Algoritmos e Estruturas de Dados II

2º Período Engenharia da Computação

Prof. Edwaldo Soares Rodrigues
Email: edwaldo.rodrigues@uemg.br

UNIVERSIDADE
DO ESTADO DE MINAS GERAIS | UEMG
UNIDADE DIVINÓPOLIS

Métodos de Ordenação - MergeSort

MergeSort

- O MergeSort (ordenação por intercalação) é um método que utiliza uma técnica de dividir para conquistar

- É um algoritmo onde a cada passo:

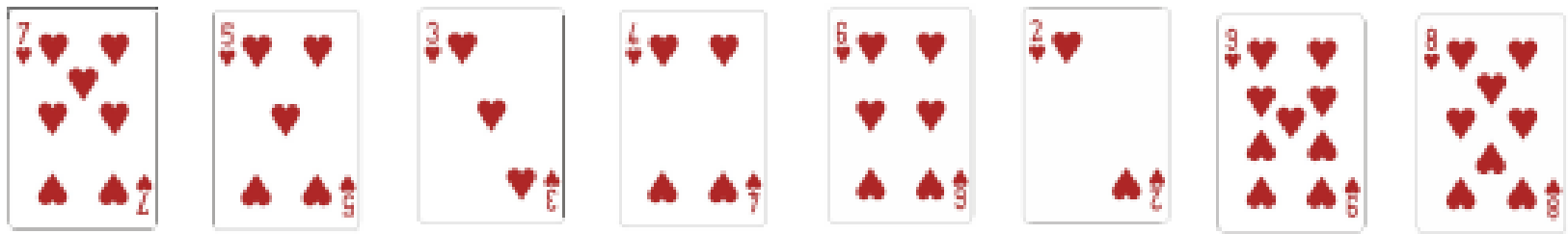
  - Divide-se o arranjo em 2 arranjos menores até que tenhamos vários arranjos de tamanho 1

- Então, iniciamos uma repetição onde:

  - 2 arranjos são fundidos em um arranjo maior ordenado até que tenhamos o arranjo original ordenado

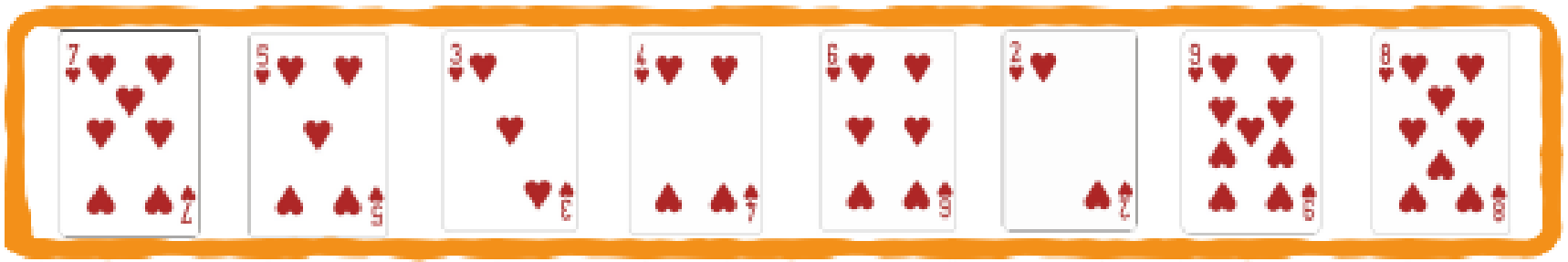
MergeSort

- Considere um arranjo com os seguintes elementos



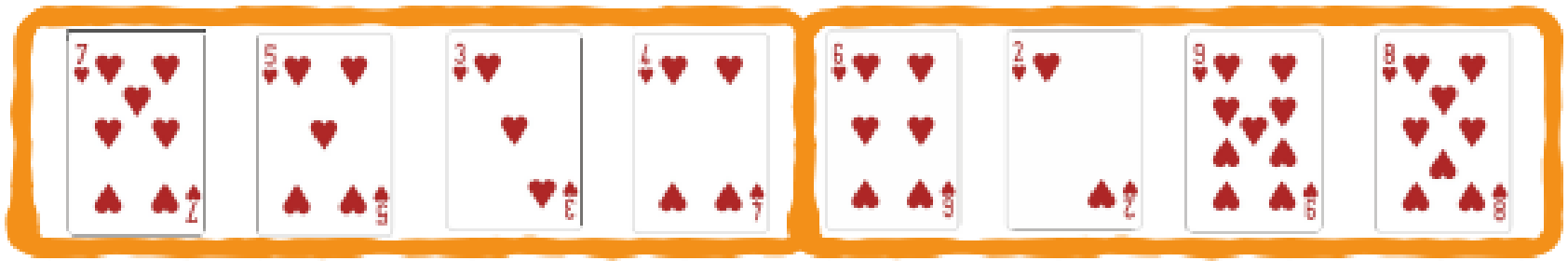
MergeSort

- Já inicialmente, temos um arranjo de 8 elementos



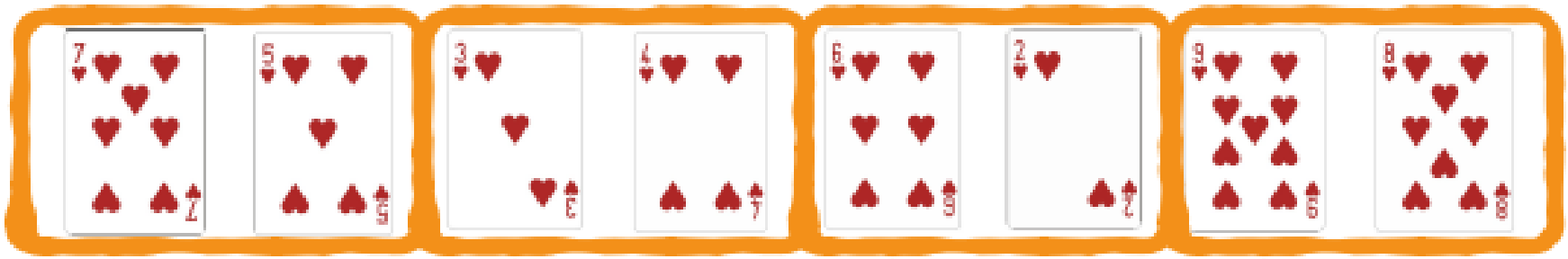
MergeSort

- Dividimos este arranjo em dois arranjos com a metade do tamanho



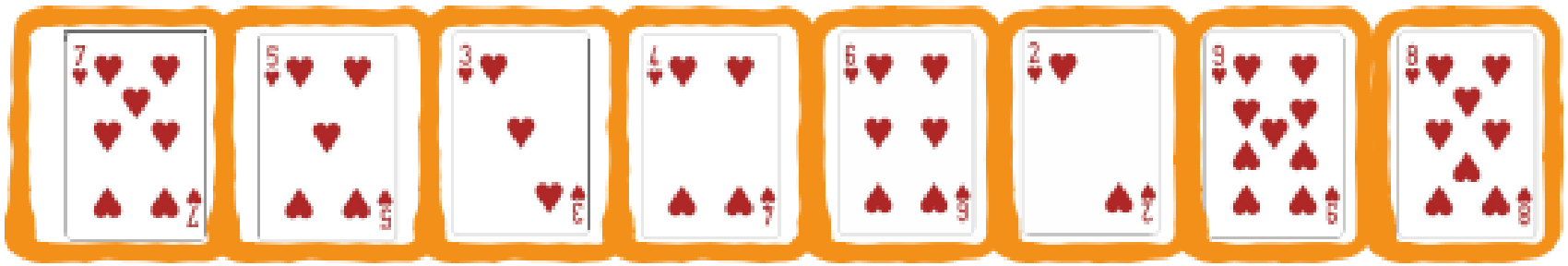
MergeSort

- Redividimos cada arranjo na metade



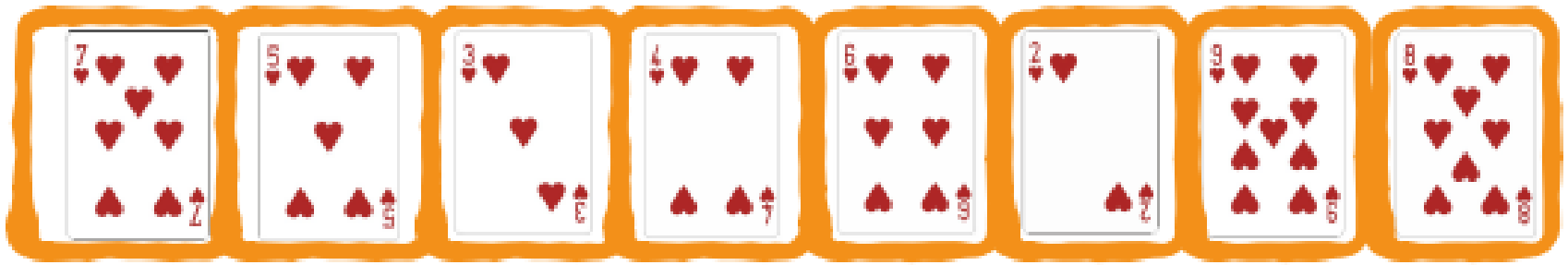
MergeSort

- Este processo continua até que tenhamos arranjos de 1 elemento cada



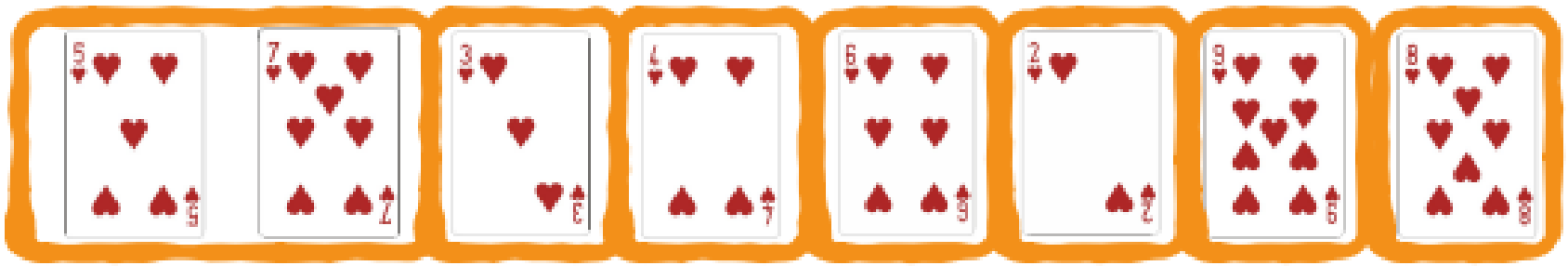
MergeSort

- Sabemos que cada um destes arranjos de 1 elemento é um arranjo ordenado. Iniciaremos agora a fase da intercalação



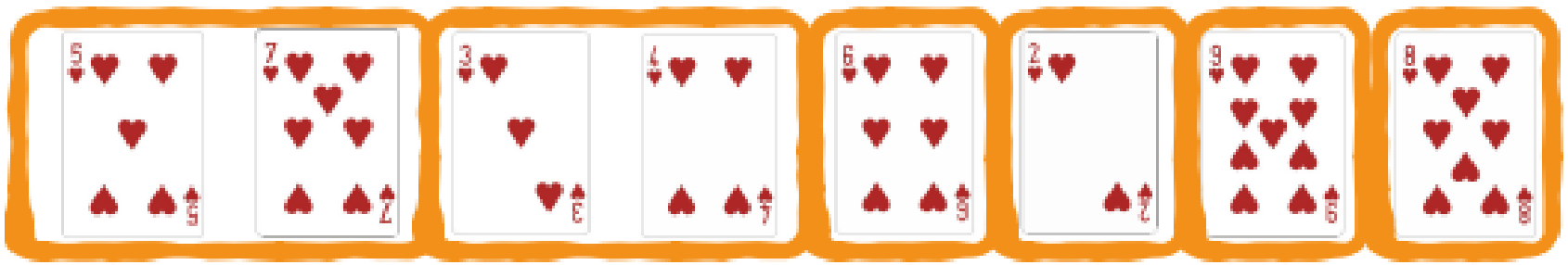
MergeSort

- Os dois primeiros arranjos ordenados são intercalados em um novo arranjo ordenado



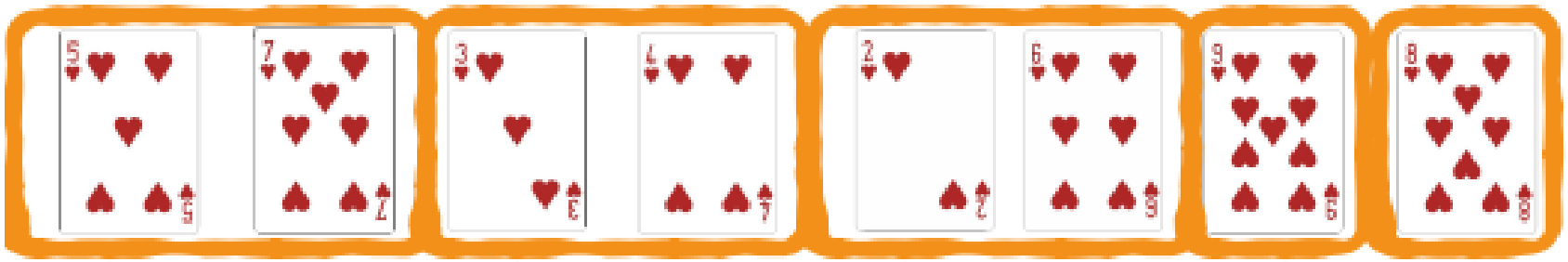
MergeSort

- Fazemos isto com todos os arranjos menores



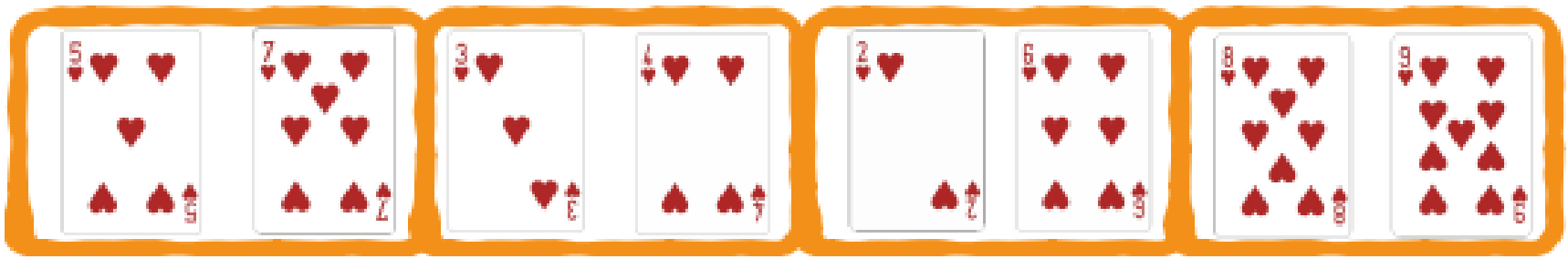
MergeSort

- Fazemos isto com todos os arranjos menores



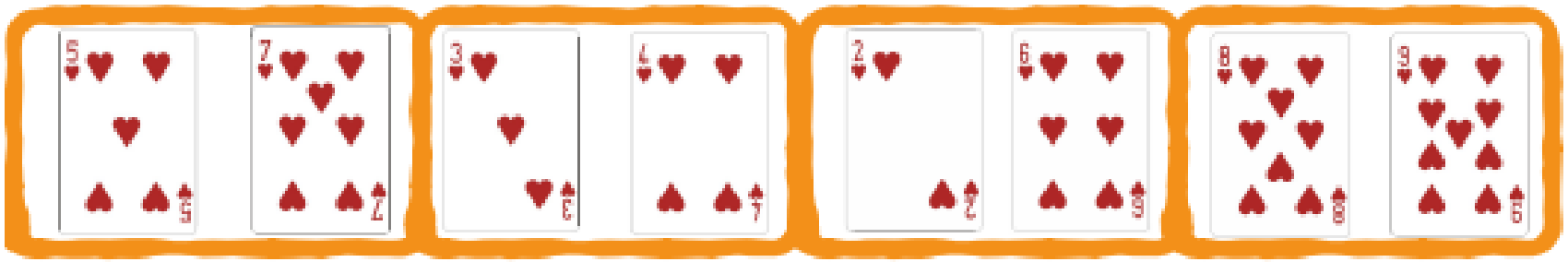
MergeSort

- Fazemos isto com todos os arranjos menores



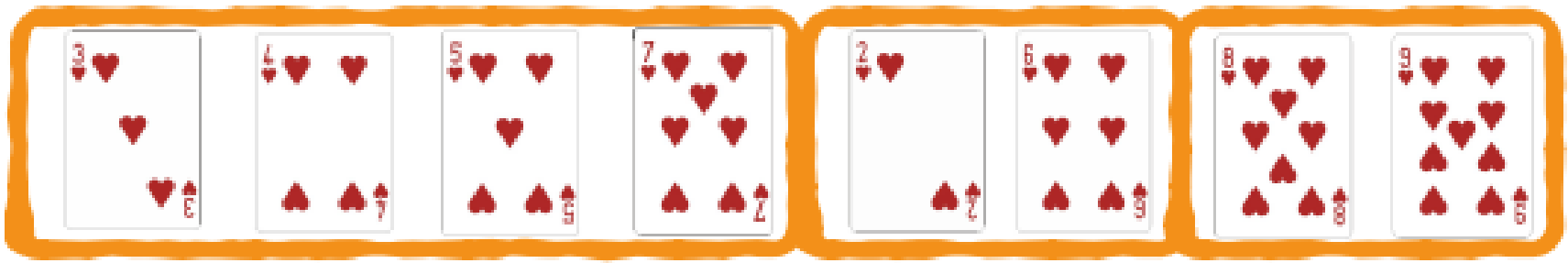
MergeSort

- O interessante é que o custo de se intercalar dois arranjos ordenados em um arranjo ordenado é menor que o custo de se colocar dois arranjos desordenados em um arranjo ordenado



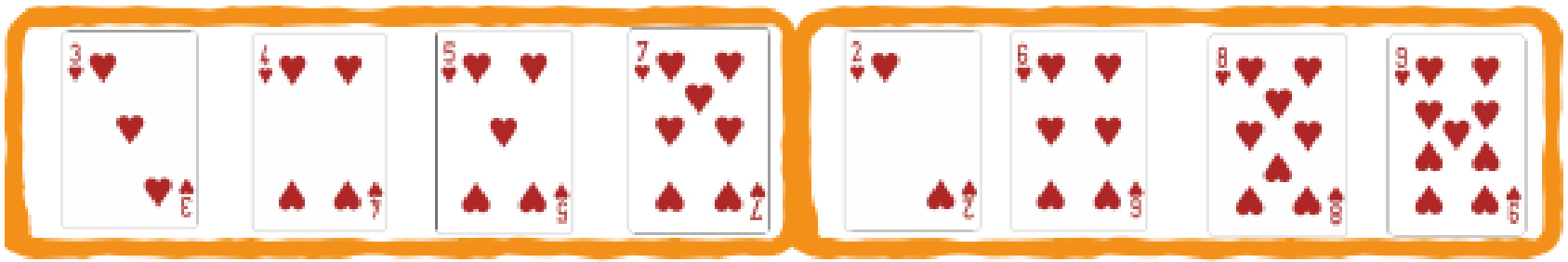
MergeSort

- Fazemos a intercalação para os arranjos de tamanho 2



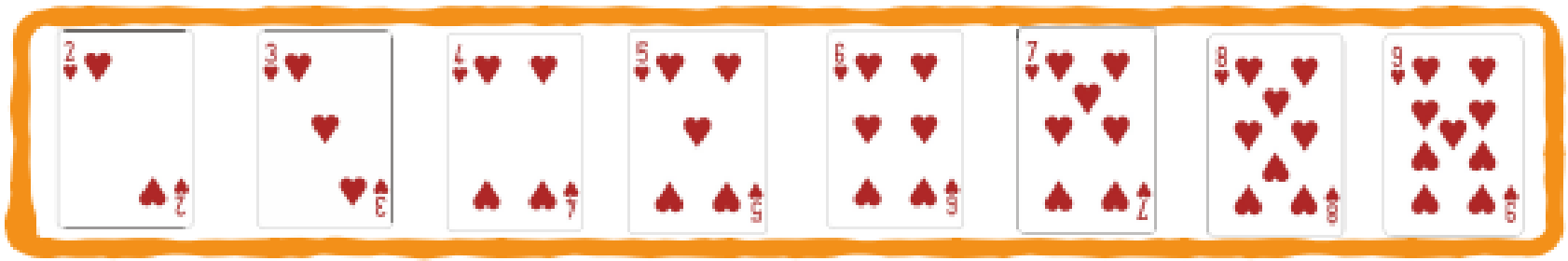
MergeSort

- Fazemos a intercalação para os arranjos de tamanho 2

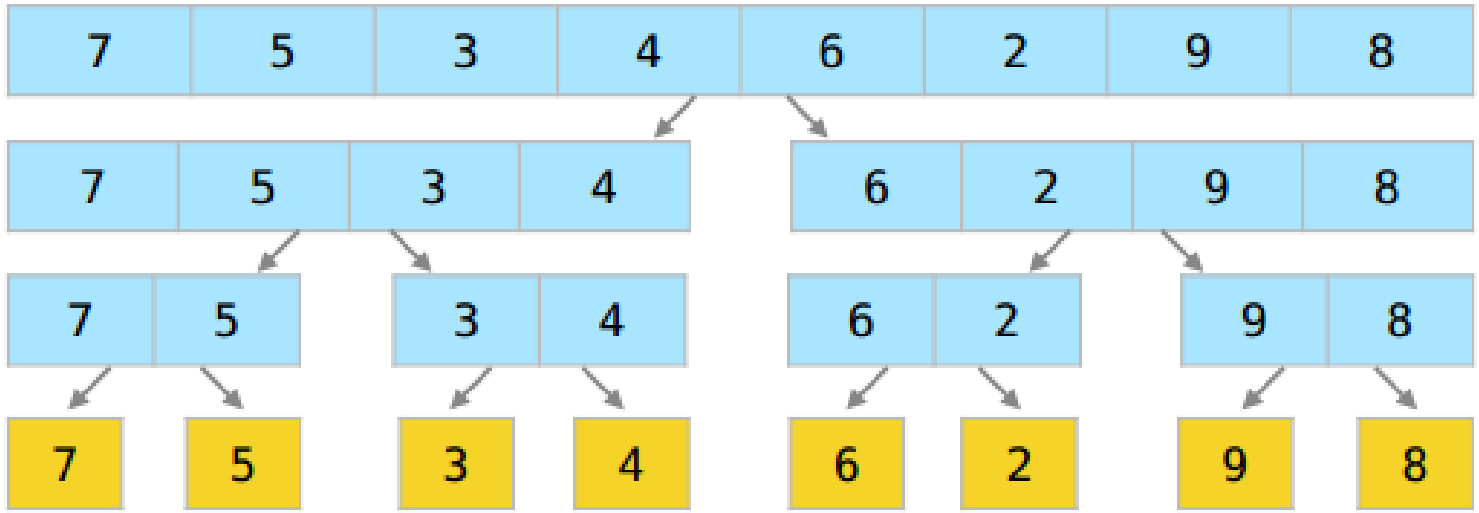


MergeSort

- Na última intercalação, temos todo o arranjo original ordenado



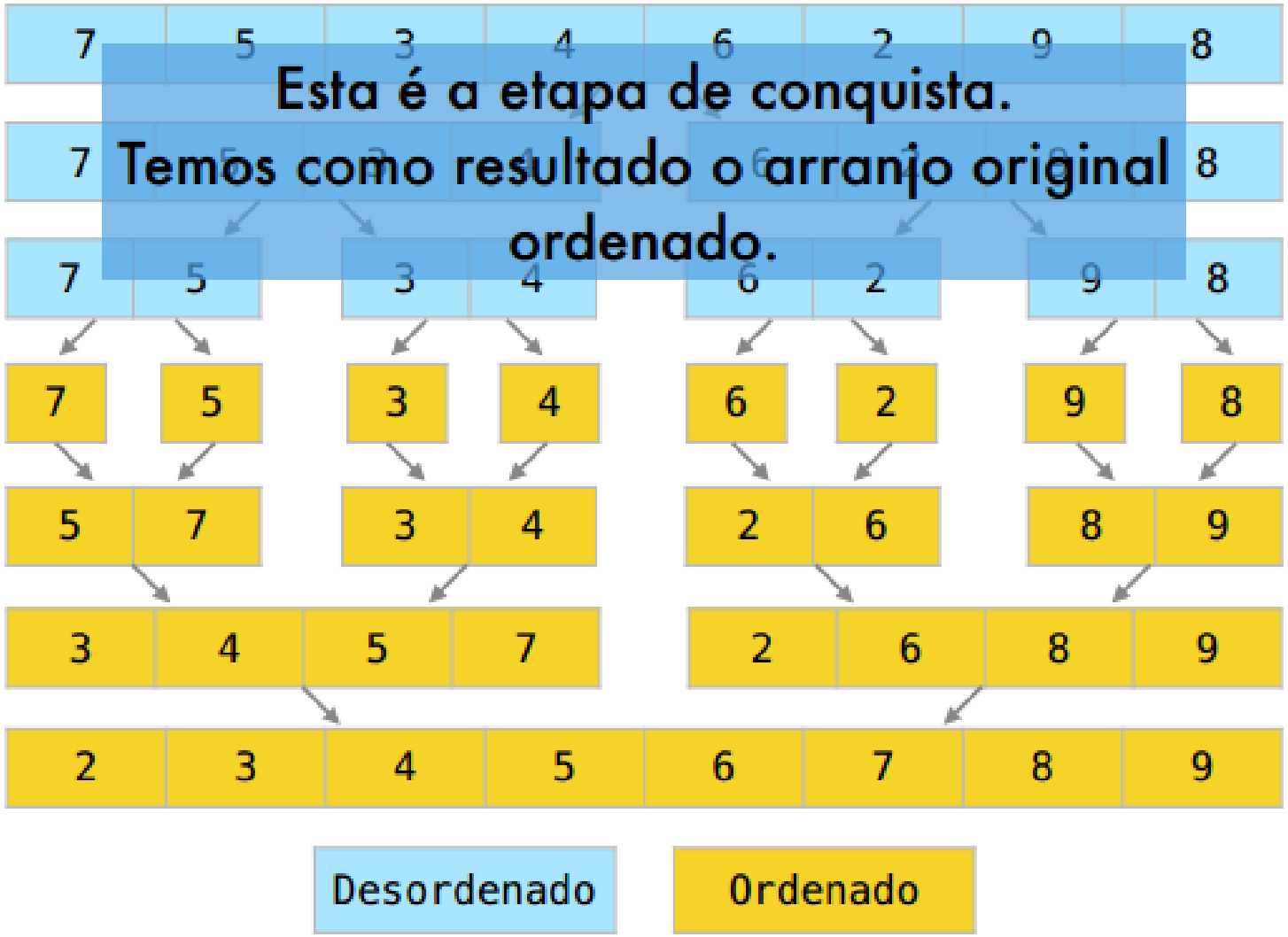
MergeSort



Esta é a etapa de divisão.
Temos como resultado *n* arranjos ordenados de tamanho 1.

