uchôa
- Juliana Galvani Greggi
- Renato Ramos da Silva

Inclui contribuições de outros professores do setor de Fundamentos de Programação do DCC/UFLA.

Esta obra está licenciado com uma Licença [Creative Commons Atribuição 4.0 Internacional](https://creativecommons.org/licenses/by/4.0/).