Desordenado Ordenado

MergeSort

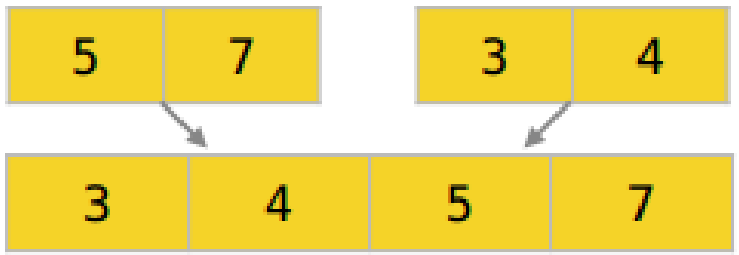

Esta é a etapa de conquista.
Temos como resultado o arranjo original ordenado.
Desordenado
Ordenado

MergeSort


Divisão
Intercalação
7 5 3 4 6 2 9 8
7 5 3 4 6 2 9 8
7 5 3 4 6 2 9 8
7 5 3 4 6 2 9 8
5 7 3 4 2 6 8 9
3 4 5 7 2 6 8 9
2 3 4 5 6 7 8 9
Desordenado
Ordenado

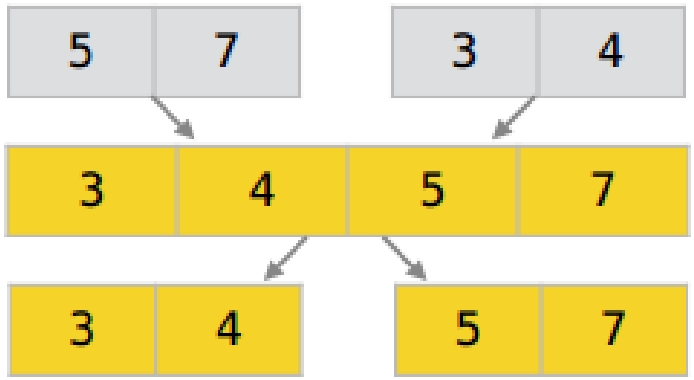
MergeSort – Etapa de intercalação

- Para a implementação de maneira mais usual, a etapa de intercalação de dois arranjos requer um arranjo auxiliar, onde serão colocados os itens



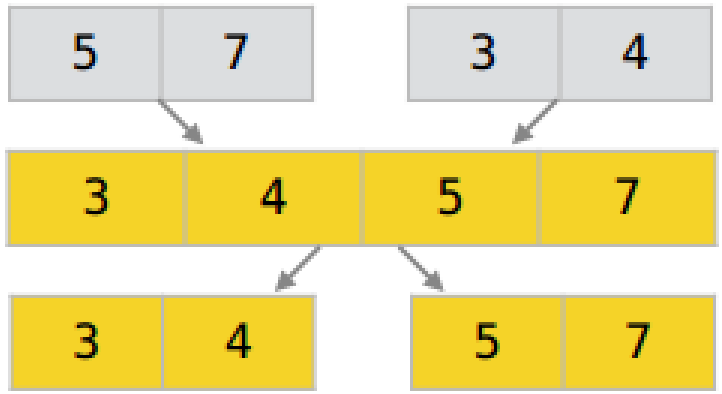
MergeSort – Etapa de intercalação

- Os elementos são intercalados em um arranjo auxiliar e então são transferidos de volta para o arranjo de onde vieram: o arranjo sendo ordenado



MergeSort – Etapa de intercalação

- Assim, para intercalar n elementos, precisamos de uma memória extra O(n)



MergeSort – Etapa de intercalação

- O algoritmo de intercalação é organizado em algumas etapas

```
void intercala(int a[], int n)
{
    int *tmp = new int[n]; // Arranjo temporário para intercalação
    int meio = n / 2; // Índice que marca o meio do arranjo
    int i, j, k; // Índices para a estrutura de repetição
    i = 0; // Índice i marca itens intercalados do primeiro arranjo
    j = meio; // Índice j marca itens intercalados do segundo arranjo
    k = 0; // Índice k marca itens já intercalados no arranjo temporário
    // Enquanto os índices i e j não tenham chegado ao fim de seus arranjos
    while (i < meio && j < n){
        // colocamos o menor item entre a[i] e a[j] no arranjo temporário
        if (a[i] < a[j]){
            tmp[k] = a[i];
            ++i;
        } else {
            tmp[k] = a[j];
            ++j;
        }
        ++k;
    }
    // se o índice i chegou ao fim de seu arranjo primeiro
    if (i == meio) {
        // os outros elementos do segundo arranjo vão para o arranjo temporário
        while (j < n) {
            tmp[k] = a[j];
            ++j;
            ++k;
        }
    // se foi o índice j que chegou ao fim de seu arranjo primeiro
    } else {
```

MergeSort – Etapa de intercalação

Primeiramente, um arranjo temporário é criado e os índices são inicializados

```
void intercala(int a[], int n)
{
    int *tmp = new int[n]; // Arranjo temporário para intercalação
    int meio = n / 2; // Índice que marca o meio do arranjo
    int i, j, k; // Índices para a estrutura de repetição
    i = 0; // Índice i marca itens intercalados do primeiro arranjo
    j = meio; // Índice j marca itens intercalados do segundo arranjo
    k = 0; // Índice k marca itens já intercalados no arranjo temporário
    // Enquanto os índices i e j não tenham chegado ao fim de seus arranjos
    while (i < meio && j < n){
        // colocamos o menor item entre a[i] e a[j] no arranjo temporário
        if (a[i] < a[j]){
            tmp[k] = a[i];
            ++i;
        } else {
            tmp[k] = a[j];
            ++j;
        }
        ++k;
    }
    // se o índice i chegou ao fim de seu arranjo primeiro
    if (i == meio) {
        // os outros elementos do segundo arranjo vão para o arranjo temporário
        while (j < n) {
            tmp[k] = a[j];
            ++j;
            ++k;
        }
    // se foi o índice j que chegou ao fim de seu arranjo primeiro
    } else {
```

MergeSort – Etapa de intercalação

Logo após, os elementos de a são intercalados neste arranjo temporário...

```
k = 0; // índice k marca itens já intercalados no arranjo temporário
// Enquanto os índices i e j não tenham chegado ao fim de seus arranjos
while (i < meio && j < n){
    // colocamos o menor item entre a[i] e a[j] no arranjo temporário
    if (a[i] < a[j]){
        tmp[k] = a[i];
        ++i;
    } else {
        tmp[k] = a[j];
        ++j;
    }
    ++k;
}
// se o índice i chegou ao fim de seu arranjo primeiro
if (i == meio) {
    // os outros elementos do segundo arranjo vão para o arranjo temporário
    while (j < n) {
        tmp[k] = a[j];
        ++j;
        ++k;
    }
// se foi o índice j que chegou ao fim de seu arranjo primeiro
} else {
    // os outros elementos do primeiro arranjo vão para o arranjo temporário
    while (i < meio) {
        tmp[k] = a[i];
        ++i;
        ++k;
    }
}
// neste ponto, o arranjo temporário tem todos os elementos intercalados
// estes elementos são copiados de volta para o arranjo int a[]
```

MergeSort – Etapa de intercalação

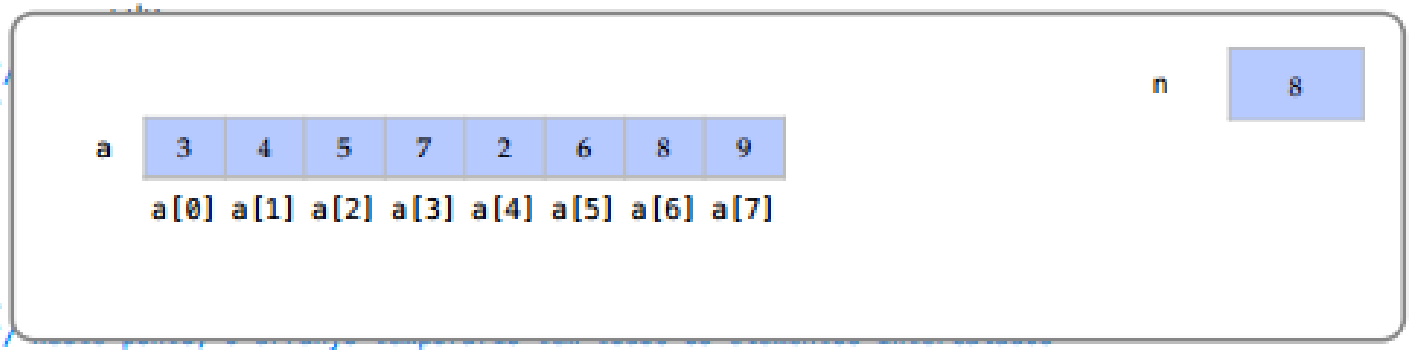
Então, os elementos intercalados no arranjo temporário são transferidos de volta para o arranjo a

```
            // colocamos o menor item entre a[i] e a[j] no arranjo temporário
            if (a[i] < a[j]){
                ++k;
            }
            // se o índice i chegou ao fim de seu arranjo primeiro
            if (i == meio) {
                // os outros elementos do segundo arranjo vão para o arranjo temporário
                while (j < n) {
                    tmp[k] = a[j];
                    ++j;
                    ++k;
                }
            // se foi o índice j que chegou ao fim de seu arranjo primeiro
            } else {
                // os outros elementos do primeiro arranjo vão para o arranjo temporário
                while (i < meio) {
                    tmp[k] = a[i];
                    ++i;
                    ++k;
                }
            }
            // neste ponto, o arranjo temporário tem todos os elementos intercalados
            // estes elementos são copiados de volta para o arranjo int a[]
            for (i = 0; i < n; ++i)
            {
                a[i] = tmp[i];
            }
            // o arranjo temporário pode então ser desalocado da memória
            delete [] tmp;
        }
```

MergeSort – Etapa de intercalação

```
void intercala(int a[], int n)
{
    int *tmp = new int[n]; // Arranjo temporário para intercalação
    int meio = n / 2; // Índice que marca o meio do arranjo
    int i, j, k; // Índices para a estrutura de repetição
    i = 0; // Índice i marca itens intercalados do primeiro arranjo
    j = meio; // Índice j marca itens intercalados do segundo arranjo
    k = 0; // Índice k marca itens já intercalados no arranjo temporário
    // Enquanto os índices i e j não tenham chegado ao fim de seus arranjos
    while (i < meio && j < n){
        // colocamos o menor item entre a[i] e a[j] no arranjo temporário
        if (a[i] < a[j]){
            tmp[k] = a[i];
            ++i;
        } else {
            tmp[k] = a[j];
            ++j;
        }
        ++k;
    }
    // se o índice i chegou ao fim de seu arranjo primeiro
    if (i == meio) {
        // os outros elementos do segundo arranjo vão para o arranjo temporário
        while (j < n) {
            tmp[k] = a[j];
            ++j;
            ++k;
        }
    }
}
```

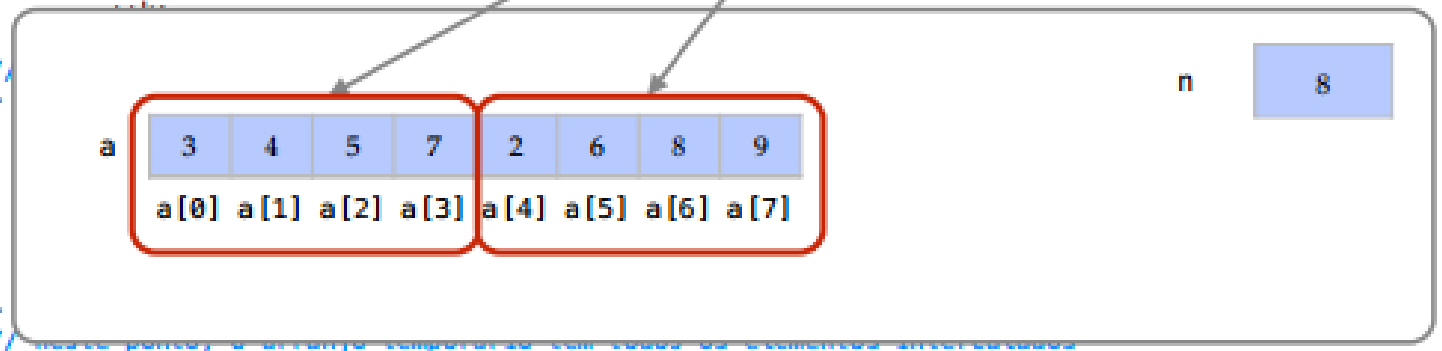
A função intercalará os elementos de a[]



MergeSort – Etapa de intercalação

```
void intercala(int a[], int n)
{
    int *tmp = new int[n]; // Arranjo temporário para intercalação
    int meio = n / 2; // Índice que marca o meio do arranjo
    int i, j, k; // Índices para a estrutura de repetição
    i = 0; // Índice i marca itens intercalados do primeiro arranjo
    j = meio; // Índice j marca itens intercalados do segundo arranjo
    k = 0; // Índice k marca itens já intercalados no arranjo temporário
    // Enquanto os índices i e j não tenham chegado ao fim de seus arranjos
    while (i < meio && j < n){
        // colocamos o menor item entre a[i] e a[j] no arranjo temporário
        if (a[i] < a[j]){
            tmp[k] = a[i];
            ++i;
        } else {
            tmp[k] = a[j];
            ++j;
        }
        ++k;
    }
    // se o índice i chegou ao fim de seu arranjo primeiro
    if (i == meio) {
        // os outros elementos do segundo arranjo vão para o arranjo temporário
        while (j < n) {
            tmp[k] = a[j];
            ++j;
            ++k;
        }
    }
```

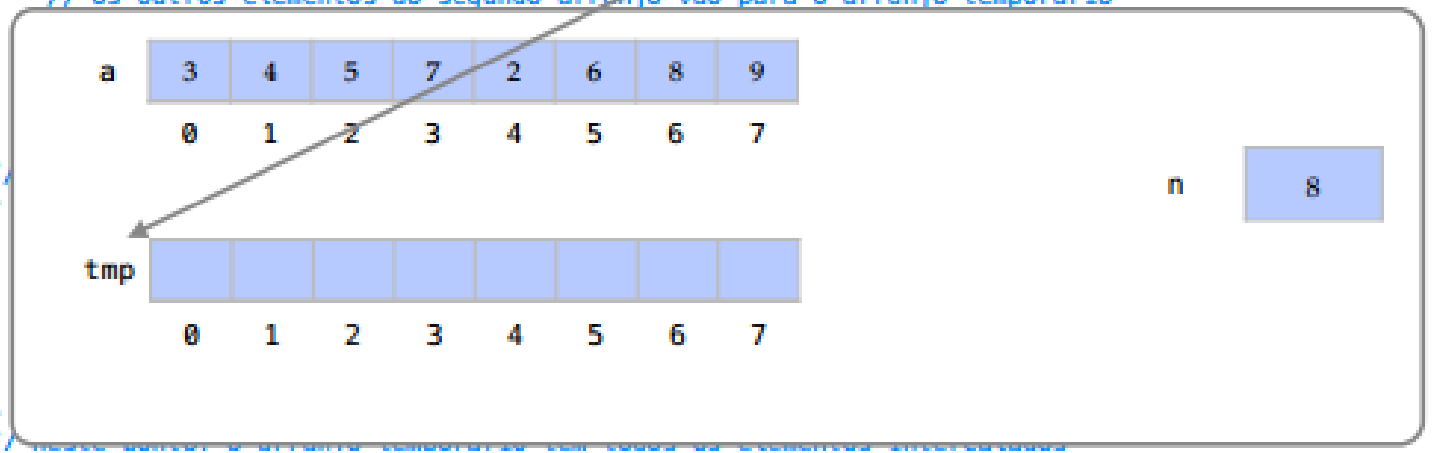
Os elementos de a[0] até a[3] serão intercalados com os elementos de a[4] a a[7]



MergeSort – Etapa de intercalação

```
void intercala(int a[], int n)
{
    int *tmp = new int[n]; // Arranjo temporário para intercalação
    int meio = n / 2; // Índice que marca o meio do arranjo
    int i, j, k; // Índices para a estrutura de repetição
    i = 0; // Índice i marca itens intercalados do primeiro arranjo
    j = meio; // Índice j marca itens intercalados do segundo arranjo
    k = 0; // Índice k marca itens já intercalados no arranjo temporário
    // Enquanto os índices i e j não tenham chegado ao fim de seus arranjos
    while (i < meio && j < n){
        // colocamos o menor item entre a[i] e a[j] no arranjo temporário
        if (a[i] < a[j]){
            tmp[k] = a[i];
            ++i;
        } else {
            tmp[k] = a[j];
            ++j;
        }
        ++k;
    }
    // se o índice i chegou ao fim de seu arranjo primeiro
    if (i == meio) {
        // os outros elementos do segundo arranjo vão para o arranjo temporário
//
    }
    }
```

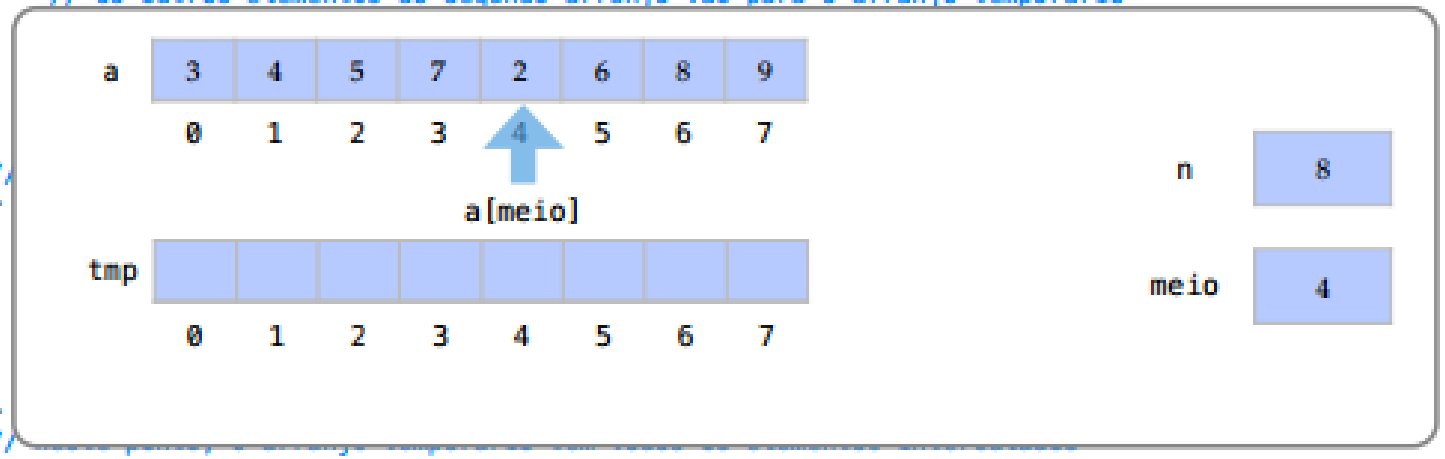
Um arranjo temporário é criado para guardar os elementos intercalados



MergeSort – Etapa de intercalação

```
void intercala(int a[], int n)
{
    int *tmp = new int[n]; // Arranjo temporário para intercalação
    int meio = n / 2; // Índice que marca o meio do arranjo
    int i, j, k; // Índices para a estrutura de repetição
    i = 0; // Índice i marca itens intercalados do primeiro arranjo
    j = meio; // Índice j marca itens intercalados do segundo arranjo
    k = 0; // Índice k marca itens já intercalados no arranjo temporário
    // Enquanto os índices i e j não tenham chegado ao fim de seus arranjos
    while (i < meio && j < n){
        // colocamos o menor item entre a[i] e a[j] no arranjo temporário
        if (a[i] < a[j]){
            tmp[k] = a[i];
            ++i;
        } else {
            tmp[k] = a[j];
            ++j;
        }
        ++k;
    }
    // se o índice i chegou ao fim de seu arranjo primeiro
    if (i == meio) {
        // os outros elementos do segundo arranjo vão para o arranjo temporário
    //
    }
    }
    //
```

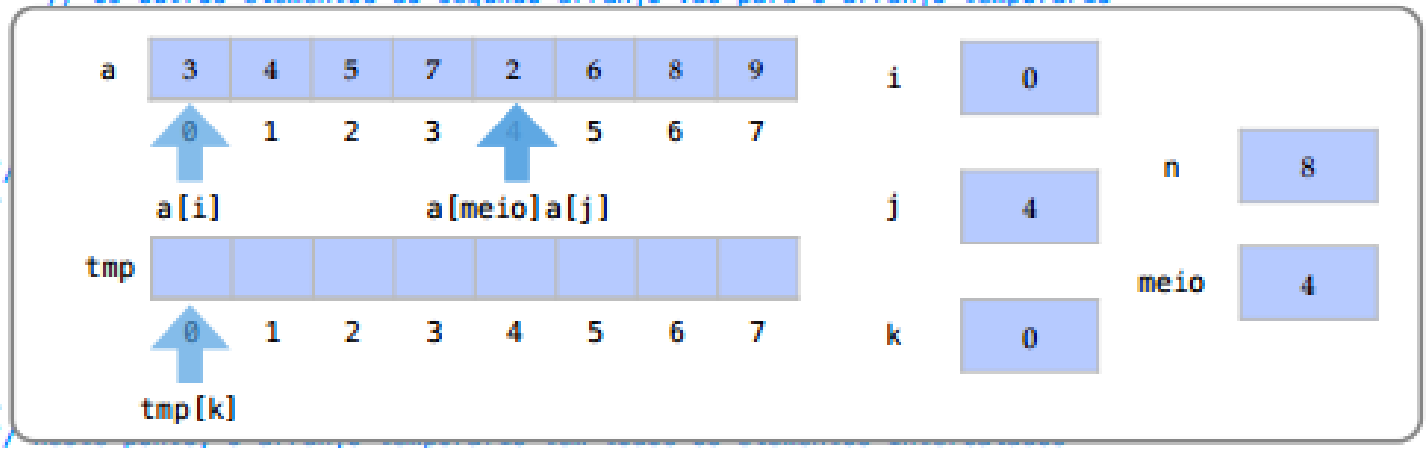
Um índice `meio` marca onde está a metade do arranjo. Ou onde começa o segundo arranjo a ser intercalado.



MergeSort – Etapa de intercalação

```
void intercala(int a[], int n)
{
    int *tmp = new int[n]; // Arranjo temporário para intercalação
    int meio = n / 2; // Índice que marca o meio do arranjo
    int i, j, k; // Índices para a estrutura de repetição
    i = 0; // Índice i marca itens intercalados do primeiro arranjo
    j = meio; // Índice j marca itens intercalados do segundo arranjo
    k = 0; // Índice k marca itens já intercalados no arranjo temporário
    // Enquanto os índices i e j não tenham chegado ao fim de seus arranjos
    while (i < meio && j < n){
        // colocamos o menor item entre a[i] e a[j] no arranjo temporário
        if (a[i] < a[j]){
            tmp[k] = a[i];
            ++i;
        } else {
            tmp[k] = a[j];
            ++j;
        }
        ++k;
    }
    // se o índice i chegou ao fim de seu arranjo primeiro
    if (i == meio) {
        // os outros elementos do segundo arranjo vão para o arranjo temporário
```

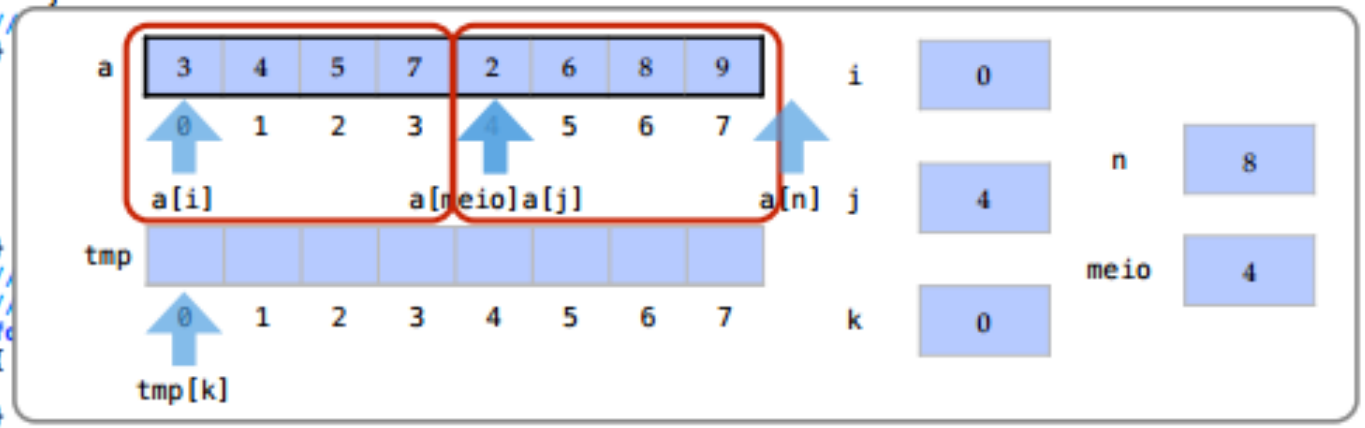
Índices i, j e k marcam onde intercalaremos no primeiro, segundo arranjo e no arranjo temporário



MergeSort – Etapa de intercalação

```
int meio = n / 2; // Índice que marca o meio do arranjo
int i, j, k; // Índices para a estrutura de repetição
i = 0; // Índice i marca itens intercalados do primeiro arranjo
j = meio; // Índice j marca itens intercalados do segundo arranjo
k = 0; // Índice k marca itens já intercalados no arranjo temporário
// Enquanto os índices i e j não tenham chegado ao fim de seus arranjos
while (i < meio && j < n){
    // colocamos o menor item entre a[i] e a[j] no arranjo temporário
    if (a[i] < a[j]){
        tmp[k] = a[i];
        ++i;
    } else {
        tmp[k] = a[j];
        ++j;
    }
    ++k;
}
// se o índice i chegou ao fim de seu arranjo primeiro
if (i == meio) {
    // os outros elementos do segundo arranjo vão para o arranjo temporário
    while (j < n) {
        tmp[k] = a[j];
        ++j;
        ++k;
    }
```

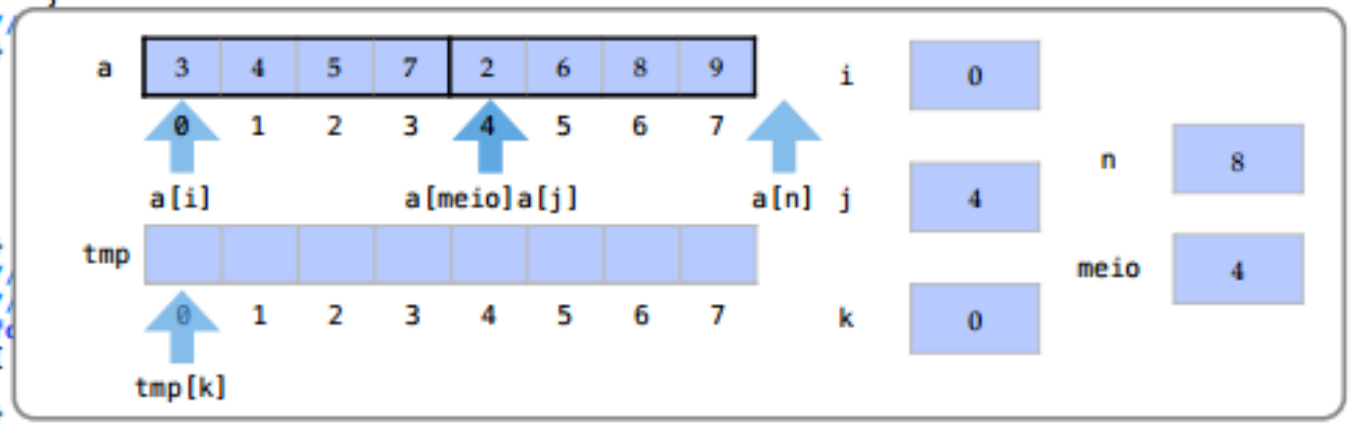
Se i é menor que meio e j é menor que n, não atingimos o fim de nenhum arranjo sendo intercalados



MergeSort – Etapa de intercalação

```
int meio = n / 2; // Índice que marca o meio do arranjo
int i, j, k; // Índices para a estrutura de repetição
i = 0; // Índice i marca itens intercalados do primeiro arranjo
j = meio; // Índice j marca itens intercalados do segundo arranjo
k = 0; // Índice k marca itens já intercalados no arranjo temporário
// Enquanto os índices i e j não tenham chegado ao fim de seus arranjos
while (i < meio && j < n){
    // colocamos o menor item entre a[i] e a[j] no arranjo temporário
    if (a[i] < a[j]){
        tmp[k] = a[i];
        ++i;
    } else {
        tmp[k] = a[j];
        ++j;
    }
    ++k;
}
// se o índice i chegou ao fim de seu arranjo primeiro
if (i == meio) {
    // os outros elementos do segundo arranjo vão para o arranjo temporário
    while (j < n) {
        tmp[k] = a[j];
        ++j;
        ++k;
    }
}
```

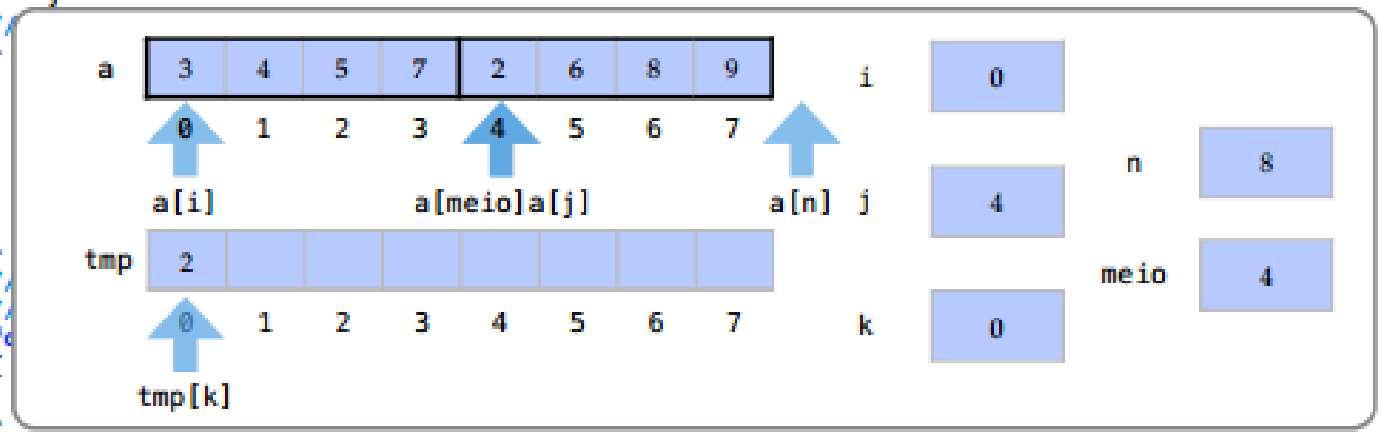
Como não atingimos o final de um dos arranjos, comparamos a[i] com a[j]



MergeSort – Etapa de intercalação

```
int meio = n / 2; // Indice que marca o meio do arranjo
int i, j, k; // Índices para a estrutura de repetição
i = 0; // Índice i marca itens intercalados do primeiro arranjo
j = meio; // Índice j marca itens intercalados do segundo arranjo
k = 0; // Índice k marca itens já intercalados no arranjo temporário
// Enquanto os índices i e j não tenham chegado ao fim de seus arranjos
while (i < meio && j < n){
    // colocamos o menor item entre a[i] e a[j] no arranjo temporário
    if (a[i] < a[j]){
        tmp[k] = a[i];
        ++i;
    } else {
        tmp[k] = a[j];
        ++j;
    }
    ++k;
}
// se o índice i chegou ao fim de seu arranjo primeiro
if (i == meio) {
    // os outros elementos do segundo arranjo vão para o arranjo temporário
    while (j < n) {
        tmp[k] = a[j];
        ++j;
        ++k;
    }
```

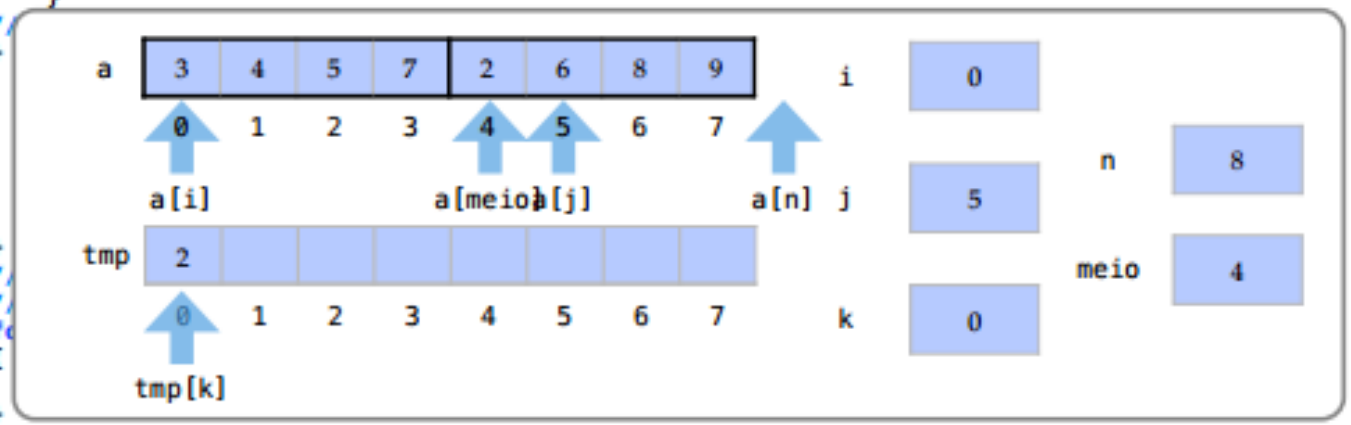
O menor é colocado no arranjo temporário...



MergeSort – Etapa de intercalação

```
int meio = n / 2; // Indice que marca o meio do arranjo
int i, j, k; // Índices para a estrutura de repetição
i = 0; // Índice i marca itens intercalados do primeiro arranjo
j = meio; // Índice j marca itens intercalados do segundo arranjo
k = 0; // Índice k marca itens já intercalados no arranjo temporário
// Enquanto os índices i e j não tenham chegado ao fim de seus arranjos
while (i < meio && j < n){
    // colocamos o menor item entre a[i] e a[j] no arranjo temporário
    if (a[i] < a[j]){
        tmp[k] = a[i];
        ++i;
    } else {
        tmp[k] = a[j];
        ++j;
    }
    ++k;
}
// se o índice i chegou ao fim de seu arranjo primeiro
if (i == meio) {
    // os outros elementos do segundo arranjo vão para o arranjo temporário
    while (j < n) {
        tmp[k] = a[j];
        ++j;
        ++k;
    }
```

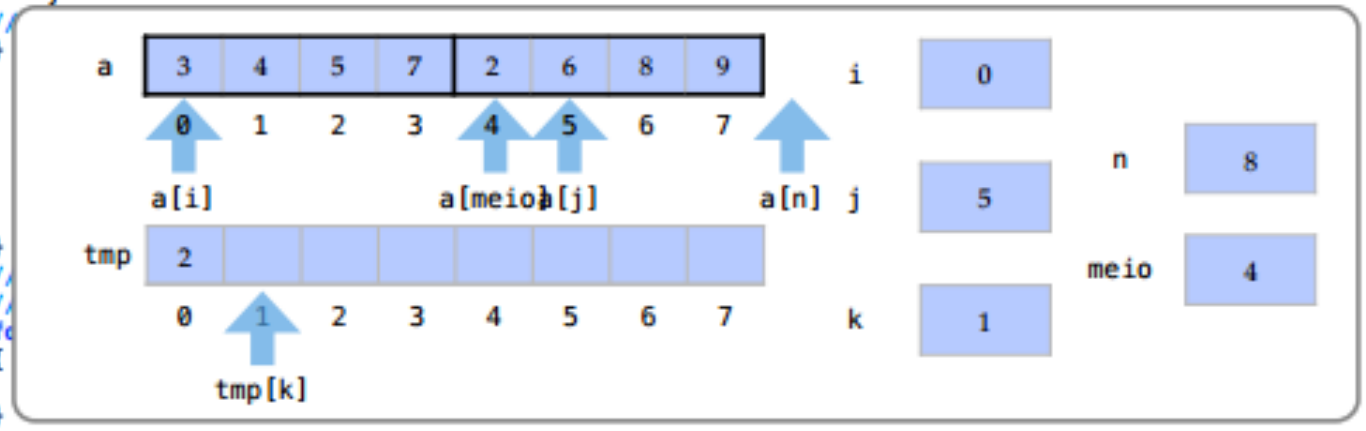
Com o elemento copiado, incrementamos o índice j, sendo usado no segundo arranjo



MergeSort – Etapa de intercalação

```
int meio = n / 2; // Índice que marca o meio do arranjo
int i, j, k; // Índices para a estrutura de repetição
i = 0; // Índice i marca itens intercalados do primeiro arranjo
j = meio; // Índice j marca itens intercalados do segundo arranjo
k = 0; // Índice k marca itens já intercalados no arranjo temporário
// Enquanto os índices i e j não tenham chegado ao fim de seus arranjos
while (i < meio && j < n){
    // colocamos o menor item entre a[i] e a[j] no arranjo temporário
    if (a[i] < a[j]){
        tmp[k] = a[i];
        ++i;
    } else {
        tmp[k] = a[j];
        ++j;
    }
    ++k;
}
// se o índice i chegou ao fim de seu arranjo primeiro
if (i == meio) {
    // os outros elementos do segundo arranjo vão para o arranjo temporário
    while (j < n) {
        tmp[k] = a[j];
        ++j;
        ++k;
    }
```

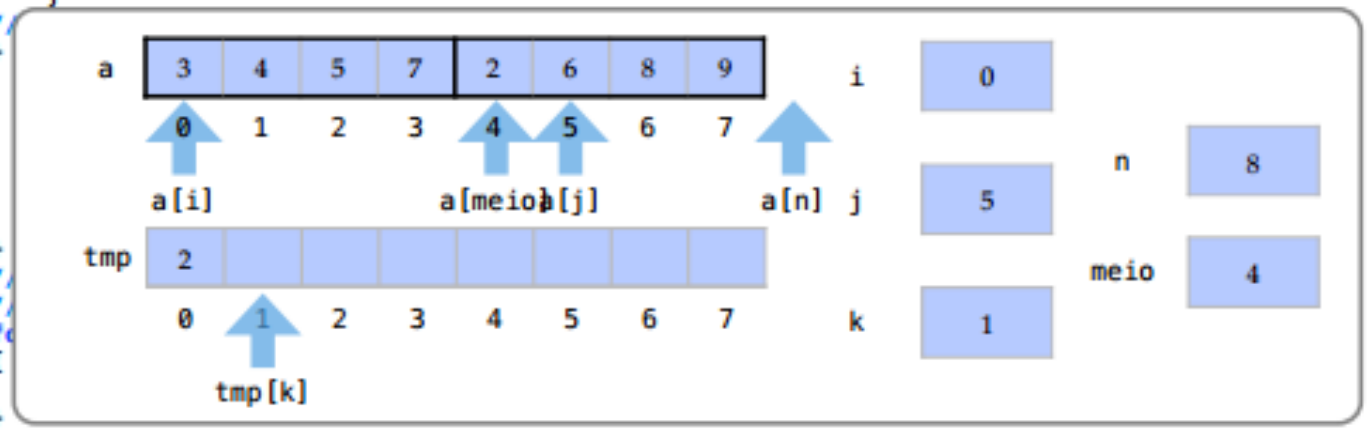
Sempre incrementamos também o índice k, usado no arranjo temporário



MergeSort – Etapa de intercalação

```
int meio = n / 2; // Índice que marca o meio do arranjo
int i, j, k; // Índices para a estrutura de repetição
i = 0; // Índice i marca itens intercalados do primeiro arranjo
j = meio; // Índice j marca itens intercalados do segundo arranjo
k = 0; // Índice k marca itens já intercalados no arranjo temporário
// Enquanto os índices i e j não tenham chegado ao fim de seus arranjos
while (i < meio && j < n){
    // colocamos o menor item entre a[i] e a[j] no arranjo temporário
    if (a[i] < a[j]){
        tmp[k] = a[i];
        ++i;
    } else {
        tmp[k] = a[j];
        ++j;
    }
    ++k;
}
// se o índice i chegou ao fim de seu arranjo primeiro
if (i == meio) {
    // os outros elementos do segundo arranjo vão para o arranjo temporário
    while (j < n) {
        tmp[k] = a[j];
        ++j;
        ++k;
    }
```

A estrutura de repetição continua até que `i` atinja `meio` ou `j` atinja `n`

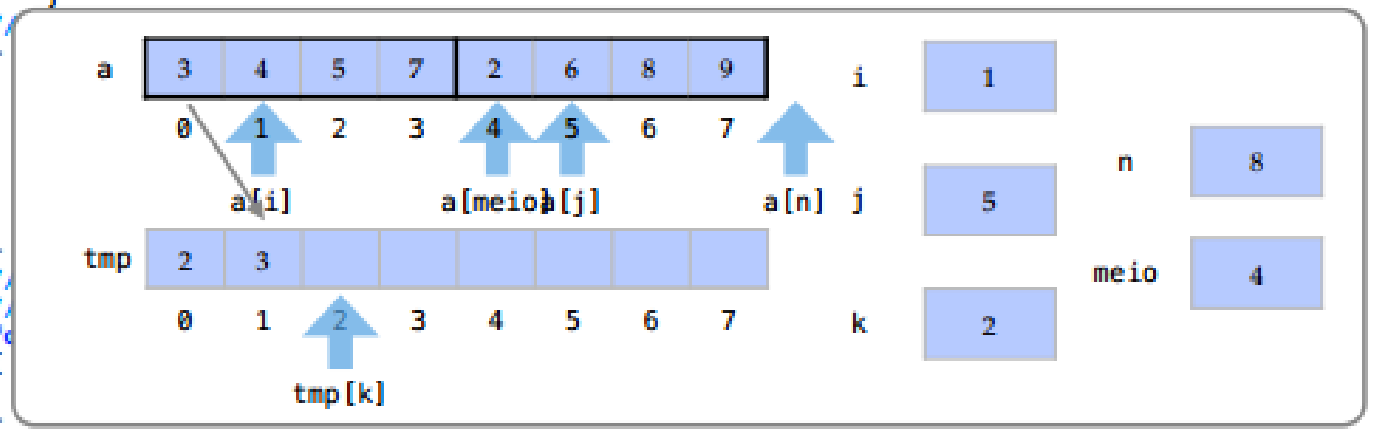


MergeSort – Etapa de intercalação

```
int meio = n / 2; // Indice que marca o meio do arranjo
int i, j, k; // Índices para a estrutura de repetição
i = 0; // Índice i marca itens intercalados do primeiro arranjo
j = meio; // Índice j marca itens intercalados do segundo arranjo
k = 0; // Índice k marca itens já intercalados no arranjo temporário
// Enquanto os índices i e j não tenham chegado ao fim de seus arranjos
while (i < meio && j < n){
    // colocamos o menor item entre a[i] e a[j] no arranjo temporário
    if (a[i] < a[j]){
        tmp[k] = a[i];
        ++i;
    } else {
        tmp[k] = a[j];
        ++j;
    }
    ++k;
}
// se o índice i chegou ao fim de seu arranjo primeiro
if (i == meio) {
    // os outros elementos do segundo arranjo vão para o arranjo temporário
    while (j < n) {
        tmp[k] = a[j];
        ++j;
        ++k;
    }
```

a[i] < a[j] é true

Então a[i] é copiado para tmp[k]

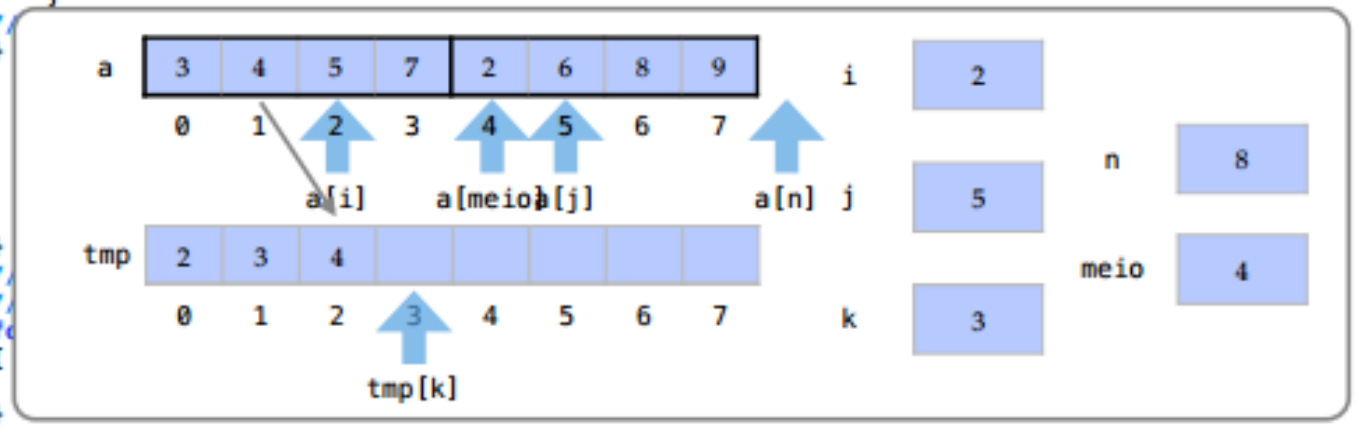


MergeSort – Etapa de intercalação

```
int meio = n / 2; // Indice que marca o meio do arranjo
int i, j, k; // Índices para a estrutura de repetição
i = 0; // Índice i marca itens intercalados do primeiro arranjo
j = meio; // Índice j marca itens intercalados do segundo arranjo
k = 0; // Índice k marca itens já intercalados no arranjo temporário
// Enquanto os índices i e j não tenham chegado ao fim de seus arranjos
while (i < meio && j < n){
    // colocamos o menor item entre a[i] e a[j] no arranjo temporário
    if (a[i] < a[j]){
        tmp[k] = a[i];
        ++i;
    } else {
        tmp[k] = a[j];
        ++j;
    }
    ++k;
}
// se o índice i chegou ao fim de seu arranjo primeiro
if (i == meio) {
    // os outros elementos do segundo arranjo vão para o arranjo temporário
    while (j < n) {
        tmp[k] = a[j];
        ++j;
        ++k;
    }
```

a[i] < a[j] é true

Então a[i] é copiado para tmp[k]

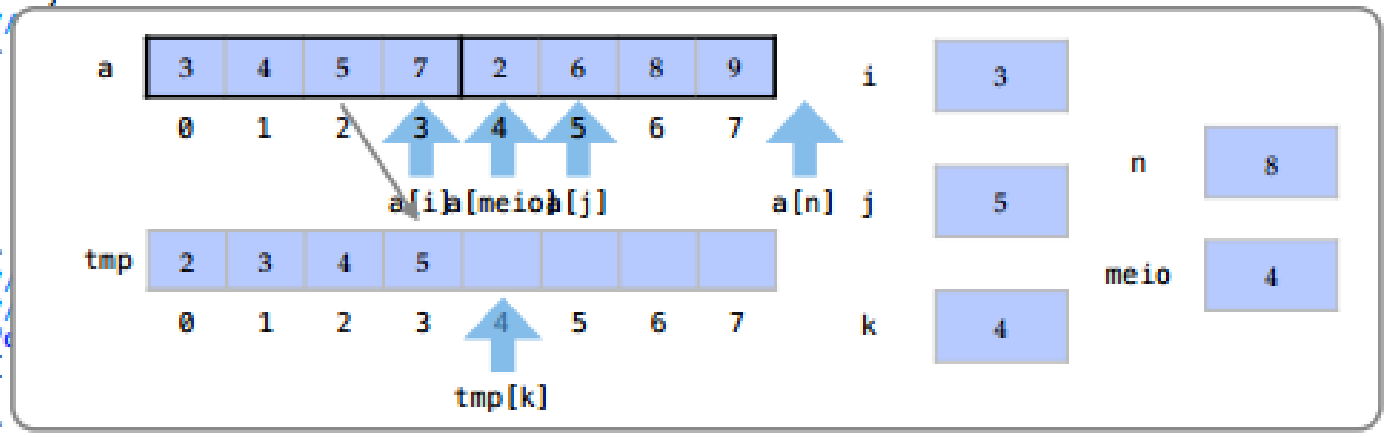


MergeSort – Etapa de intercalação

```
int meio = n / 2; // Índice que marca o meio do arranjo
int i, j, k; // Índices para a estrutura de repetição
i = 0; // Índice i marca itens intercalados do primeiro arranjo
j = meio; // Índice j marca itens intercalados do segundo arranjo
k = 0; // Índice k marca itens já intercalados no arranjo temporário
// Enquanto os índices i e j não tenham chegado ao fim de seus arranjos
while (i < meio && j < n){
    // colocamos o menor item entre a[i] e a[j] no arranjo temporário
    if (a[i] < a[j]){
        tmp[k] = a[i];
        ++i;
    } else {
        tmp[k] = a[j];
        ++j;
    }
    ++k;
}
// se o índice i chegou ao fim de seu arranjo primeiro
if (i == meio) {
    // os outros elementos do segundo arranjo vão para o arranjo temporário
    while (j < n) {
        tmp[k] = a[j];
        ++j;
        ++k;
    }
```

a[i] < a[j] é true

Então a[i] é copiado para tmp[k]

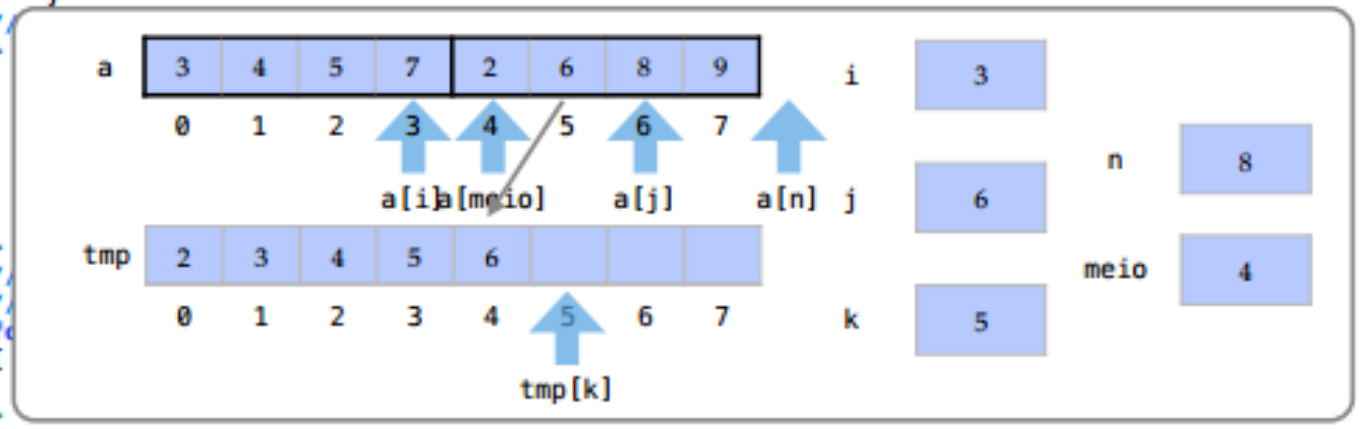


MergeSort – Etapa de intercalação

```
int meio = n / 2; // Índice que marca o meio do arranjo
int i, j, k; // Índices para a estrutura de repetição
i = 0; // Índice i marca itens intercalados do primeiro arranjo
j = meio; // Índice j marca itens intercalados do segundo arranjo
k = 0; // Índice k marca itens já intercalados no arranjo temporário
// Enquanto os índices i e j não tenham chegado ao fim de seus arranjos
while (i < meio && j < n){
    // colocamos o menor item entre a[i] e a[j] no arranjo temporário
    if (a[i] < a[j]){
        tmp[k] = a[i];
        ++i;
    } else {
        tmp[k] = a[j];
        ++j;
    }
    ++k;
}
// se o índice i chegou ao fim de seu arranjo primeiro
if (i == meio) {
    // os outros elementos do segundo arranjo vão para o arranjo temporário
    while (j < n) {
        tmp[k] = a[j];
        ++j;
        ++k;
    }
```

a[i] < a[j] é false

Então a[j] é copiado para tmp[k]

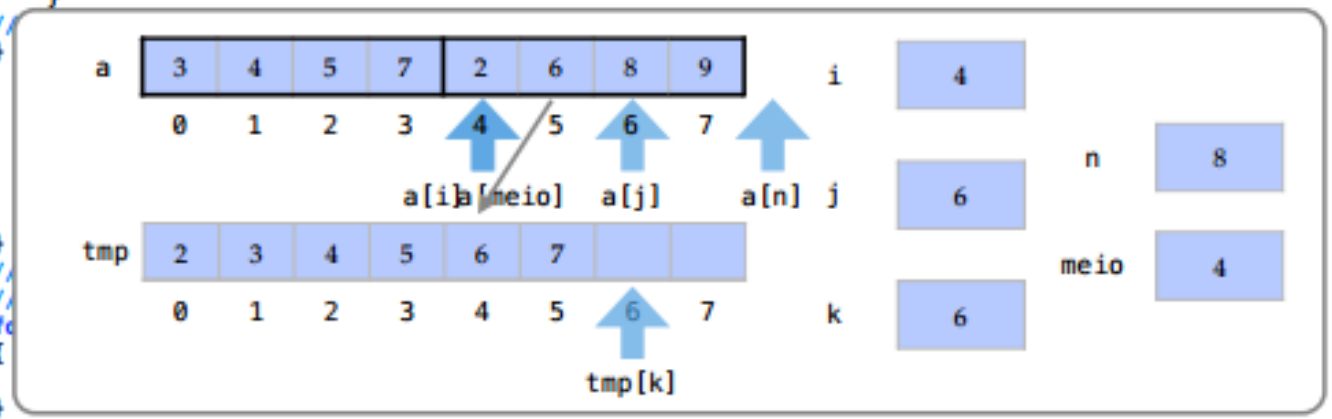


MergeSort – Etapa de intercalação

```
int meio = n / 2; // Índice que marca o meio do arranjo
int i, j, k; // Índices para a estrutura de repetição
i = 0; // Índice i marca itens intercalados do primeiro arranjo
j = meio; // Índice j marca itens intercalados do segundo arranjo
k = 0; // Índice k marca itens já intercalados no arranjo temporário
// Enquanto os índices i e j não tenham chegado ao fim de seus arranjos
while (i < meio && j < n){
    // colocamos o menor item entre a[i] e a[j] no arranjo temporário
    if (a[i] < a[j]){
        tmp[k] = a[i];
        ++i;
    } else {
        tmp[k] = a[j];
        ++j;
    }
    ++k;
}
// se o índice i chegou ao fim de seu arranjo primeiro
if (i == meio) {
    // os outros elementos do segundo arranjo vão para o arranjo temporário
    while (j < n) {
        tmp[k] = a[j];
        ++j;
        ++k;
    }
```

a[i] < a[j] é true

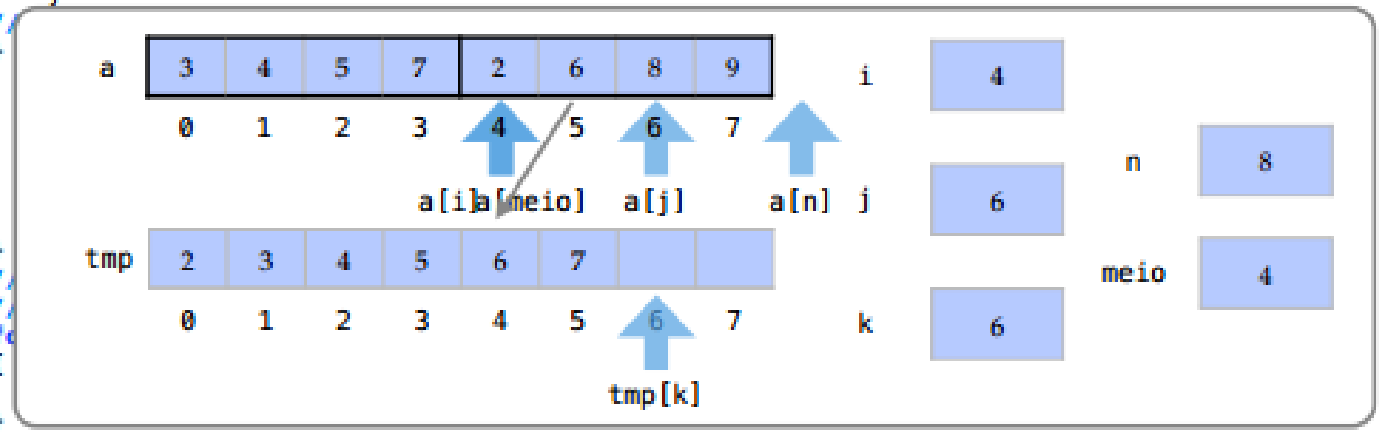
Então a[i] é copiado para tmp[k]



MergeSort – Etapa de intercalação

```
int meio = n / 2; // Índice que marca o meio do arranjo
int i, j, k; // Índices para a estrutura de repetição
i = 0; // Índice i marca itens intercalados do primeiro arranjo
j = meio; // Índice j marca itens intercalados do segundo arranjo
k = 0; // Índice k marca itens já intercalados no arranjo temporário
// Enquanto os índices i e j não tenham chegado ao fim de seus arranjos
while (i < meio && j < n){
    // colocamos o menor item entre a[i] e a[j] no arranjo temporário
    if (a[i] < a[j]){
        tmp[k] = a[i];
        ++i;
    } else {
        tmp[k] = a[j];
        ++j;
    }
    ++k;
}
// se o índice i chegou ao fim de seu arranjo primeiro
if (i == meio) {
    // os outros elementos do segundo arranjo vão para o arranjo temporário
    while (j < n) {
        tmp[k] = a[j];
        ++j;
        ++k;
    }
```

Como `i` tem os mesmo valor de `meio`, o primeiro arranjo está intercalado e a repetição termina.



MergeSort – Etapa de intercalação

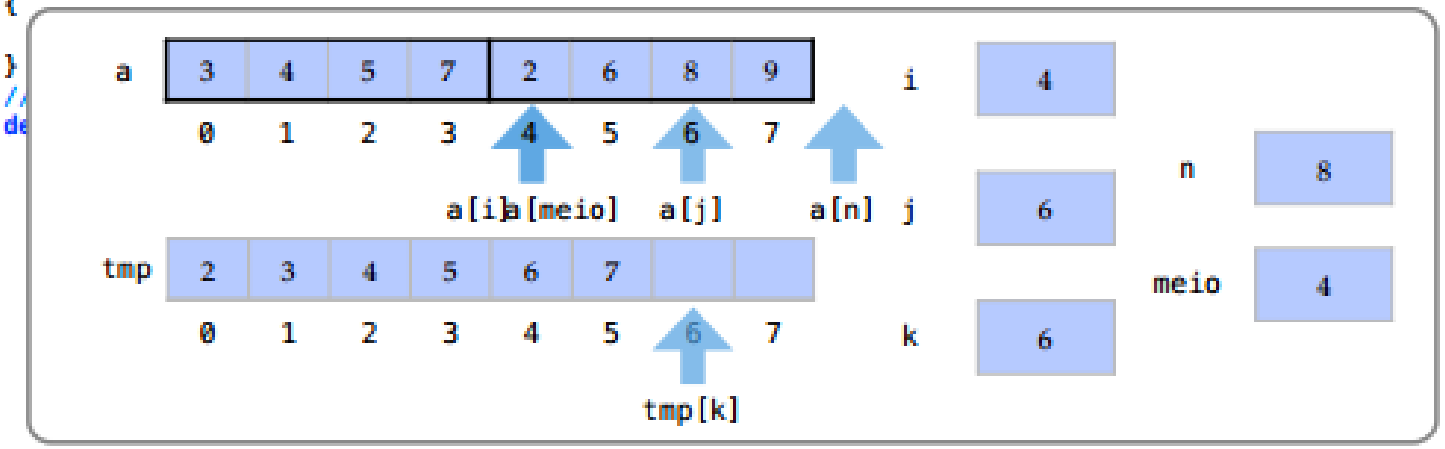
```
        ++k;
    }
    // se o índice i chegou ao fim de seu arranjo primeiro
    if (i == meio) {
        // os outros elementos do segundo arranjo vão para o arranjo temporário
        while (j < n) {
            tmp[k] = a[j];
            ++j;
            ++k;
        }
    // se foi o índice j que chegou ao fim de seu arranjo primeiro
    } else {
        // os outros elementos do primeiro arranjo vão para o arranjo temporário
        while (i < meio) {
            tmp[k] = a[i];
            ++i;
            ++k;
        }
    }
    // neste ponto, o arranjo temporário tem todos os elementos intercalados
    // estes elementos são copiados de volta para o arranjo int a[]
    for (i = 0; i < n; ++i)
    {

    }
    //
    de
}
```

Queremos agora saber se...

A repetição anterior terminou porque

i == meio ou porque j == n?



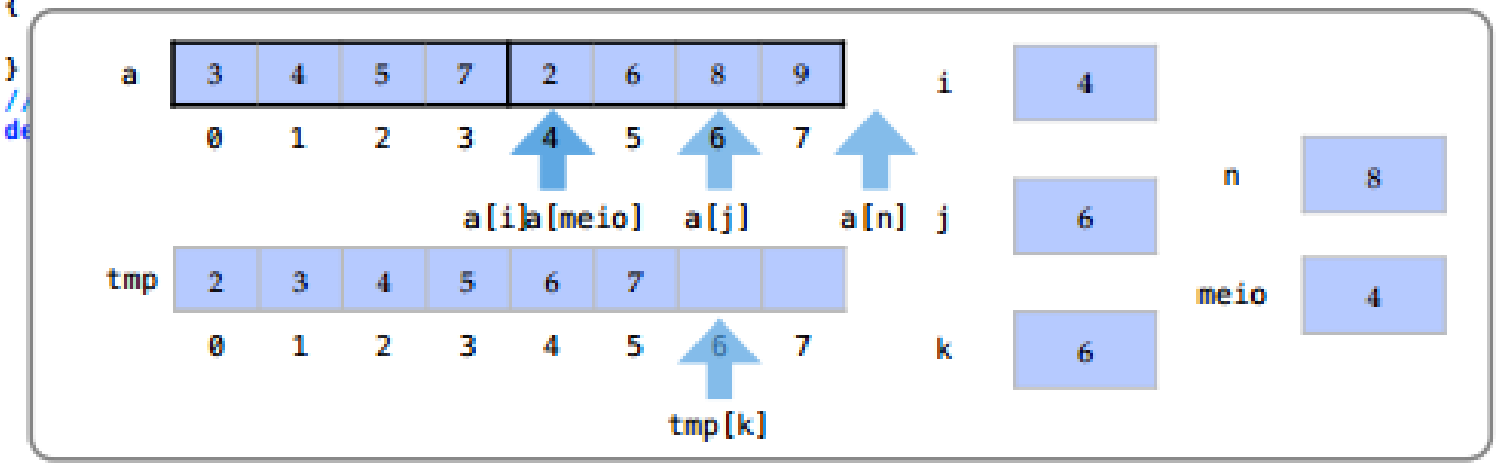
MergeSort – Etapa de intercalação

```
        ++k;
    }
    // se o índice i chegou ao fim de seu arranjo primeiro
    if (i == meio) {
        // os outros elementos do segundo arranjo vão para o arranjo temporário
        while (j < n) {
            tmp[k] = a[j];
            ++j;
            ++k;
        }
    // se foi o índice j que chegou ao fim de seu arranjo primeiro
    } else {
        // os outros elementos do primeiro arranjo vão para o arranjo temporário
        while (i < meio) {
            tmp[k] = a[i];
            ++i;
            ++k;
        }
    }
    // neste ponto, o arranjo temporário tem todos os elementos intercalados
    // estes elementos são copiados de volta para o arranjo int a[]
    for (i = 0; i < n; ++i)
    {

    }
    //
    de
}
```

true

Neste caso, i==meio.



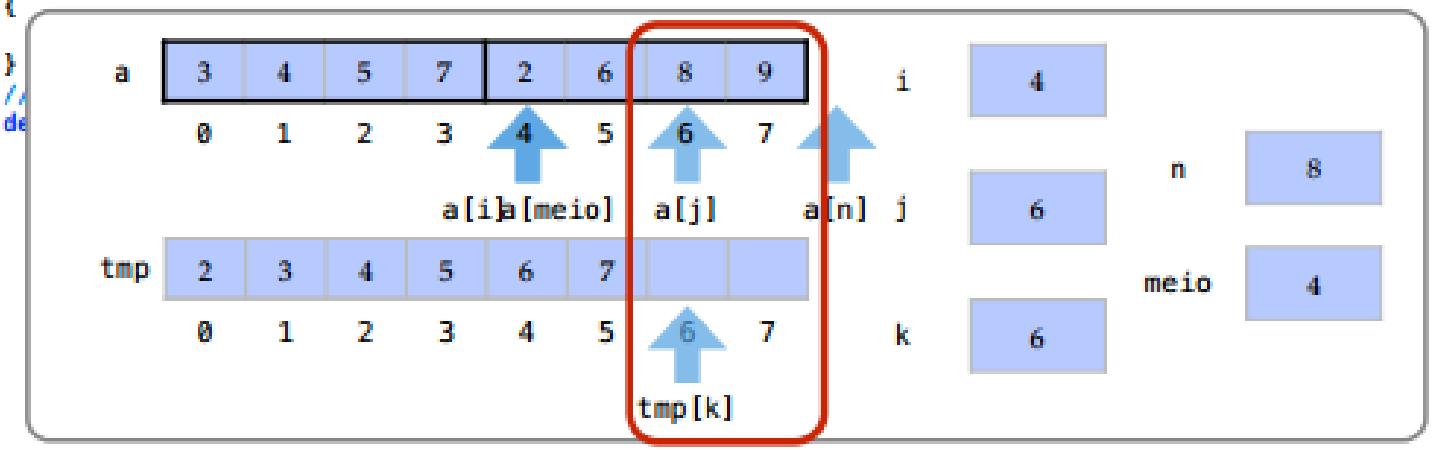
MergeSort – Etapa de intercalação

```
        }
        ++k;
    }
    // se o índice i chegou ao fim de seu arranjo primeiro
    if (i == meio) {
        // os outros elementos do segundo arranjo vão para o arranjo temporário
        while (j < n) {
            tmp[k] = a[j];
            ++j;
            ++k;
        }
    // se foi o índice j que chegou ao fim de seu arranjo primeiro
    } else {
        // os outros elementos do primeiro arranjo vão para o arranjo temporário
        while (i < meio) {
            tmp[k] = a[i];
            ++i;
            ++k;
        }
    }
    // neste ponto, o arranjo temporário tem todos os elementos intercalados
    // estes elementos são copiados de volta para o arranjo int a[]
    for (i = 0; i < n; ++i)
    {
    }
    /
    de
}
```

true

Neste caso, i==meio. Isto quer dizer que...

...os outros elementos do segundo arranjo ainda precisam ser copiados.



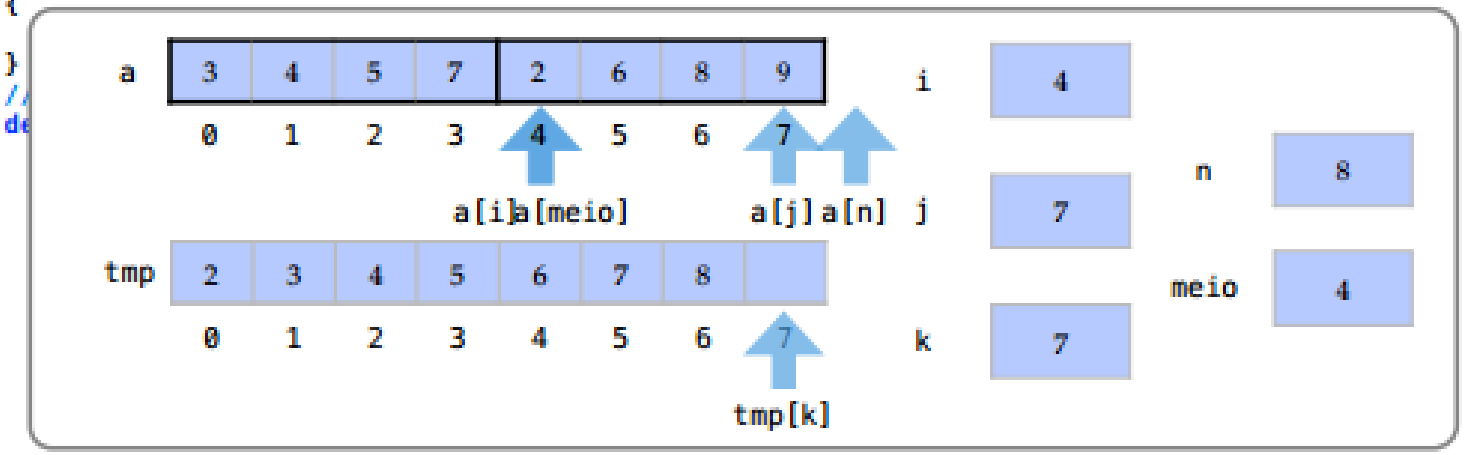
MergeSort – Etapa de intercalação

```
        ++k;
    }
    // se o índice i chegou ao fim de seu arranjo primeiro
    if (i == meio) {
        // os outros elementos do segundo arranjo vão para o arranjo temporário
        while (j < n) {
            tmp[k] = a[j];
            ++j;
            ++k;
        }
    // se foi o índice j que chegou ao fim de seu arranjo primeiro
    } else {
        // os outros elementos do primeiro arranjo vão para o arranjo temporário
        while (i < meio) {
            tmp[k] = a[i];
            ++i;
            ++k;
        }
    }
    // neste ponto, o arranjo temporário tem todos os elementos intercalados
    // estes elementos são copiados de volta para o arranjo int a[]
    for (i = 0; i < n; ++i)
    {

    }
    //
    de
}
```

Enquanto j for menor que n

Copiamos a[j] para a[k] e incrementamos j e k



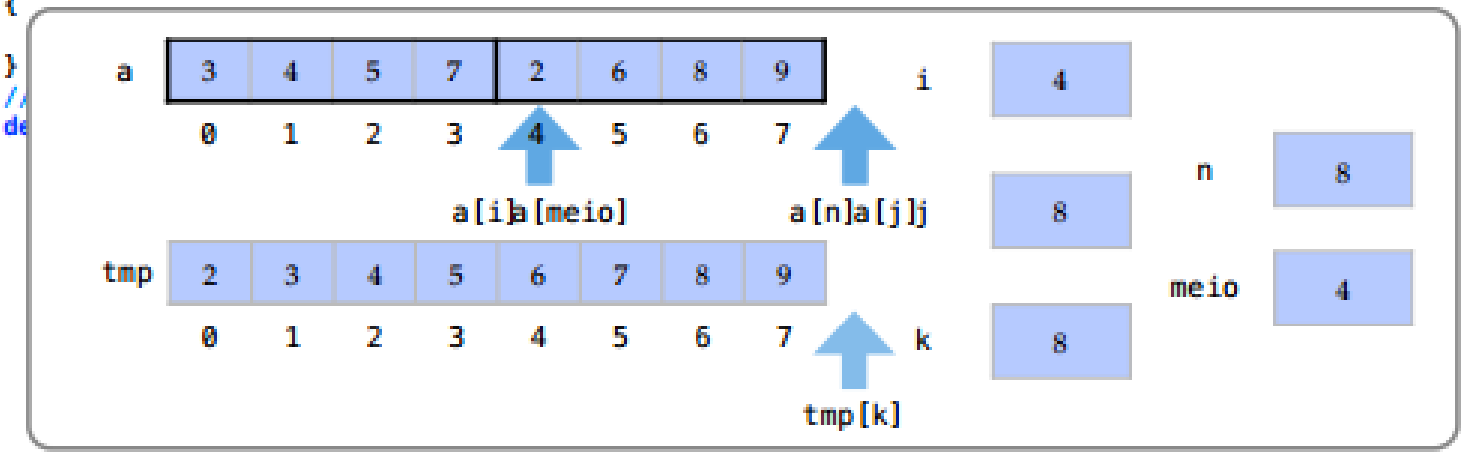
MergeSort – Etapa de intercalação

```
        ++k;
    }
    // se o índice i chegou ao fim de seu arranjo primeiro
    if (i == meio) {
        // os outros elementos do segundo arranjo vão para o arranjo temporário
        while (j < n) {
            tmp[k] = a[j];
            ++j;
            ++k;
        }
    // se foi o índice j que chegou ao fim de seu arranjo primeiro
    } else {
        // os outros elementos do primeiro arranjo vão para o arranjo temporário
        while (i < meio) {
            tmp[k] = a[i];
            ++i;
            ++k;
        }
    }
    // neste ponto, o arranjo temporário tem todos os elementos intercalados
    // estes elementos são copiados de volta para o arranjo int a[]
    for (i = 0; i < n; ++i)
    {

    }
    //
    de
}
```

Enquanto j for menor que n

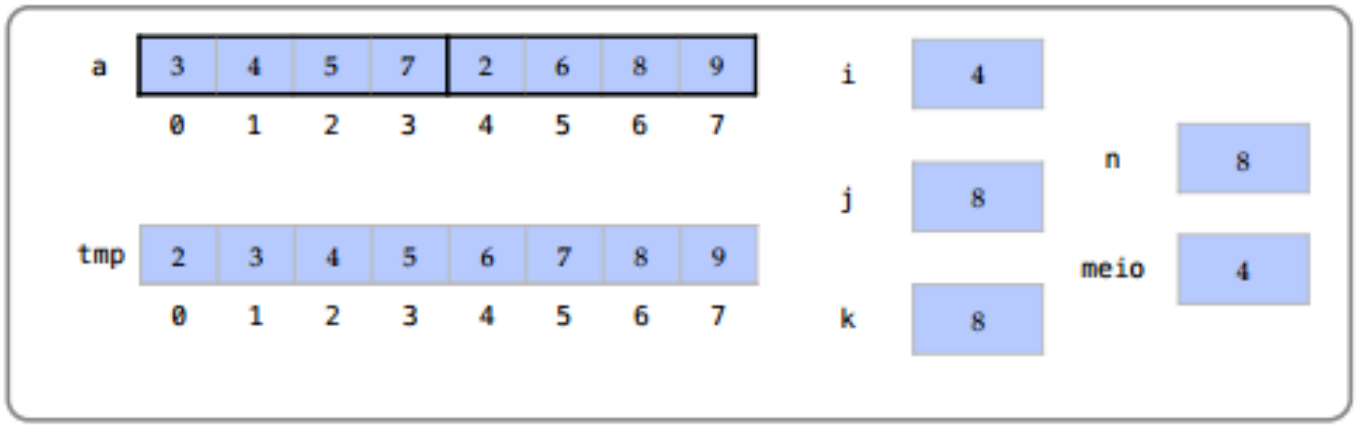
Copiamos a[j] para a[k] e incrementamos j e k



MergeSort – Etapa de intercalação

```
    // se foi o índice j que chegou ao fim de seu arranjo primeiro
    } else {
        // os outros elementos do primeiro arranjo vão para o arranjo temporário
        while (i < meio) {
            tmp[k] = a[i];
            ++i;
            ++k;
        }
    }
    // neste ponto, o arranjo temporário tem todos os elementos intercalados
    // estes elementos são copiados de volta para o arranjo int a[]
    for (i = 0; i < n; ++i)
    {
        a[i] = tmp[i];
    }
    // o arranjo temporário pode então ser desalocado da memória
    delete [] tmp;
}
```

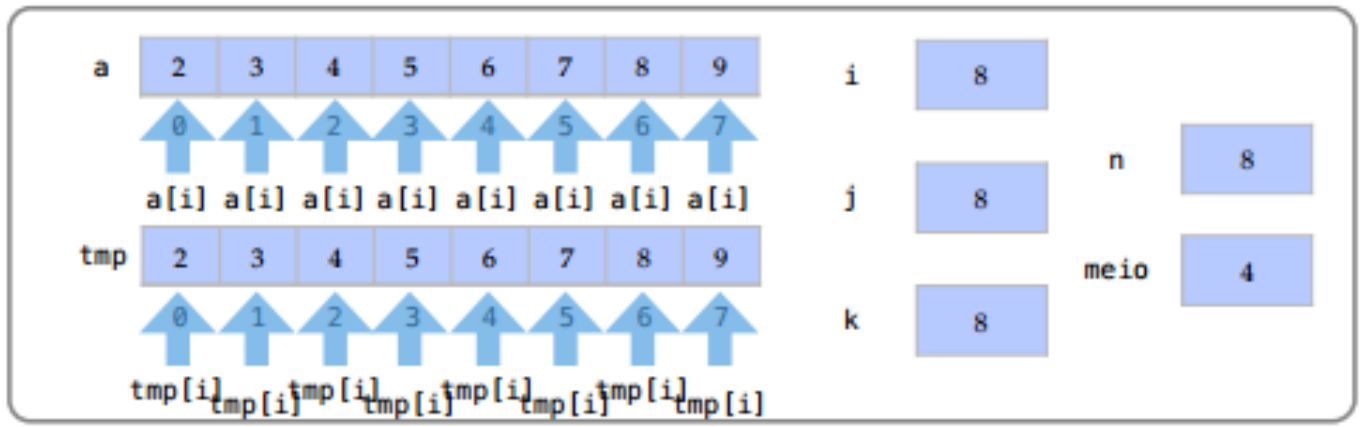
Ao chegar neste ponto, o vetor `temp` tem todos os elementos intercalados



MergeSort – Etapa de intercalação

```
    // se foi o índice j que chegou ao fim de seu arranjo primeiro
    } else {
        // os outros elementos do primeiro arranjo vão para o arranjo temporário
        while (i < meio) {
            tmp[k] = a[i];
            ++i;
            ++k;
        }
    }
    // neste ponto, o arranjo temporário tem todos os elementos intercalados
    // estes elementos são copiados de volta para o arranjo int a[]
    for (i = 0; i < n; ++i)
    {
        a[i] = tmp[i];
    }
    // o arranjo temporário pode então ser desalocado da memória
    delete [] tmp;
}
```

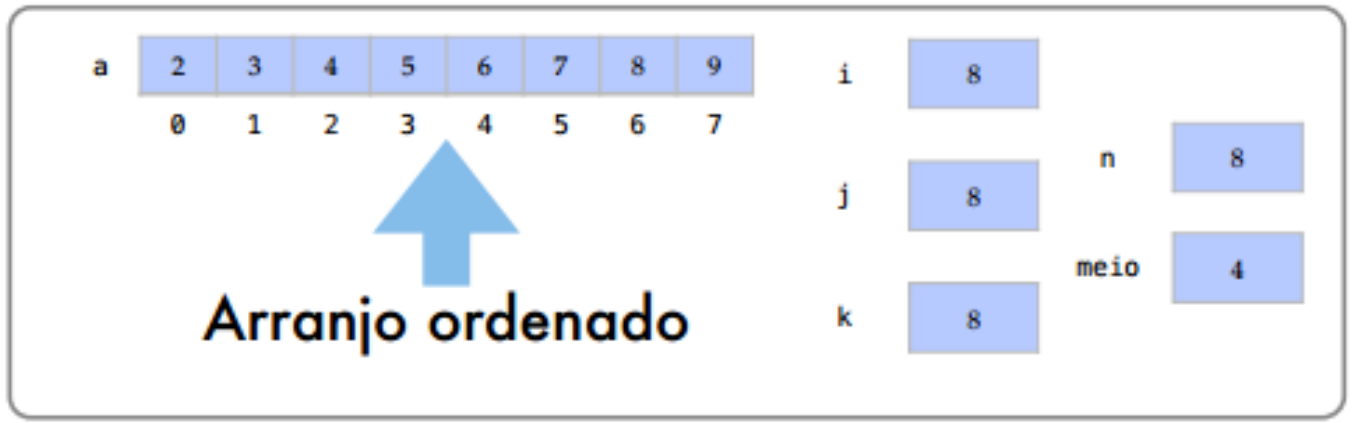
Copiamos então todos os elementos de volta para o arranjo a



MergeSort – Etapa de intercalação

```
    // se foi o índice j que chegou ao fim de seu arranjo primeiro
    } else {
        // os outros elementos do primeiro arranjo vão para o arranjo temporário
        while (i < meio) {
            tmp[k] = a[i];
            ++i;
            ++k;
        }
    }
    // neste ponto, o arranjo temporário tem todos os elementos intercalados
    // estes elementos são copiados de volta para o arranjo int a[]
    for (i = 0; i < n; ++i)
    {
        a[i] = tmp[i];
    }
    // o arranjo temporário pode então ser desalocado da memória
    delete [] tmp;
}
```

Agora, com os elementos já intercalados em a[], podemos deslocar o arranjo temporário e finalizar a função.



MergeSort – Algoritmo de Ordenação

- A utilidade de funções é justamente quebrar problemas em problemas menores
- Usando a função apresentada de intercalação como uma função auxiliar, podemos definir a função de ordenação
- A função de ordenação pode ser facilmente expressa em termos recursivos

MergeSort – Algoritmo de Ordenação

- Utilizando a função de intercalação como uma função auxiliar, esta função recursiva ordena um arranjo com o método MergeSort

```
void mergeSort(int a[], int n) {
  int meio;
  if (n > 1) {
    meio = n / 2;
    mergeSort(a, meio);
    mergeSort(a + meio, n - meio);
    intercala(a, n);
  }
}
```

MergeSort – Algoritmo de Ordenação

- Dividimos o arranjo ao meio

```
void mergeSort(int a[], int n) {
  int meio;
  if (n > 1) {
    meio = n / 2;
    mergeSort(a, meio);
    mergeSort(a + meio, n - meio);
    intercala(a, n);
  }
}
```

MergeSort – Algoritmo de Ordenação

- Usamos o próprio método recursivamente para ordenar o arranjo até a metade

```
void mergeSort(int a[], int n) {
  int meio;
  if (n > 1) {
    meio = n / 2;
    mergeSort(a, meio);
    mergeSort(a + meio, n - meio);
    intercala(a, n);
  }
}
```

MergeSort – Algoritmo de Ordenação

- Usamos o próprio método recursivamente para ordenar o arranjo da metade até o final

```
void mergeSort(int a[], int n) {
  int meio;
  if (n > 1) {
    meio = n / 2;
    mergeSort(a, meio);
    mergeSort(a + meio, n - meio);
    intercala(a, n);
  }
}
```

MergeSort – Algoritmo de Ordenação

- Intercalamos os dois arranjos ordenados com a função já declarada. O primeiro de a[0] até a[meio-1] e o outro de a[meio] até a[n-1]

```
void mergeSort(int a[], int n) {
  int meio;
  if (n > 1) {
    meio = n / 2;
    mergeSort(a, meio);
    mergeSort(a + meio, n - meio);
    intercala(a, n);
  }
}
```

MergeSort – Algoritmo de Ordenação

- Como vimos, funções recursivas precisam de um caso base para funcionar. Caso contrário, o algoritmo entraria em uma recursão infinita

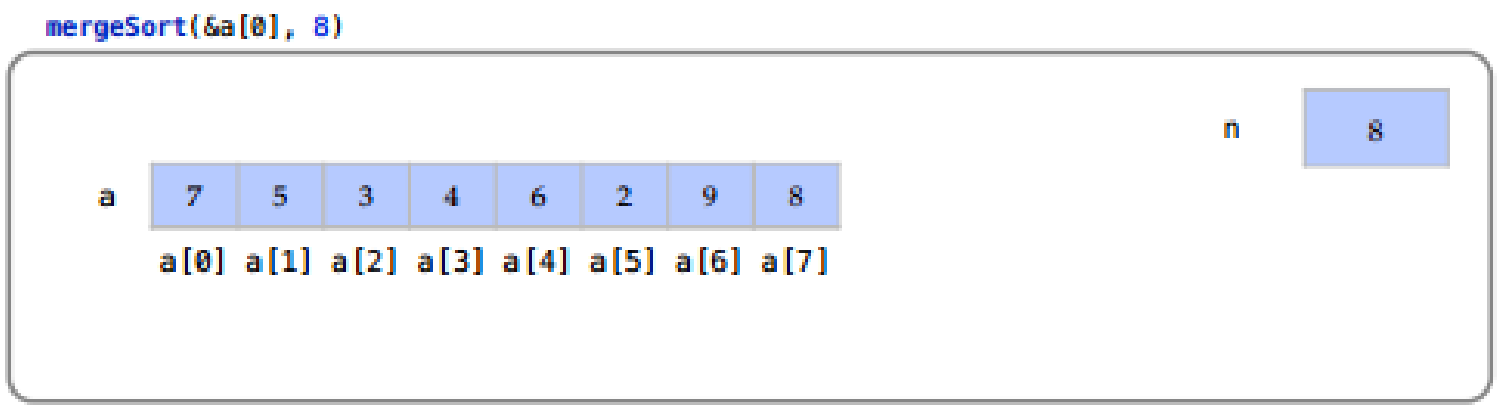
```
void mergeSort(int a[], int n) {
  int meio;
  if (n > 1) {
    meio = n / 2;
    mergeSort(a, meio);
    mergeSort(a + meio, n - meio);
    intercala(a, n);
  }
}
```

- Neste algoritmo o caso base é quando o arranjo tem apenas um elemento, pois sabemos que um arranjo de apenas um elemento já está ordenado

MergeSort – Algoritmo de Ordenação

- A função ordena o arranjo a, que contém n elementos. A função recebe o endereço de a (ou de a[0]), e ordena, 8 elementos a partir dele

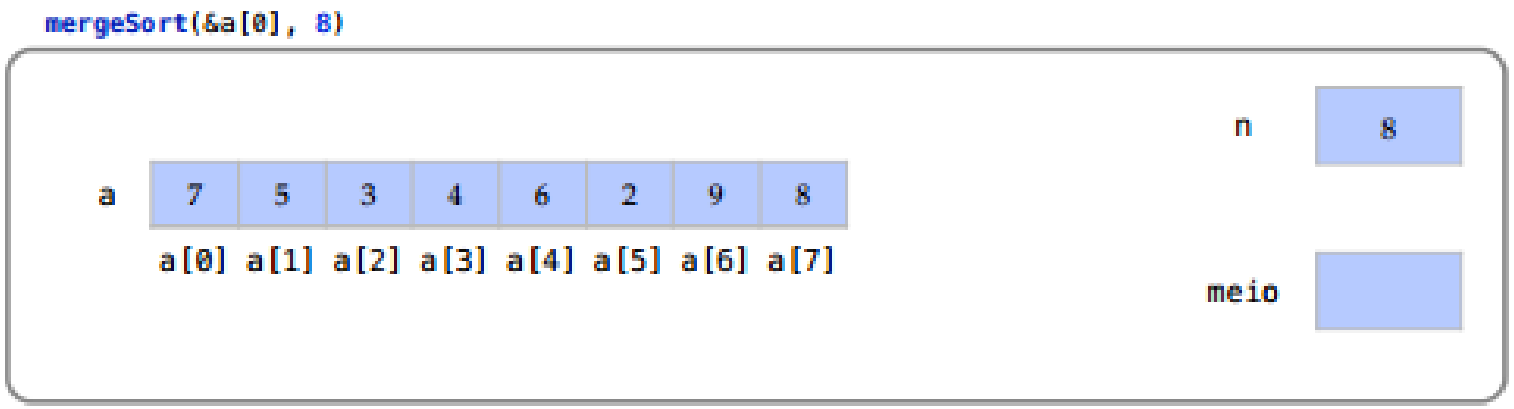
```
void mergeSort(int a[], int n) {
  int meio;
  if (n > 1) {
    meio = n / 2;
    mergeSort(a, meio);
    mergeSort(a + meio, n - meio);
    intercala(a, n);
  }
}
```



MergeSort – Algoritmo de Ordenação

- A variável meio, dividirá o arranjo em 2

```
void mergeSort(int a[], int n) {
  int meio;
  if (n > 1) {
    meio = n / 2;
    mergeSort(a, meio);
    mergeSort(a + meio, n - meio);
    intercala(a, n);
  }
}
```

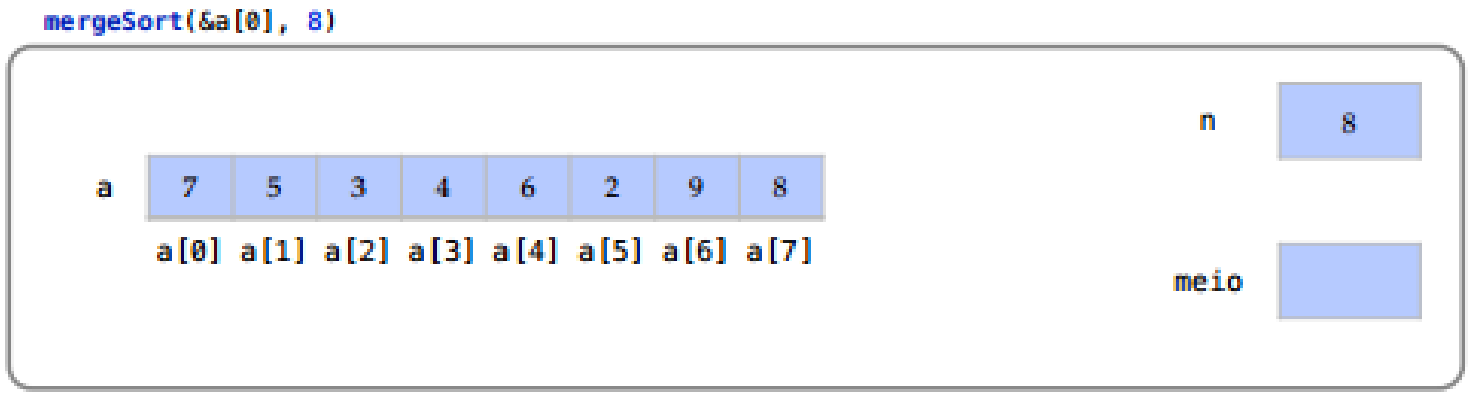


MergeSort – Algoritmo de Ordenação

- Como o tamanho do arranjo for maior que 1, prosseguiremos com o processo de divisão

```
void mergeSort(int a[], int n) {
  int meio;
  if (n > 1) {
    meio = n / 2;
    mergeSort(a, meio);
    mergeSort(a + meio, n - meio);
    intercala(a, n);
  }
}
```

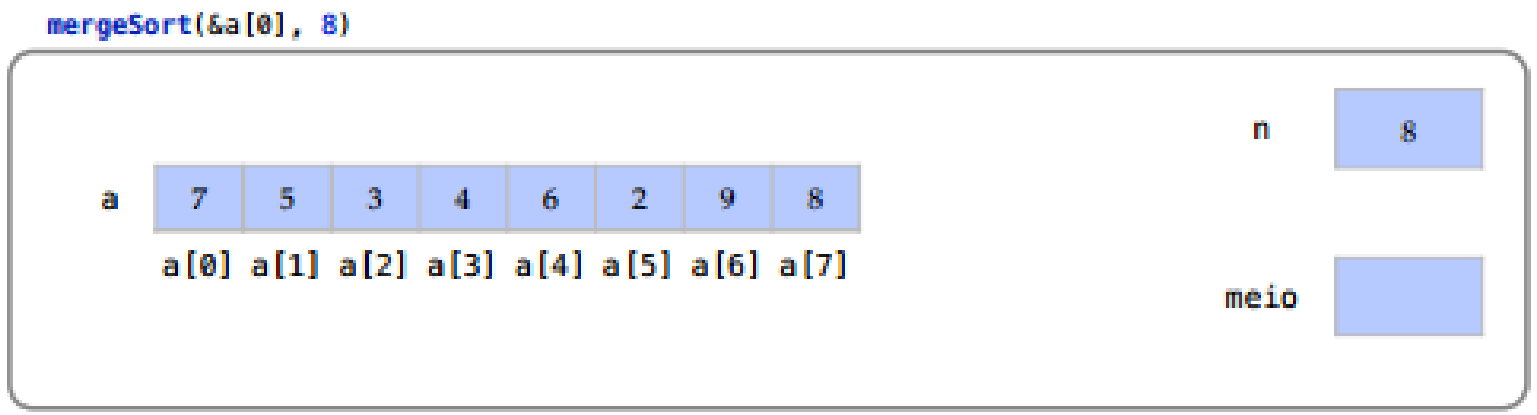
Caso contrário, se o tamanho do arranjo fosse 1, sabemos que este já seria, por definição, um arranjo ordenado.



MergeSort – Algoritmo de Ordenação

- Atualizamos a posição do meio

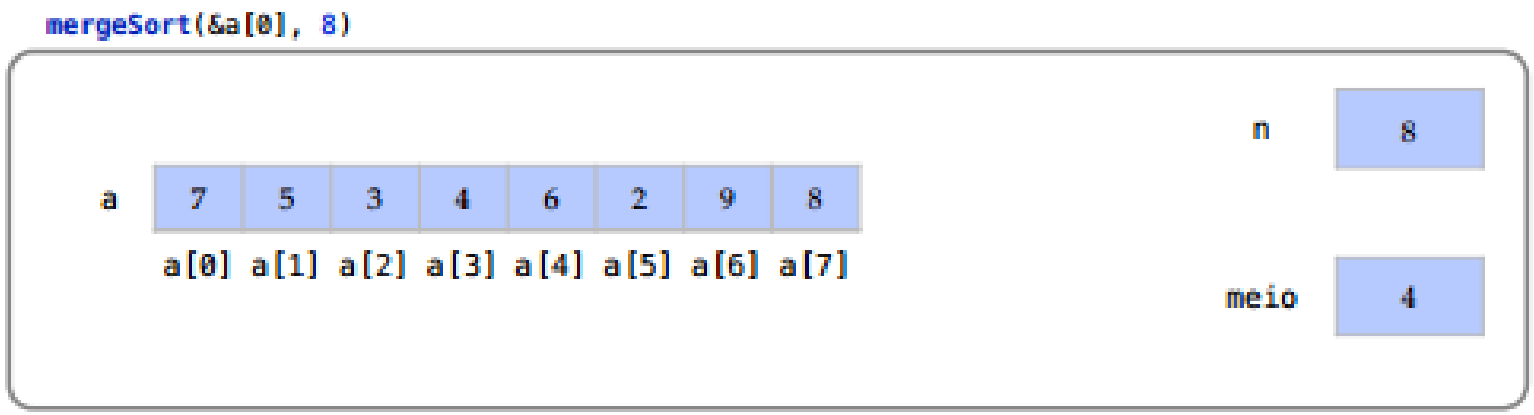
```
void mergeSort(int a[], int n) {
  int meio;
  if (n > 1) {
    meio = n / 2;
    mergeSort(a, meio);
    mergeSort(a + meio, n - meio);
    intercala(a, n);
  }
}
```



MergeSort – Algoritmo de Ordenação

- Atualizamos a posição do meio

```
void mergeSort(int a[], int n) {
  int meio;
  if (n > 1) {
    meio = n / 2;
    mergeSort(a, meio);
    mergeSort(a + meio, n - meio);
    intercala(a, n);
  }
}
```

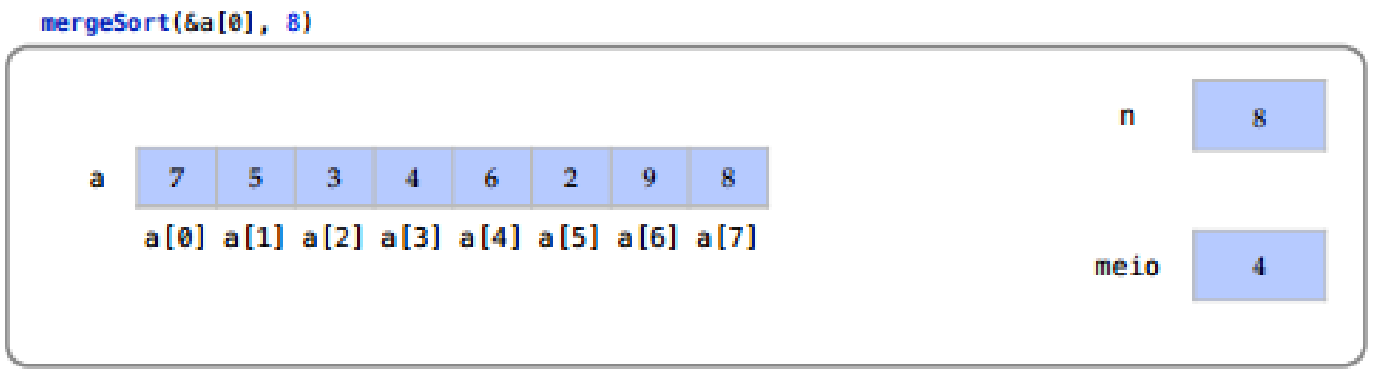


MergeSort – Algoritmo de Ordenação

- Neste passo, pedimos o algoritmo para ordenar o arranjo que se inicia na mesma posição de memória de a[] e tem tamanho meio, ou seja, 4

```
void mergeSort(int a[], int n) {
  int meio;
  if (n > 1) {
    meio = n / 2;
    mergeSort(a, meio);
    mergeSort(a + meio, n - meio);
    intercala(a, n);
  }
}
```

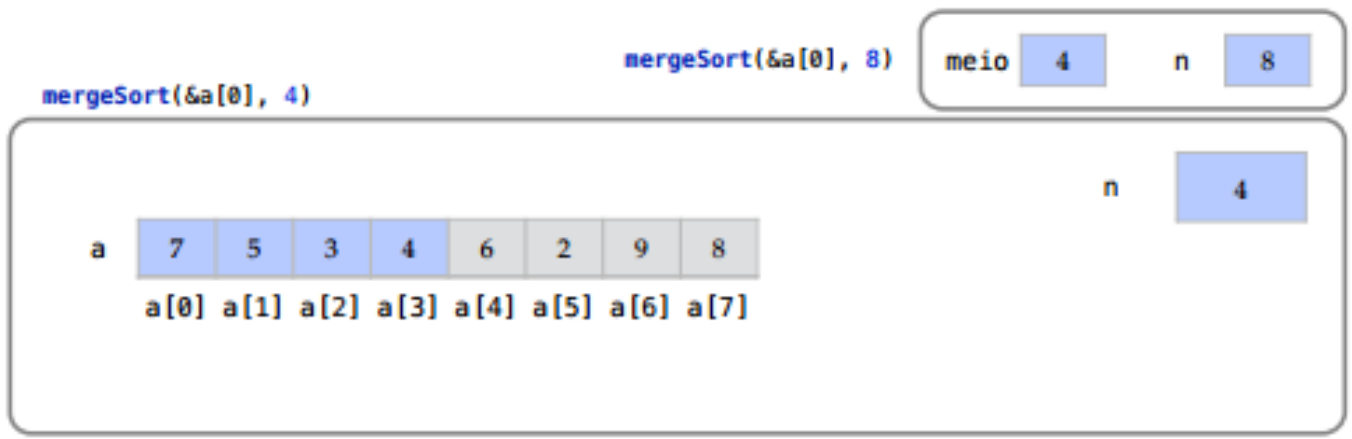
Lembre-se que um arranjo nada mais é que um endereço onde se inicia uma série de elementos alocados na memória.



MergeSort – Algoritmo de Ordenação

- A função mergeSort anterior é guardada na pilha de chamadas de função, e a nova chamada de mergeSort é executada

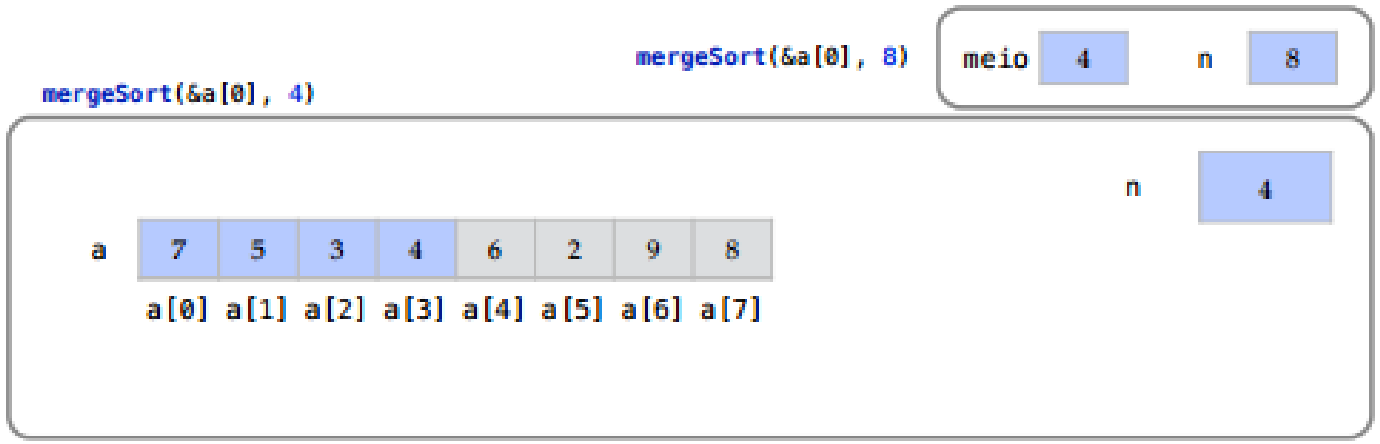
```
void mergeSort(int a[], int n) {
  int meio;
  if (n > 1) {
    meio = n / 2;
    mergeSort(a, meio);
    mergeSort(a + meio, n - meio);
    intercala(a, n);
  }
}
```



MergeSort – Algoritmo de Ordenação

- Como n é igual a 4, pedimos a esta chamada da função para considerar apenas 4 elementos a partir do endereço de a

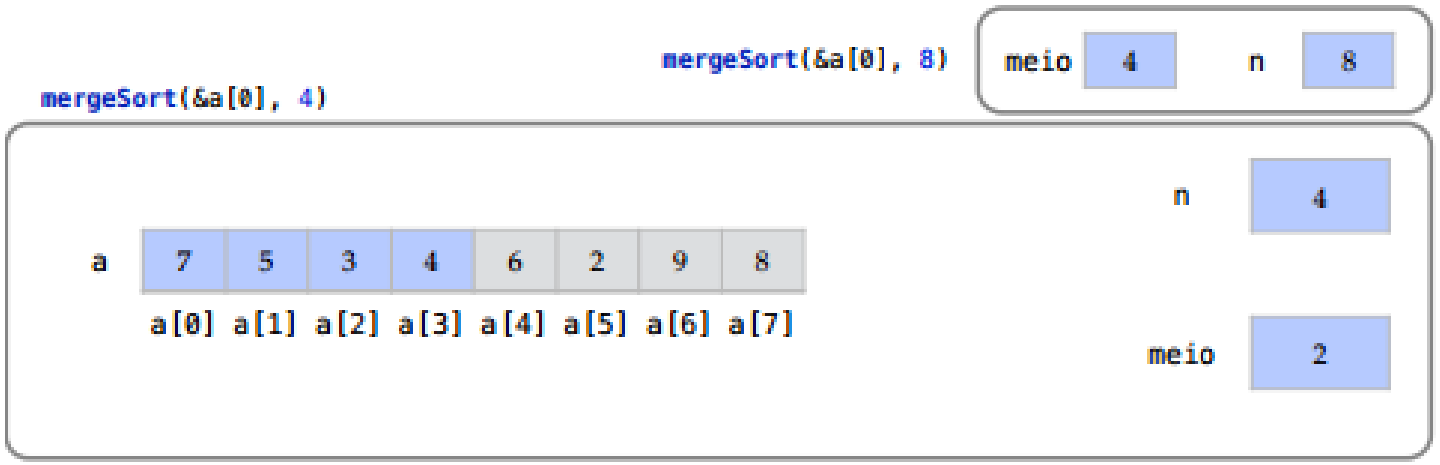
```
void mergeSort(int a[], int n) {
  int meio;
  if (n > 1) {
    meio = n / 2;
    mergeSort(a, meio);
    mergeSort(a + meio, n - meio);
    intercala(a, n);
  }
}
```



MergeSort – Algoritmo de Ordenação

- Criamos a variável que representará o meio e damos o seu valor

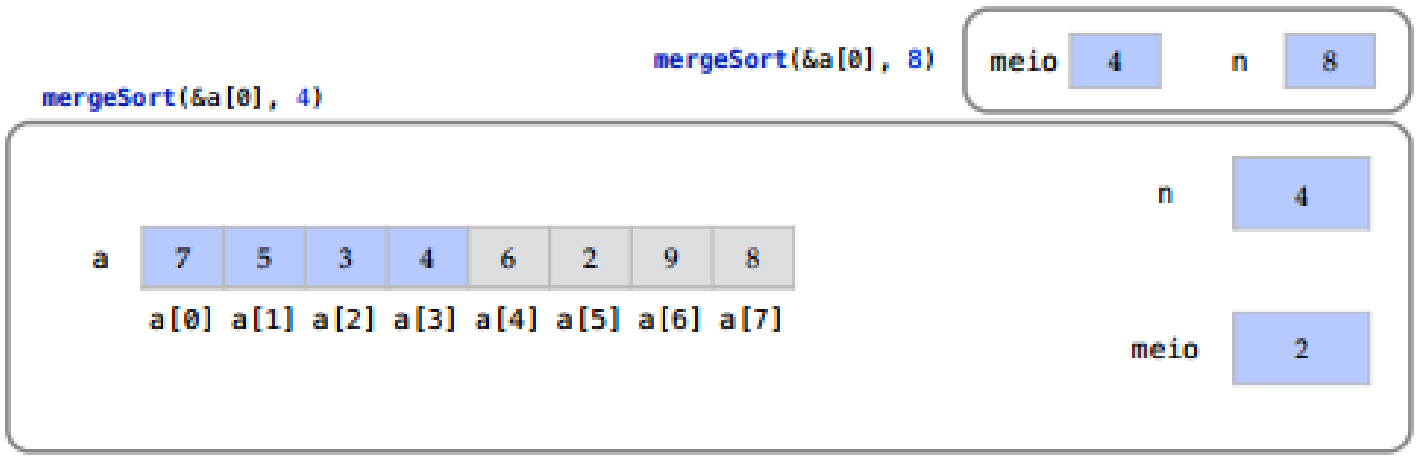
```
void mergeSort(int a[], int n) {
  int meio;
  if (n > 1) {
    meio = n / 2;
    mergeSort(a, meio);
    mergeSort(a + meio, n - meio);
    intercala(a, n);
  }
}
```



MergeSort – Algoritmo de Ordenação

- A chamada atual da função vai para a pilha e fica em espera enquanto a nova função mergeSort(a, 2) é executada

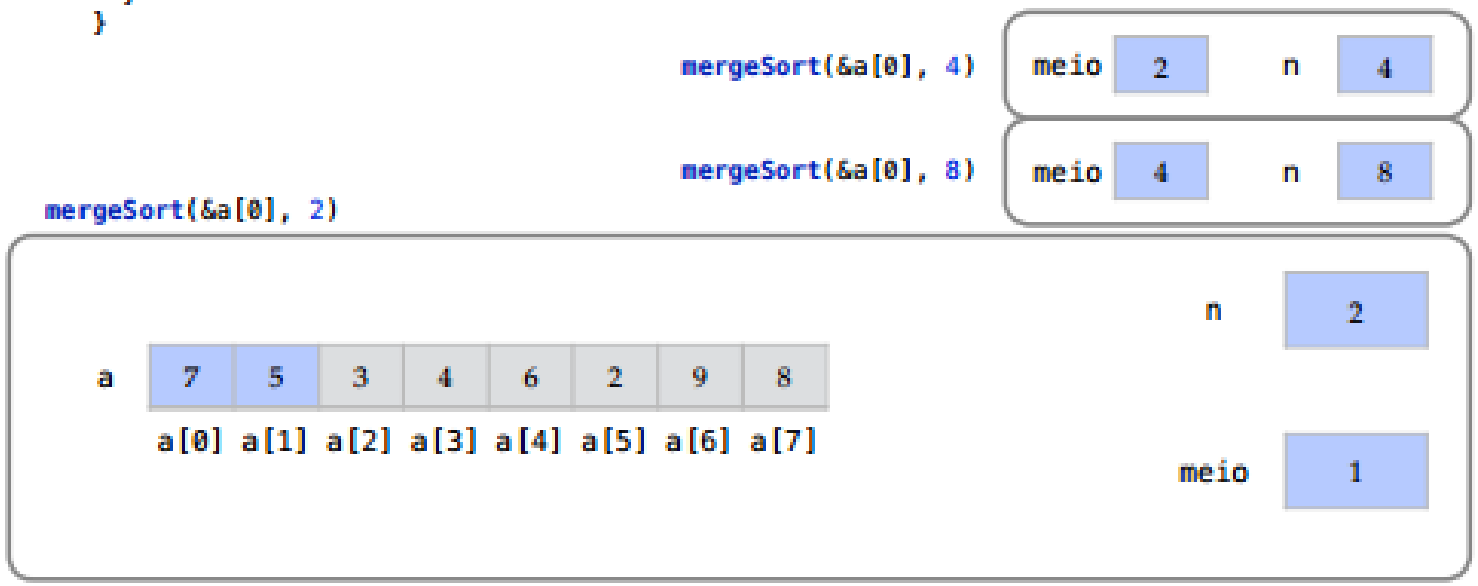
```
void mergeSort(int a[], int n) {
  int meio;
  if (n > 1) {
    meio = n / 2;
    mergeSort(a, meio);
    mergeSort(a + meio, n - meio);
    intercala(a, n);
  }
}
```



MergeSort – Algoritmo de Ordenação

- O arranjo de tamanho 2 será dividido em dois arranjos de tamanho 1

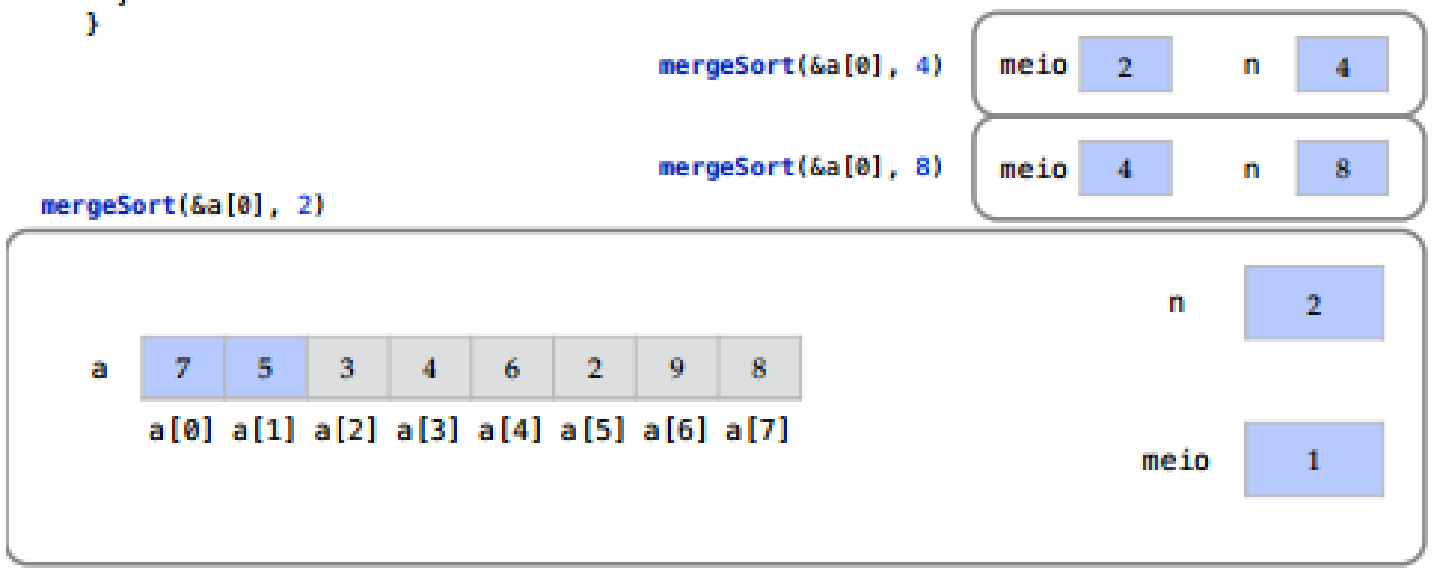
```
void mergeSort(int a[], int n) {
  int meio;
  if (n > 1) {
    meio = n / 2;
    mergeSort(a, meio);
    mergeSort(a + meio, n - meio);
    intercala(a, n);
  }
}
```



MergeSort – Algoritmo de Ordenação

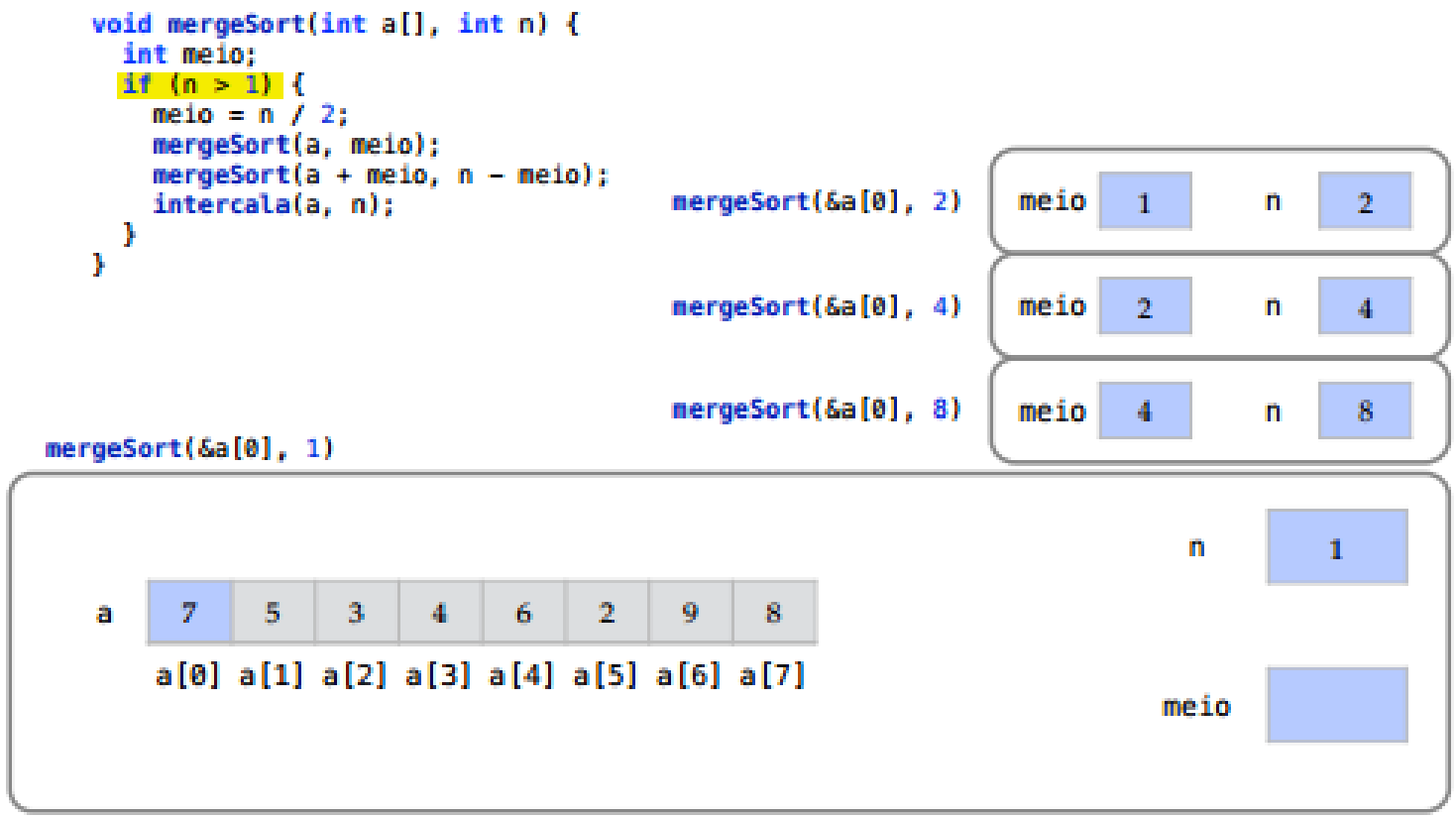
- Esta função também entrará na pilha e chamaremos a função para um arranjo de apenas um (1) elemento

```
void mergeSort(int a[], int n) {
  int meio;
  if (n > 1) {
    meio = n / 2;
    mergeSort(a, meio);
    mergeSort(a + meio, n - meio);
    intercala(a, n);
  }
}
```



MergeSort – Algoritmo de Ordenação

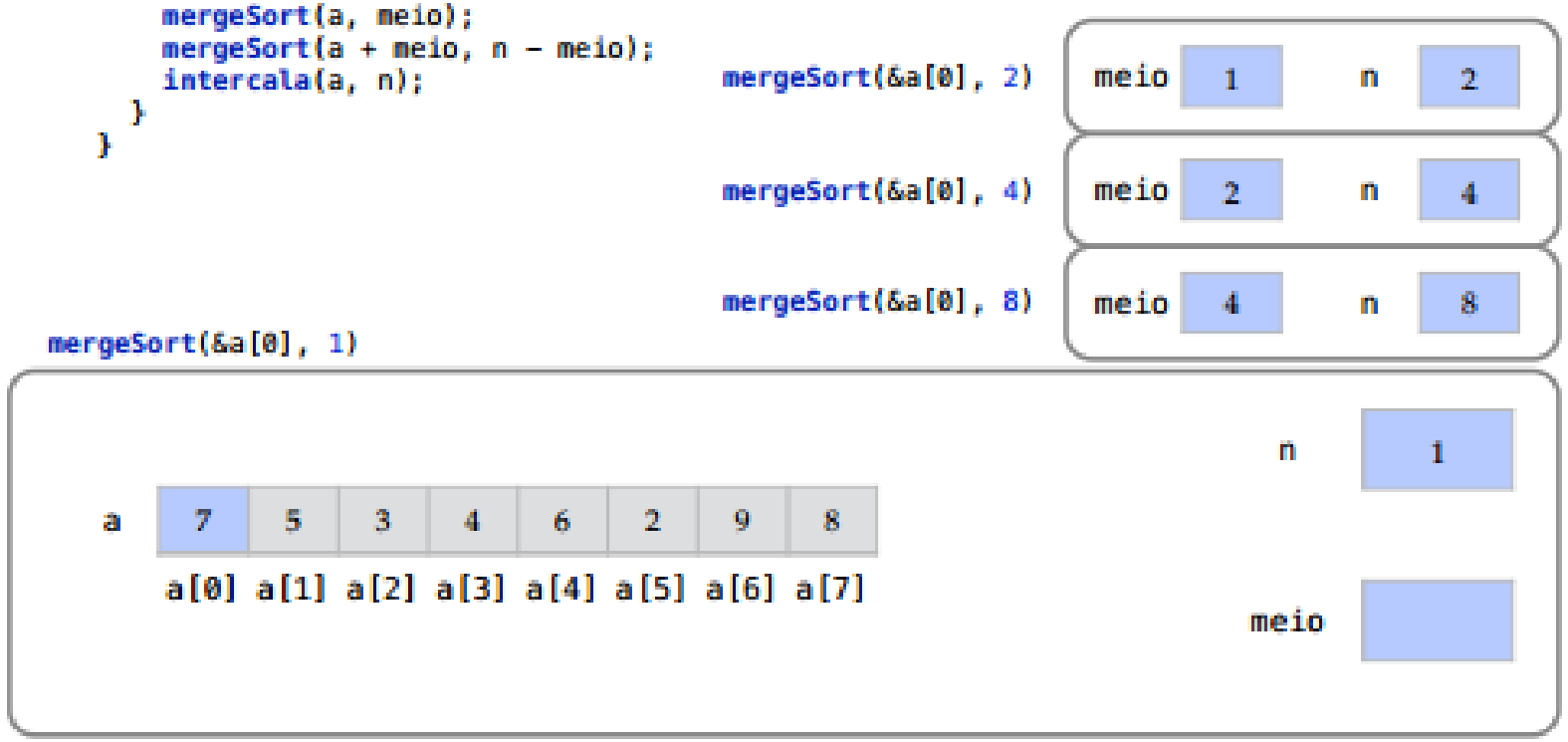
- Como estamos considerando um arranjo de apenas um elemento, porém, o caso base foi atingido e a função retorna sem fazer nada



MergeSort – Algoritmo de Ordenação

- A função retornará sem fazer nada e a função chamadora sai da pilha e volta a ser executada

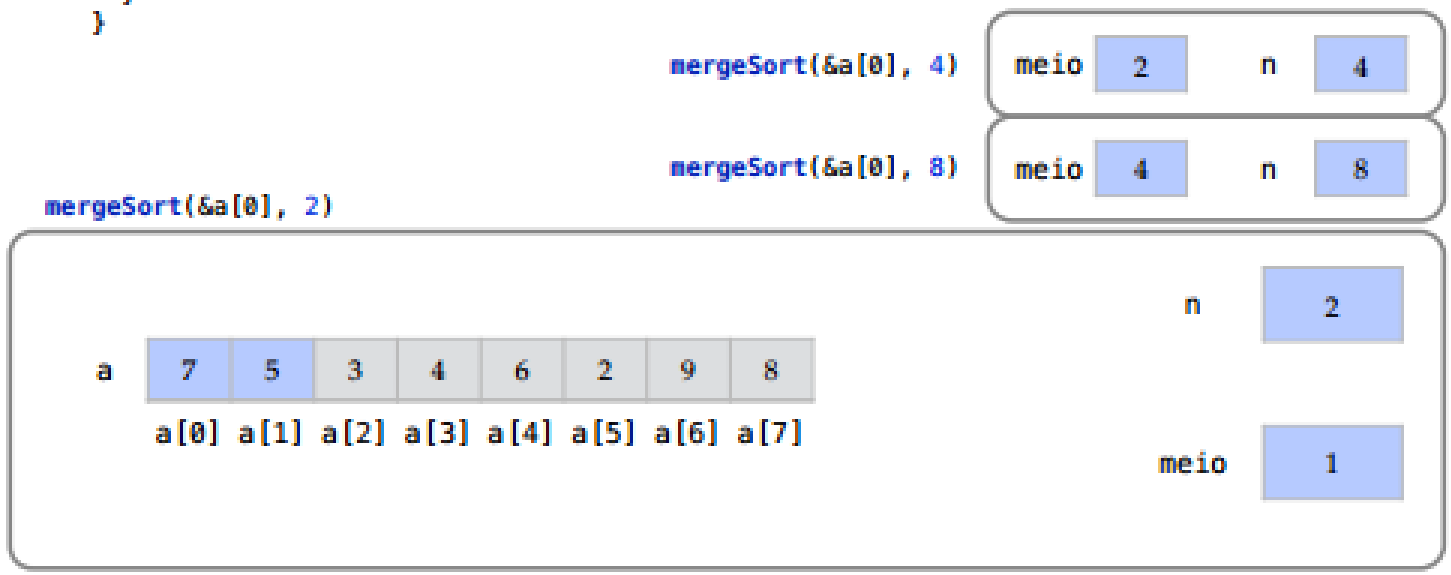
```
void mergeSort(int a[], int n) {
  int meio;
  if (n > 1) {
    meio = n / 2;
    mergeSort(a, meio);
    mergeSort(a + meio, n - meio);
    intercala(a, n);
  }
}
```



MergeSort – Algoritmo de Ordenação

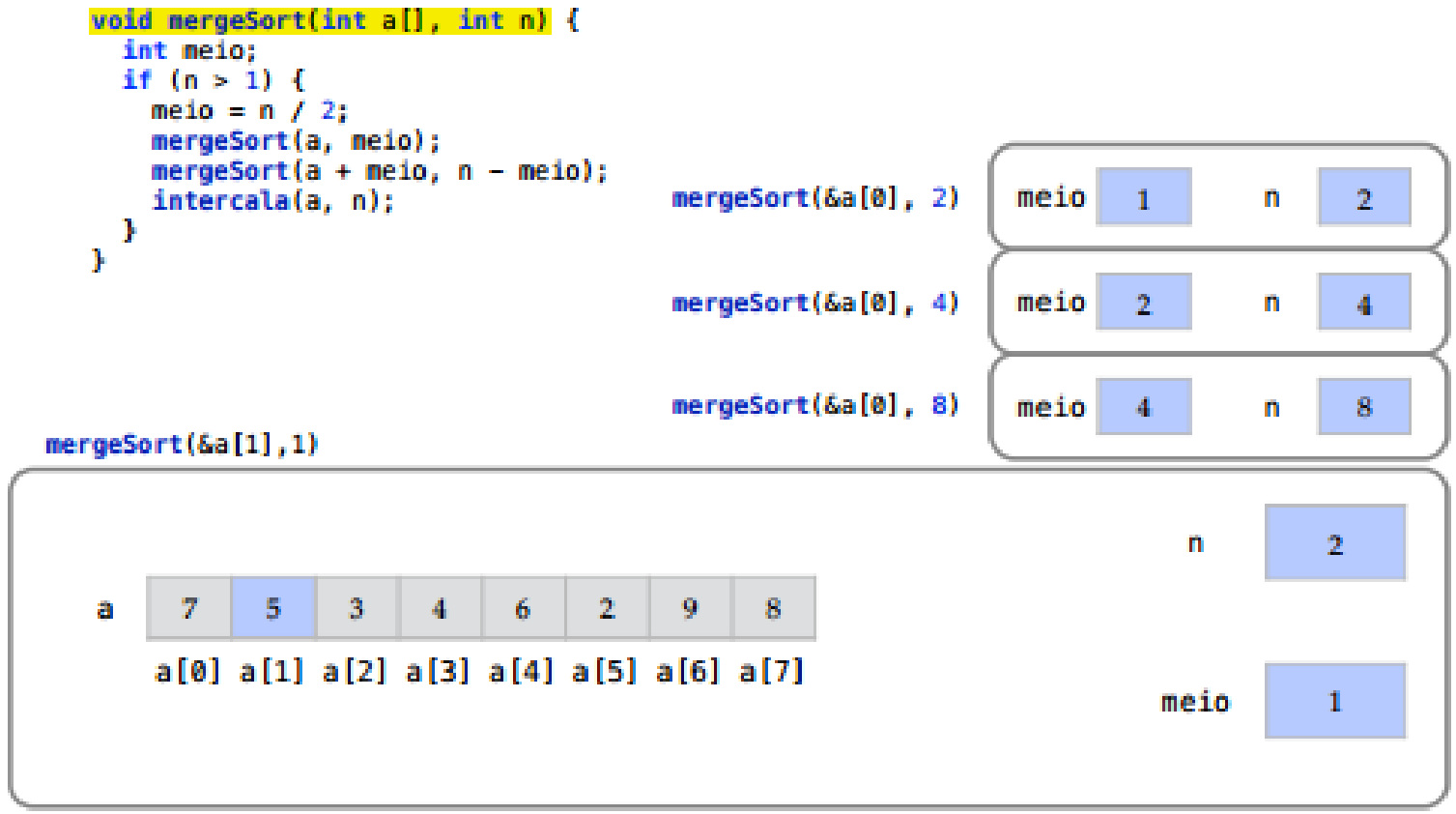
- Ao retornar, esta função mergeSort volta de onde parou. Neste ponto, ele chamará a função para 1 elemento a partir de &a[1]

```
void mergeSort(int a[], int n) {
  int meio;
  if (n > 1) {
    meio = n / 2;
    mergeSort(a, meio);
    mergeSort(a + meio, n - meio);
    intercala(a, n);
  }
}
```



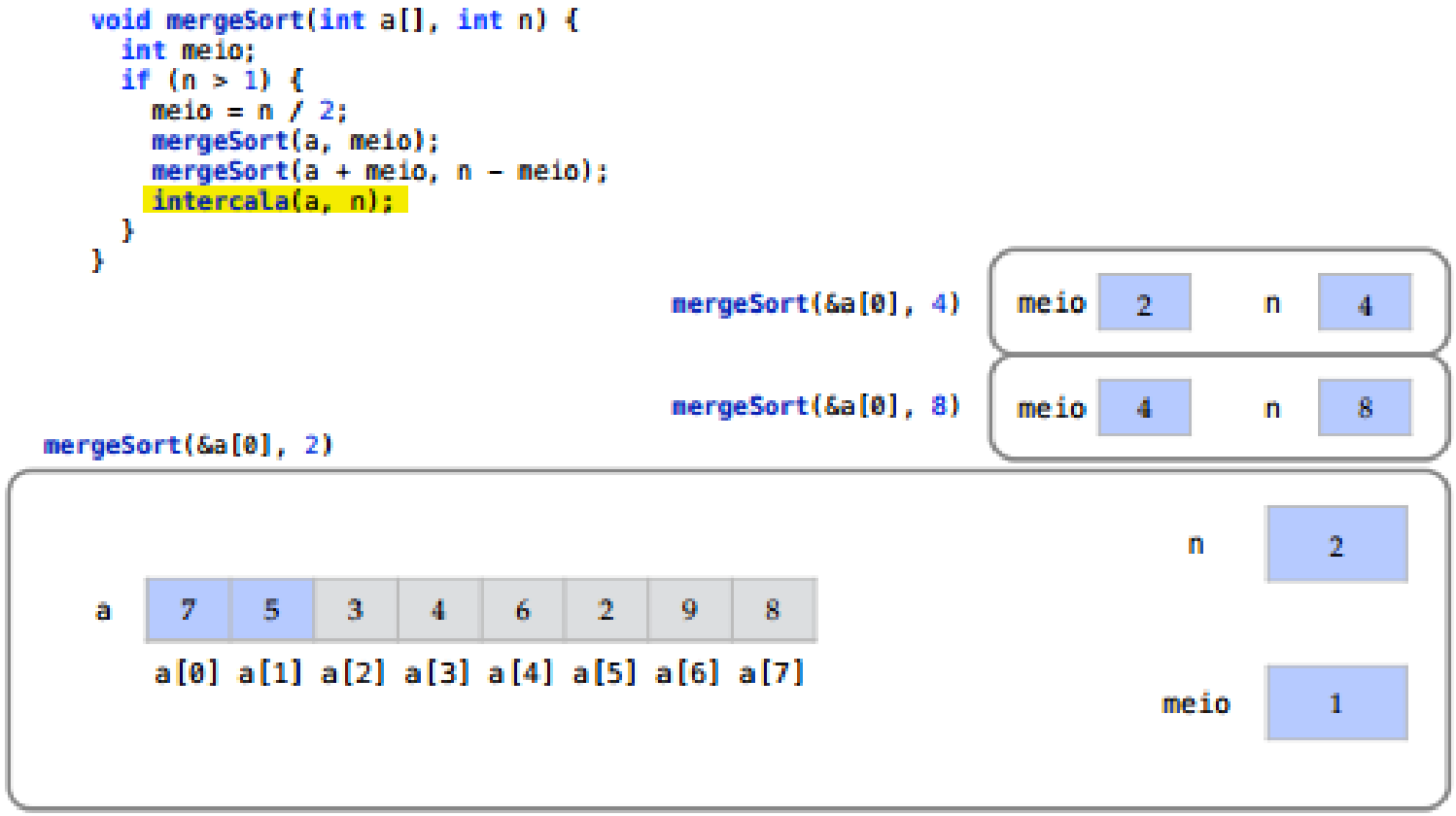
MergeSort – Algoritmo de Ordenação

- Como apenas um elemento está sendo considerado, esta função entrará no caso base e retornará



MergeSort – Algoritmo de Ordenação

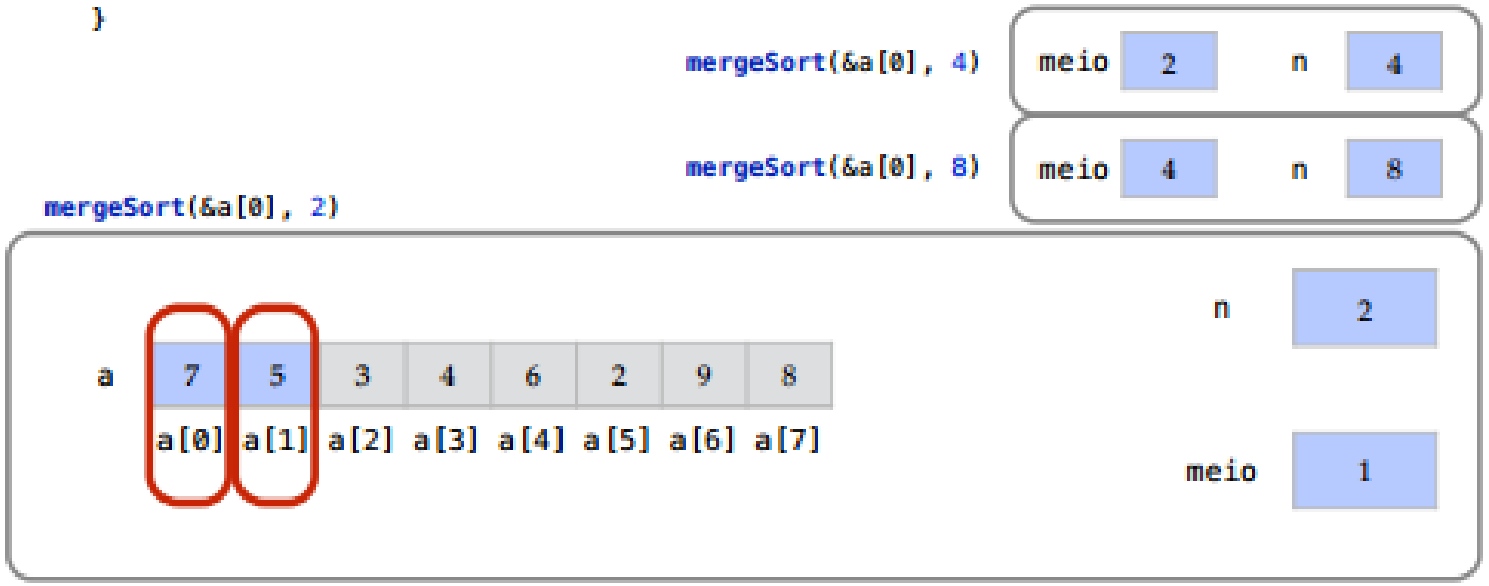
- A função mergeSort do topo da pilha volta então a ser executada e seu próximo passo é de intercalação



MergeSort – Algoritmo de Ordenação

- Como vimos, a função de intercalação intercala duas metades ordenadas de um arranjo

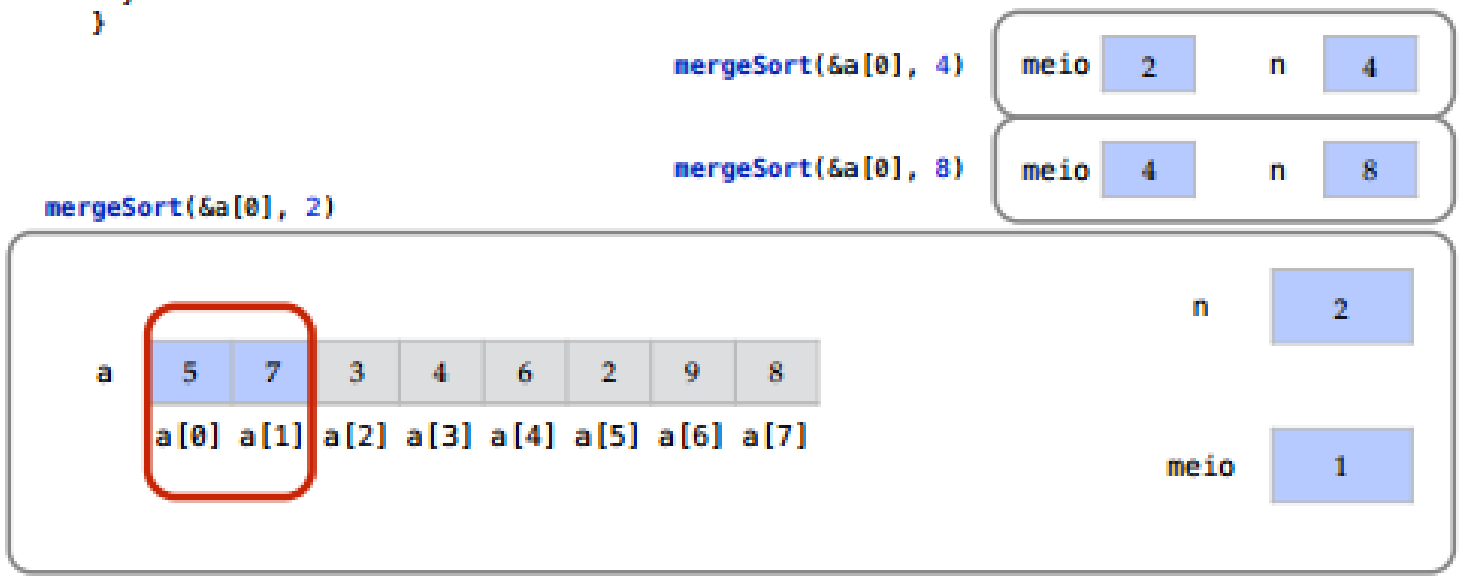
```
void mergeSort(int a[], int n) {
  int meio;
  if (n > 1) {
    meio = n / 2;
    mergeSort(a, meio);
    mergeSort(a + meio, n - meio);
    intercala(a, n);
  }
}
```



MergeSort – Algoritmo de Ordenação

- Com as duas metades intercaladas. Esta execução da função mergeSort termina e a execução do topo da pilha volta a ser executada

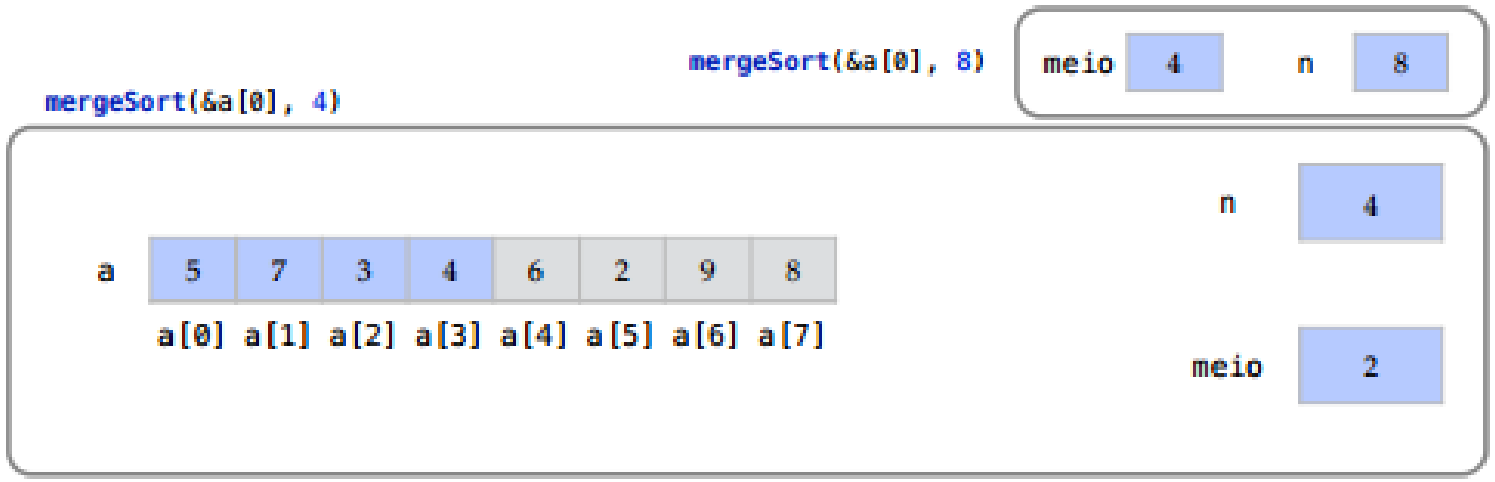
```
void mergeSort(int a[], int n) {
  int meio;
  if (n > 1) {
    meio = n / 2;
    mergeSort(a, meio);
    mergeSort(a + meio, n - meio);
    intercala(a, n);
  }
}
```



MergeSort – Algoritmo de Ordenação

- O próximo passo desta função é usar o próprio mergeSort para ordenar o arranjo que começa em &a[2]e tem 2 elementos

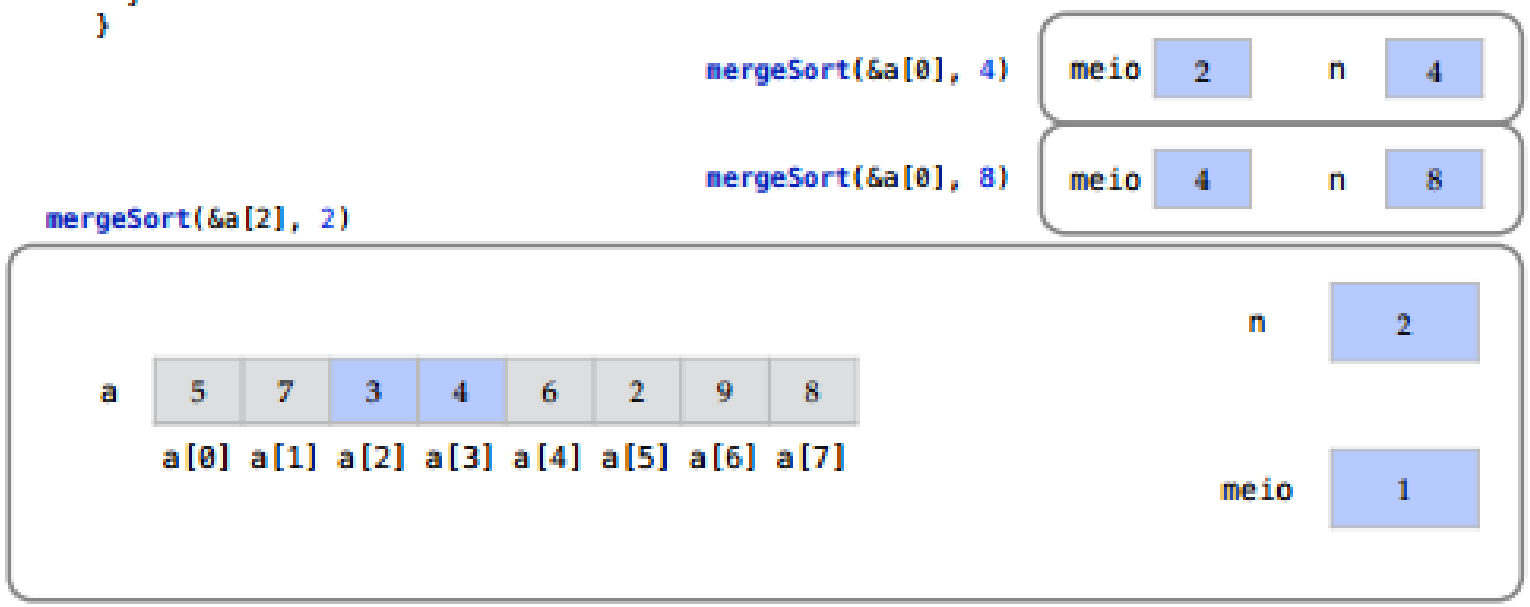
```
void mergeSort(int a[], int n) {
  int meio;
  if (n > 1) {
    meio = n / 2;
    mergeSort(a, meio);
    mergeSort(a + meio, n - meio);
    intercala(a, n);
  }
}
```



MergeSort – Algoritmo de Ordenação

- Esta função precisará dividir o arranjo em problemas menores novamente

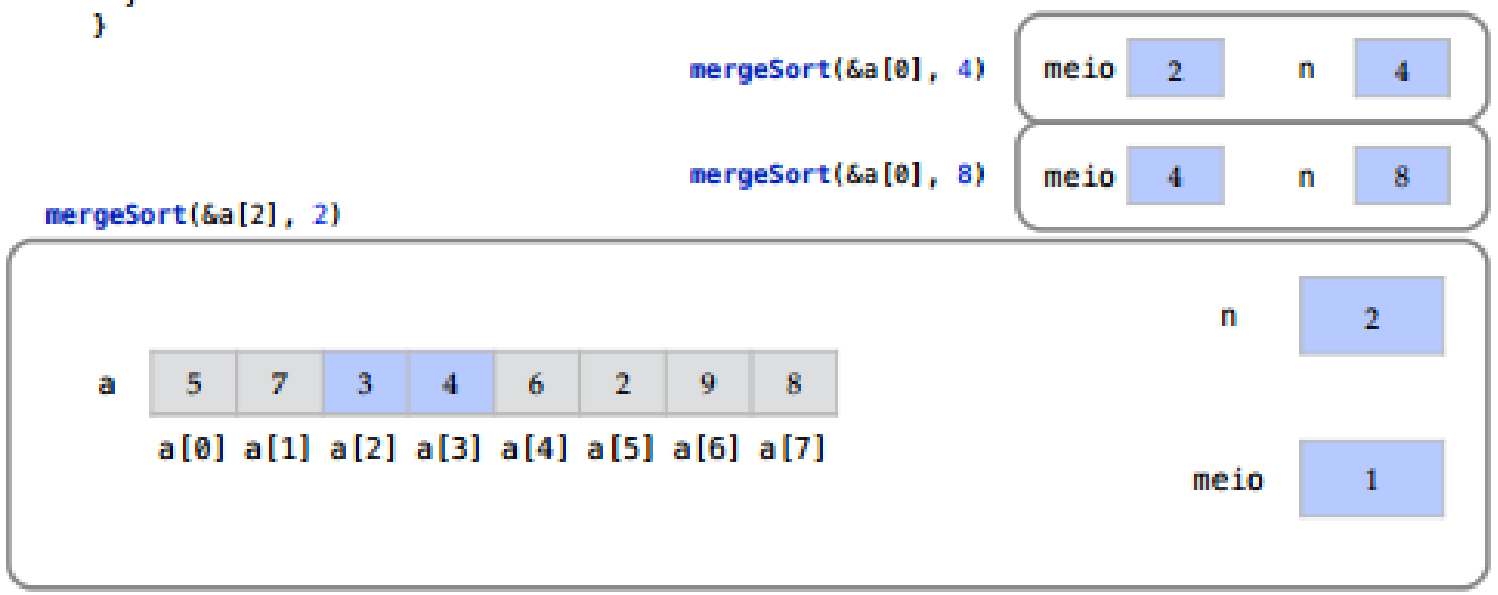
```
void mergeSort(int a[], int n) {
  int meio;
  if (n > 1) {
    meio = n / 2;
    mergeSort(a, meio);
    mergeSort(a + meio, n - meio);
    intercala(a, n);
  }
}
```



MergeSort – Algoritmo de Ordenação

- Esta chamada considerará apenas um arranjo iniciando em &a[2] com um elemento e retornará pelo caso base

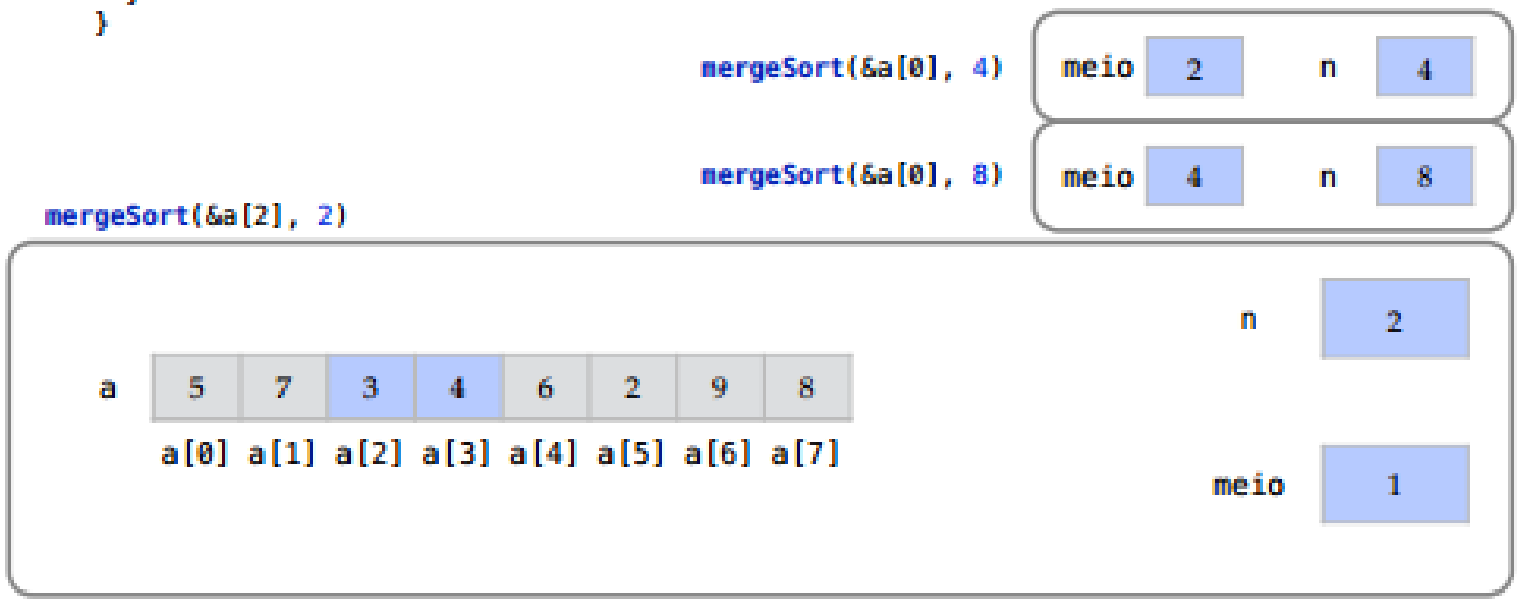
```
void mergeSort(int a[], int n) {
  int meio;
  if (n > 1) {
    meio = n / 2;
    mergeSort(a, meio);
    mergeSort(a + meio, n - meio);
    intercala(a, n);
  }
}
```



MergeSort – Algoritmo de Ordenação

- Esta chamada considerará apenas um arranjo iniciando em &a[3] com um elemento e retornará pelo caso base

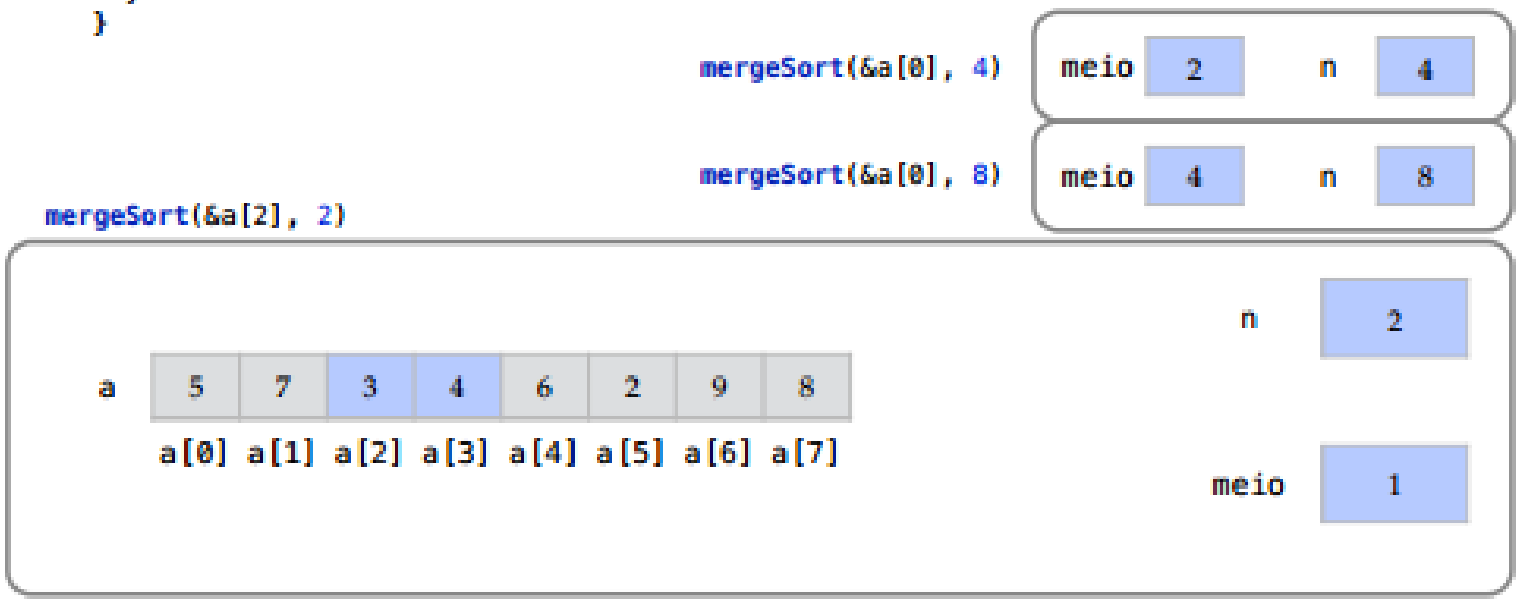
```
void mergeSort(int a[], int n) {
  int meio;
  if (n > 1) {
    meio = n / 2;
    mergeSort(a, meio);
    mergeSort(a + meio, n - meio);
    intercala(a, n);
  }
}
```



MergeSort – Algoritmo de Ordenação

- Esta chamada intercala os elementos do arranjo que se inicia em &a[2] e tem 2 elementos

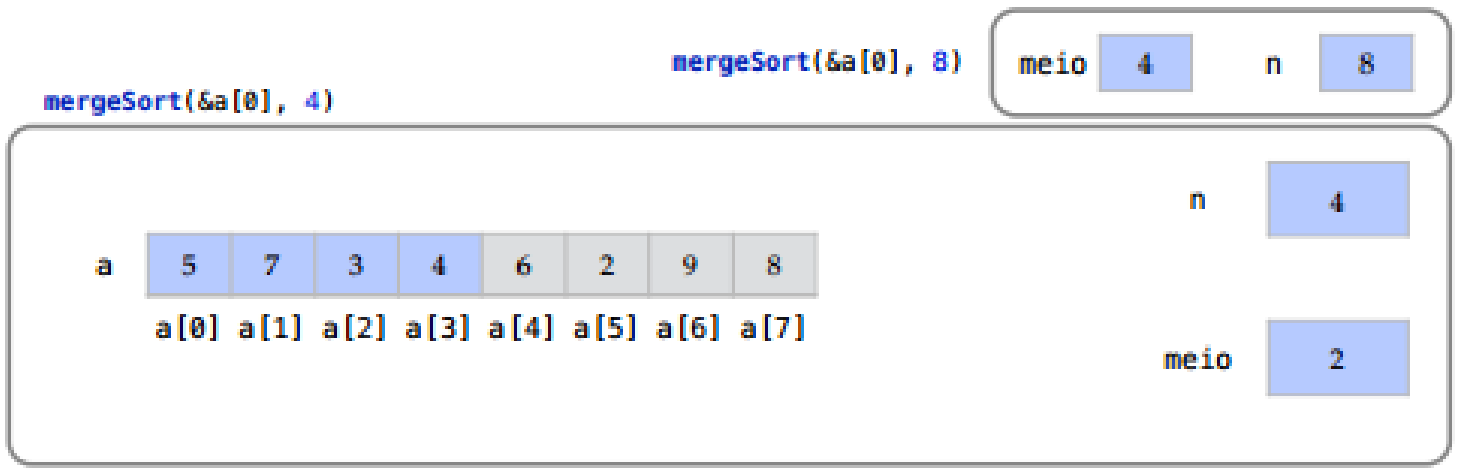
```
void mergeSort(int a[], int n) {
  int meio;
  if (n > 1) {
    meio = n / 2;
    mergeSort(a, meio);
    mergeSort(a + meio, n - meio);
    intercala(a, n);
  }
}
```



MergeSort – Algoritmo de Ordenação

- Com o fim da intercalação, a função do topo da pilha volta a ser executada. A função mergeSort (&a[0], 4) também está na fase de intercalação

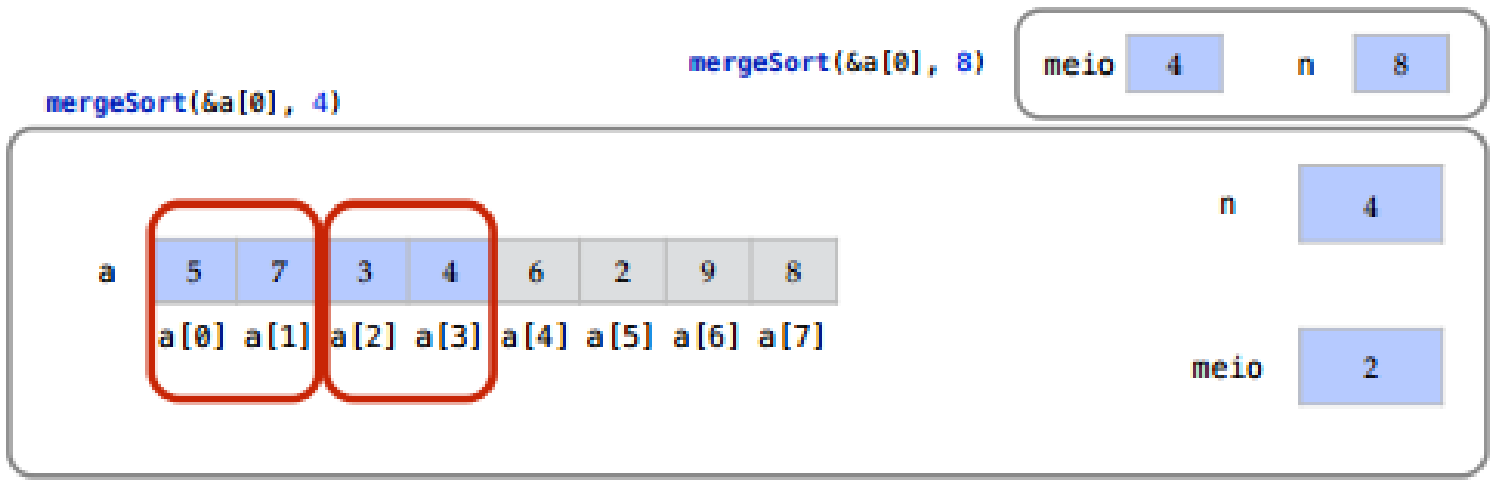
```
void mergeSort(int a[], int n) {
  int meio;
  if (n > 1) {
    meio = n / 2;
    mergeSort(a, meio);
    mergeSort(a + meio, n - meio);
    intercala(a, n);
  }
}
```



MergeSort – Algoritmo de Ordenação

- Como já sabemos, a função de intercalação é aplicada nas duas metades ordenadas do arranjo

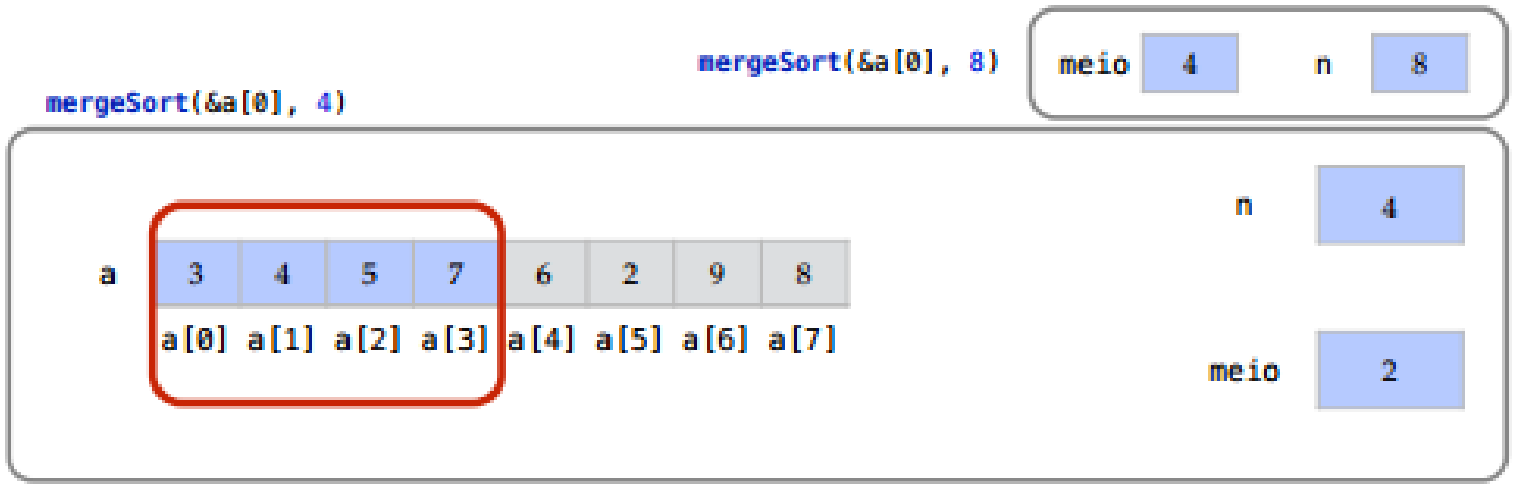
```
void mergeSort(int a[], int n) {
  int meio;
  if (n > 1) {
    meio = n / 2;
    mergeSort(a, meio);
    mergeSort(a + meio, n - meio);
    intercala(a, n);
  }
}
```



MergeSort – Algoritmo de Ordenação

- Estas duas metades são intercaladas em um arranjo com todos os elementos ordenados

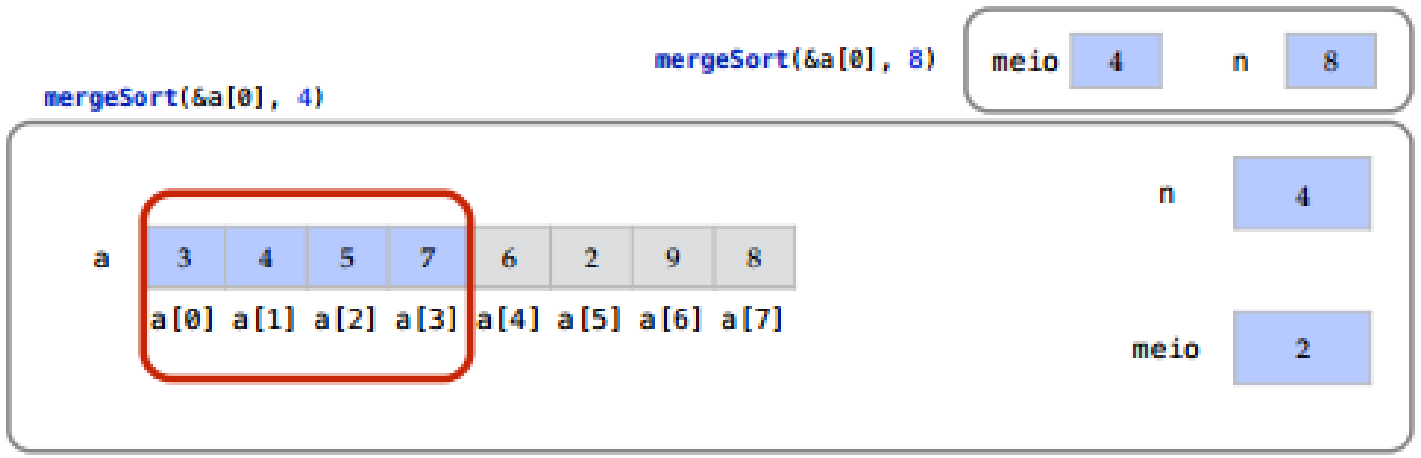
```
void mergeSort(int a[], int n) {
  int meio;
  if (n > 1) {
    meio = n / 2;
    mergeSort(a, meio);
    mergeSort(a + meio, n - meio);
    intercala(a, n);
  }
}
```



MergeSort – Algoritmo de Ordenação

- Com o fim desta intercalação, a primeira metade do arranjo é agora um arranjo ordenado e esta função é retornada, retirando a primeira função da pilha

```
void mergeSort(int a[], int n) {
  int meio;
  if (n > 1) {
    meio = n / 2;
    mergeSort(a, meio);
    mergeSort(a + meio, n - meio);
    intercala(a, n);
  }
}
```

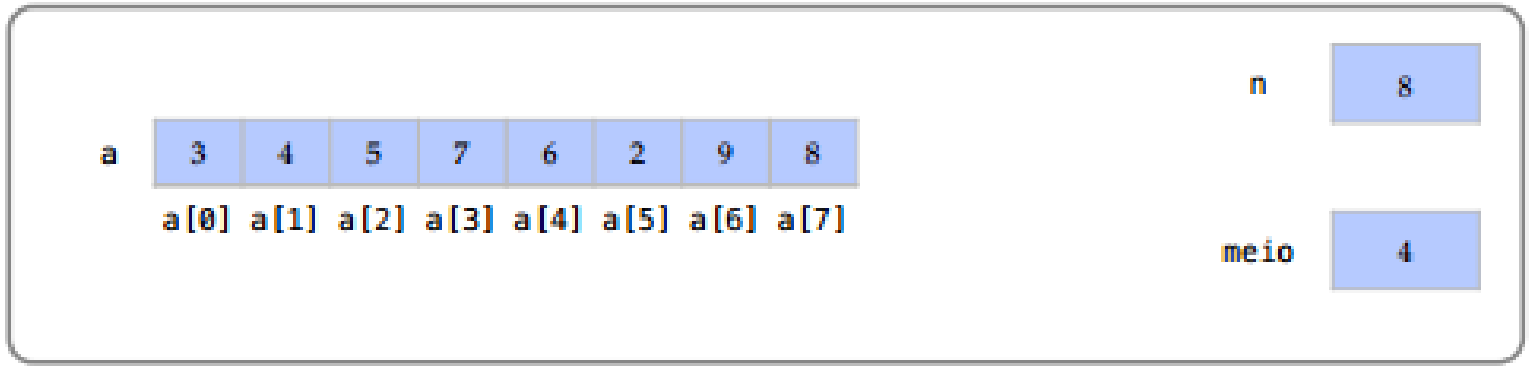


MergeSort – Algoritmo de Ordenação

- A função original era a que considerava todos os elementos do arranjo

```
void mergeSort(int a[], int n) {
  int meio;
  if (n > 1) {
    meio = n / 2;
    mergeSort(a, meio);
    mergeSort(a + meio, n - meio);
    intercala(a, n);
  }
}
```

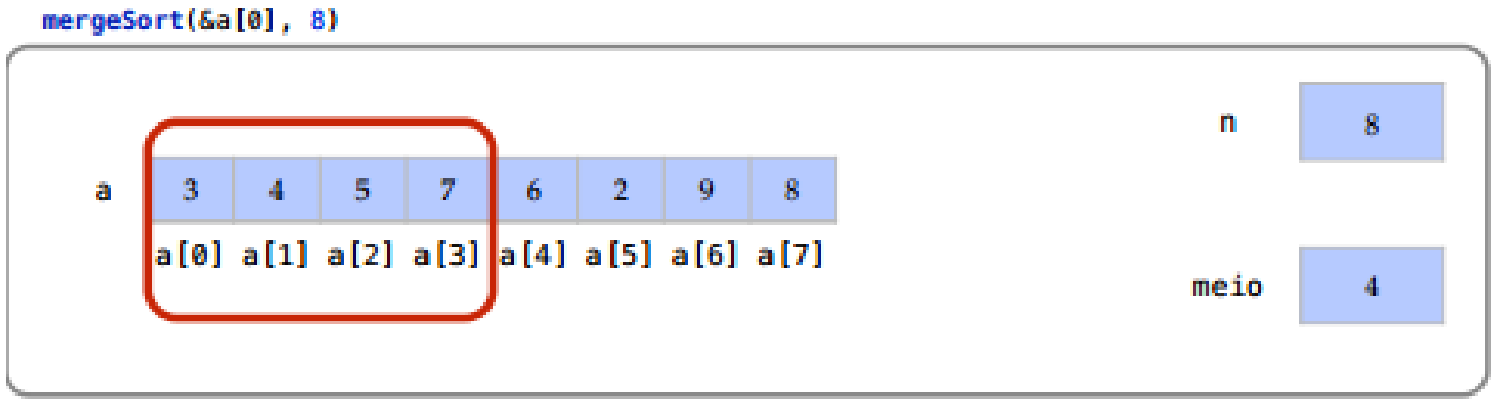
```
mergeSort(&a[0], 8)
```



MergeSort – Algoritmo de Ordenação

- Como vimos, esta função já conseguiu criar um arranjo ordenado dos primeiros elementos do arranjo chamando o método mergeSort recursivamente uma linha acima

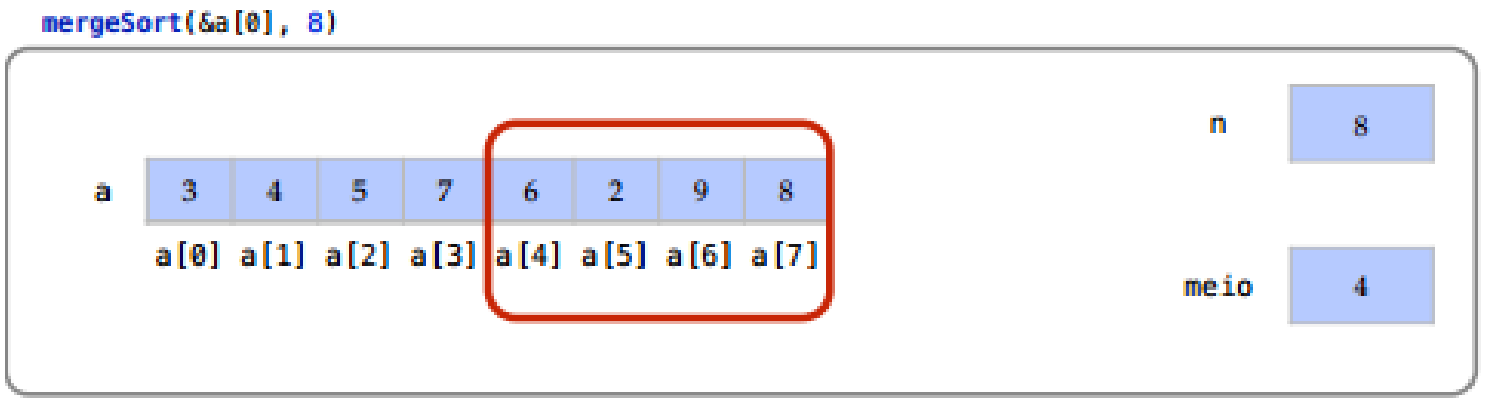
```
void mergeSort(int a[], int n) {
  int meio;
  if (n > 1) {
    meio = n / 2;
    mergeSort(a, meio);
    mergeSort(a + meio, n - meio);
    intercala(a, n);
  }
}
```



MergeSort – Algoritmo de Ordenação

- Nesta chamada do mergeSort, a mesma coisa será feita. Porém, consideraremos o arranjo que se inicia na posição &a[4] e tem 4 elementos

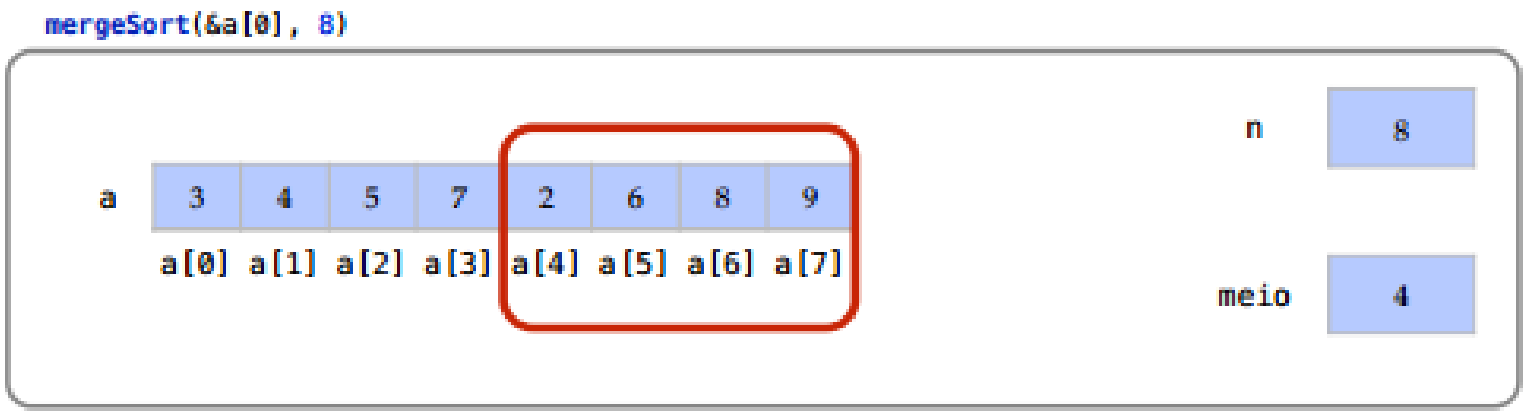
```
void mergeSort(int a[], int n) {
  int meio;
  if (n > 1) {
    meio = n / 2;
    mergeSort(a, meio);
    mergeSort(a + meio, n - meio);
    intercala(a, n);
  }
}
```



MergeSort – Algoritmo de Ordenação

- Como já vimos na linha de código acima, a função mergeSort tem a capacidade de ordenar este arranjo recursivamente

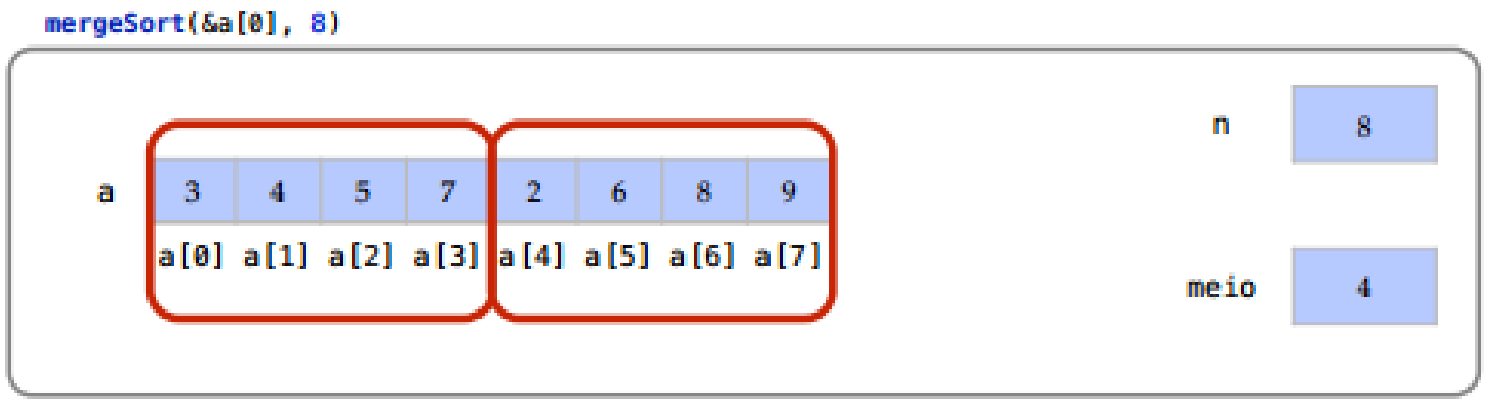
```
void mergeSort(int a[], int n) {
  int meio;
  if (n > 1) {
    meio = n / 2;
    mergeSort(a, meio);
    mergeSort(a + meio, n - meio);
    intercala(a, n);
  }
}
```



MergeSort – Algoritmo de Ordenação

- Já o passo de intercalação desta chamada de função considerará o arranjo que começa em &a[0] e tem 8 elementos

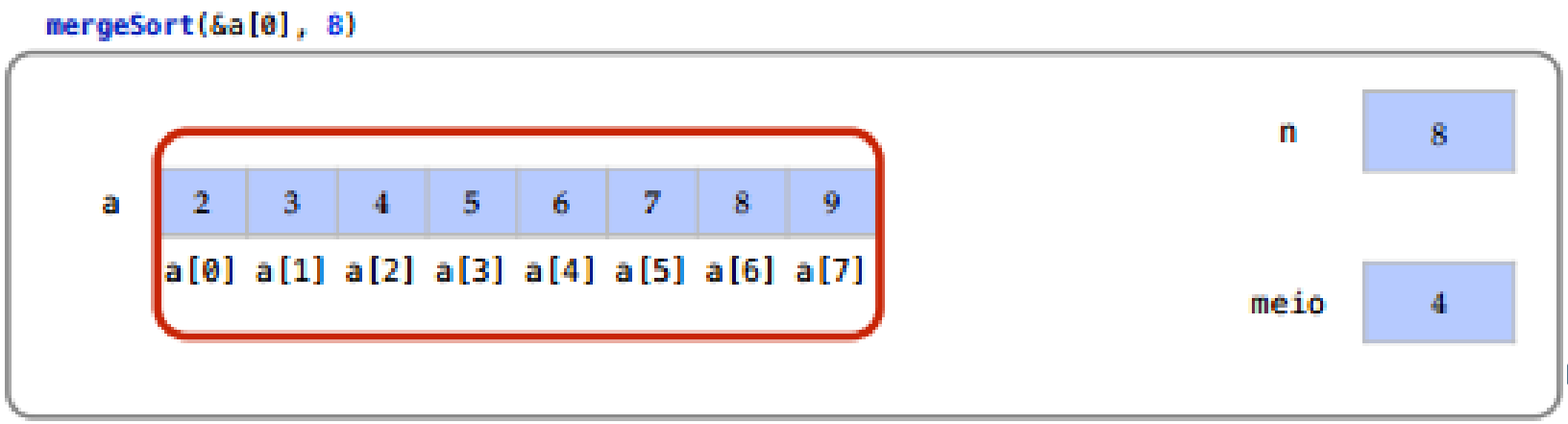
```
void mergeSort(int a[], int n) {
  int meio;
  if (n > 1) {
    meio = n / 2;
    mergeSort(a, meio);
    mergeSort(a + meio, n - meio);
    intercala(a, n);
  }
}
```



MergeSort – Algoritmo de Ordenação

- A função intercala então ordena as duas metades ordenadas deste arranjo

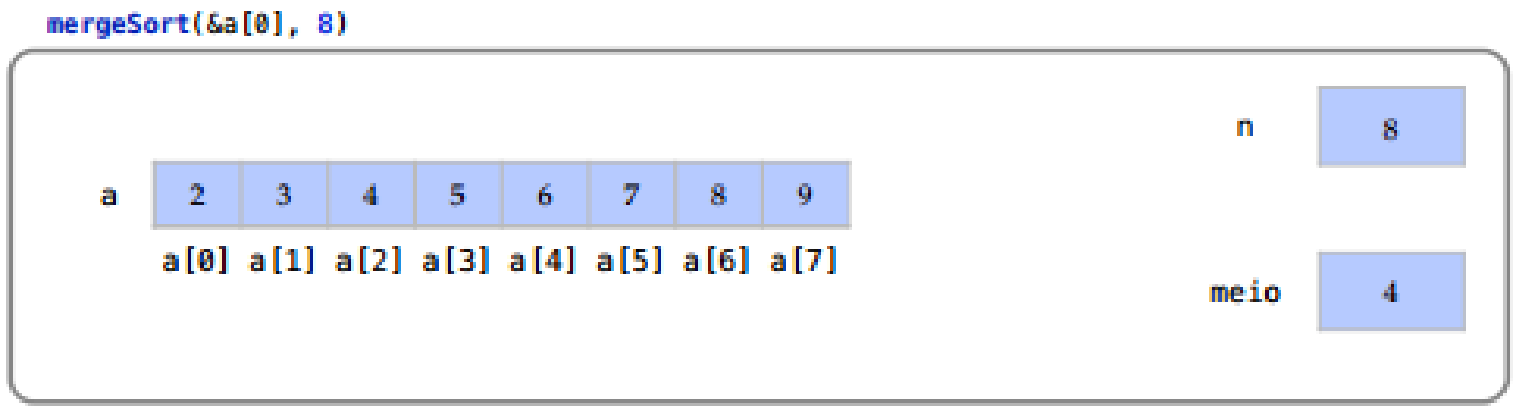
```
void mergeSort(int a[], int n) {
  int meio;
  if (n > 1) {
    meio = n / 2;
    mergeSort(a, meio);
    mergeSort(a + meio, n - meio);
    intercala(a, n);
  }
}
```



MergeSort – Algoritmo de Ordenação

- Com isto, temos nosso arranjo inicial ordenado e podemos retornar da função

```
void mergeSort(int a[], int n) {
  int meio;
  if (n > 1) {
    meio = n / 2;
    mergeSort(a, meio);
    mergeSort(a + meio, n - meio);
    intercala(a, n);
  }
}
```



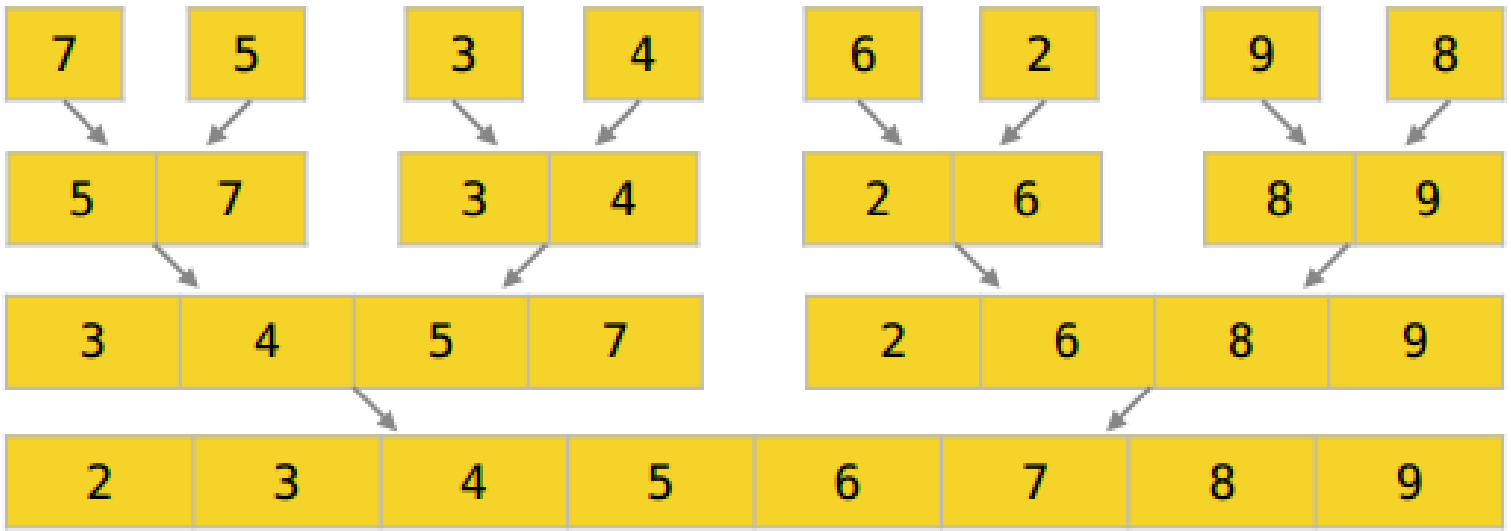
MergeSort – Análise

- Uma peça fundamental do algoritmo é a intercalação de dois arranjos

- Esta intercalação tem um custo O(n) pois passamos 1 vez por cada elemento o colocando no arranjo temporário

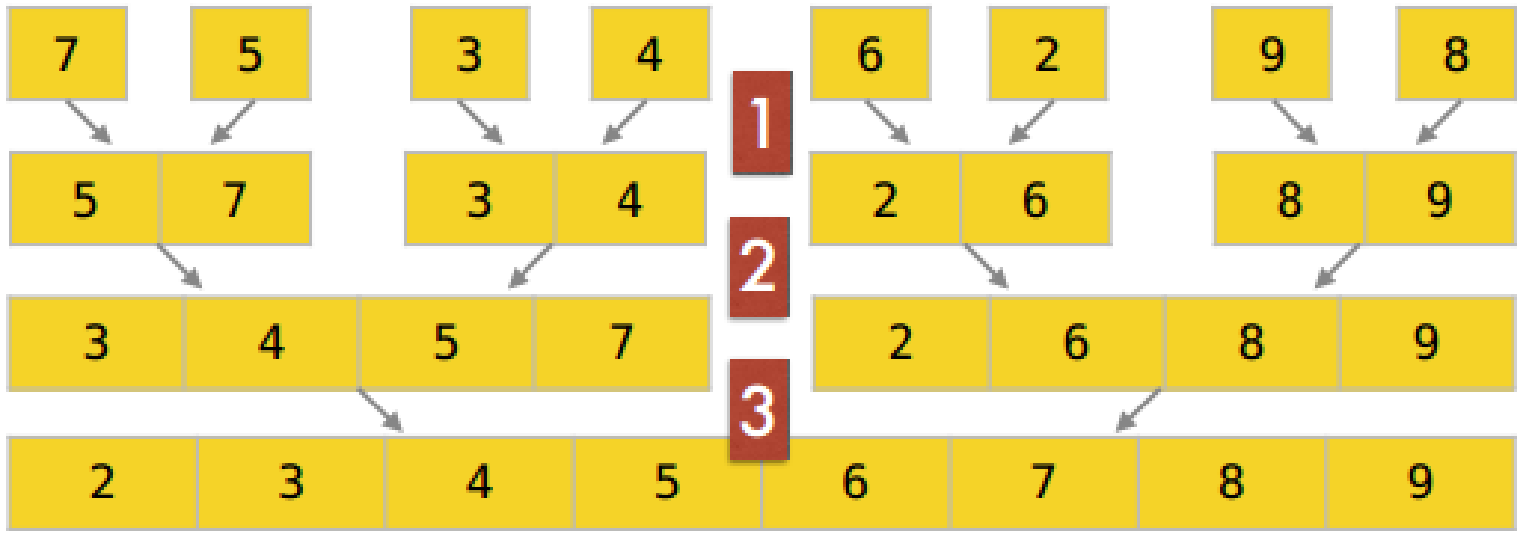
MergeSort – Análise

- O custo do algoritmo depende então do número de intercalações feitas
  - Para isto, analise a seguinte intercalação:



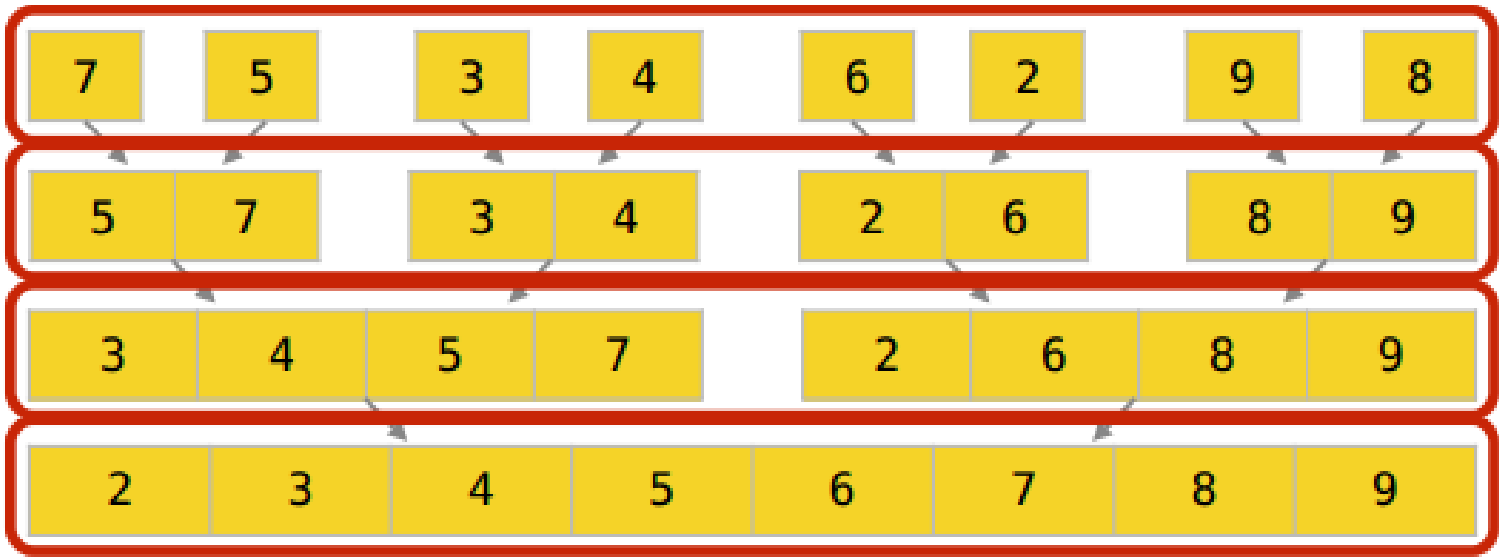
MergeSort – Análise

- Neste exemplo foram feitos 3 passos de intercalação



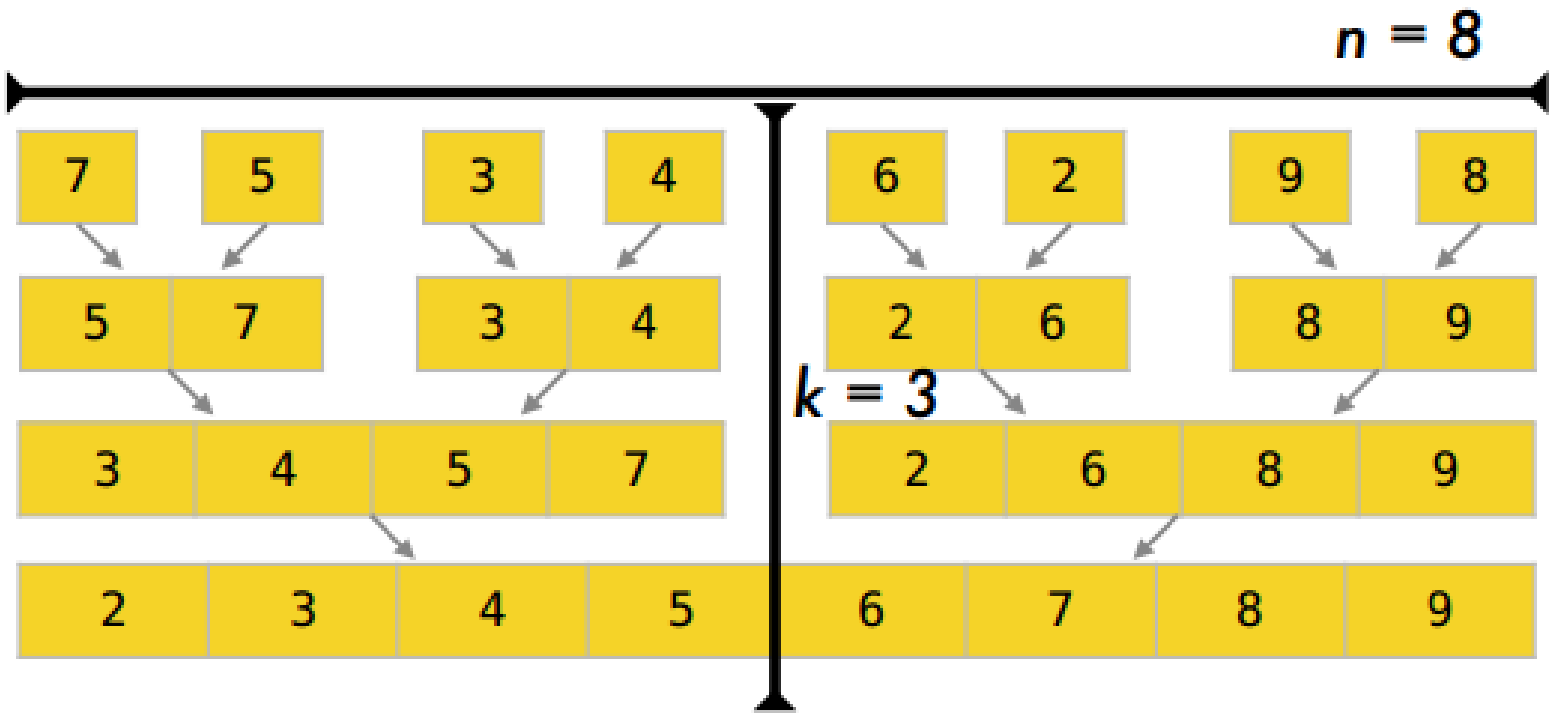
MergeSort – Análise

- A cada um destes passos, 8 elementos foram intercalados



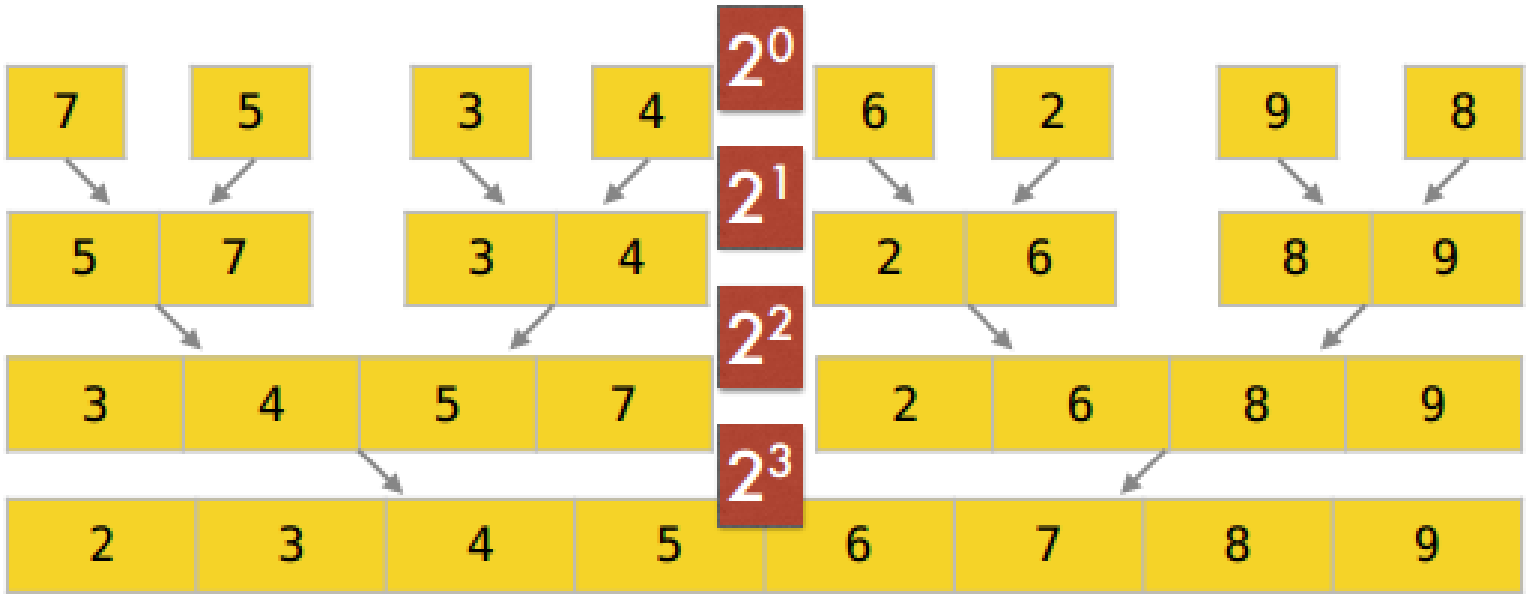
MergeSort – Análise

- Se temos n elementos, cada intercalação com custo O(n) e k passos, o custo total do algoritmo é O(nk). Nos resta saber o valor de k



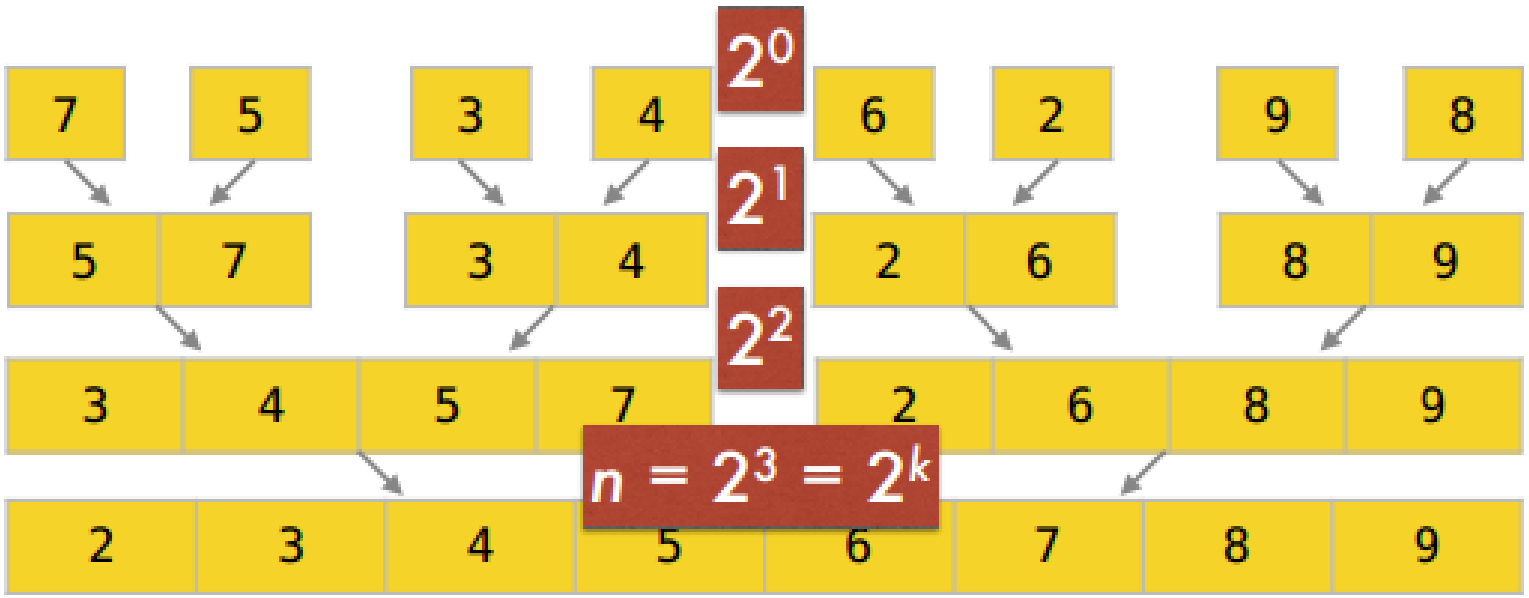
MergeSort – Análise

- A cada passo, dobramos o tamanho de cada subarranjo, ou seja o tamanho dos subarranjos depois de k passos é 2^k



MergeSort – Análise

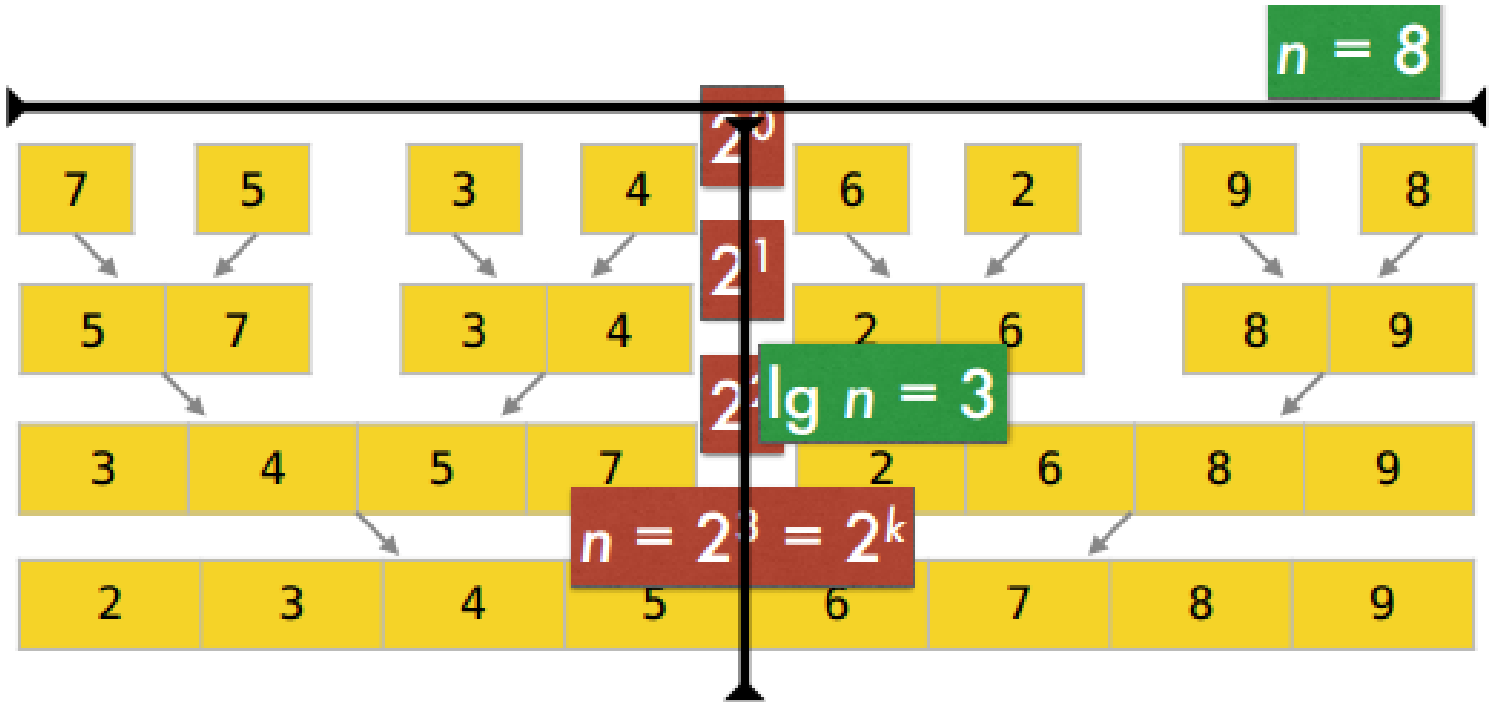
- O algoritmo termina quando o tamanho do arranjo intercalado seja $2^k = n$



MergeSort – Análise

$2^k = n \rightarrow k = \log n$

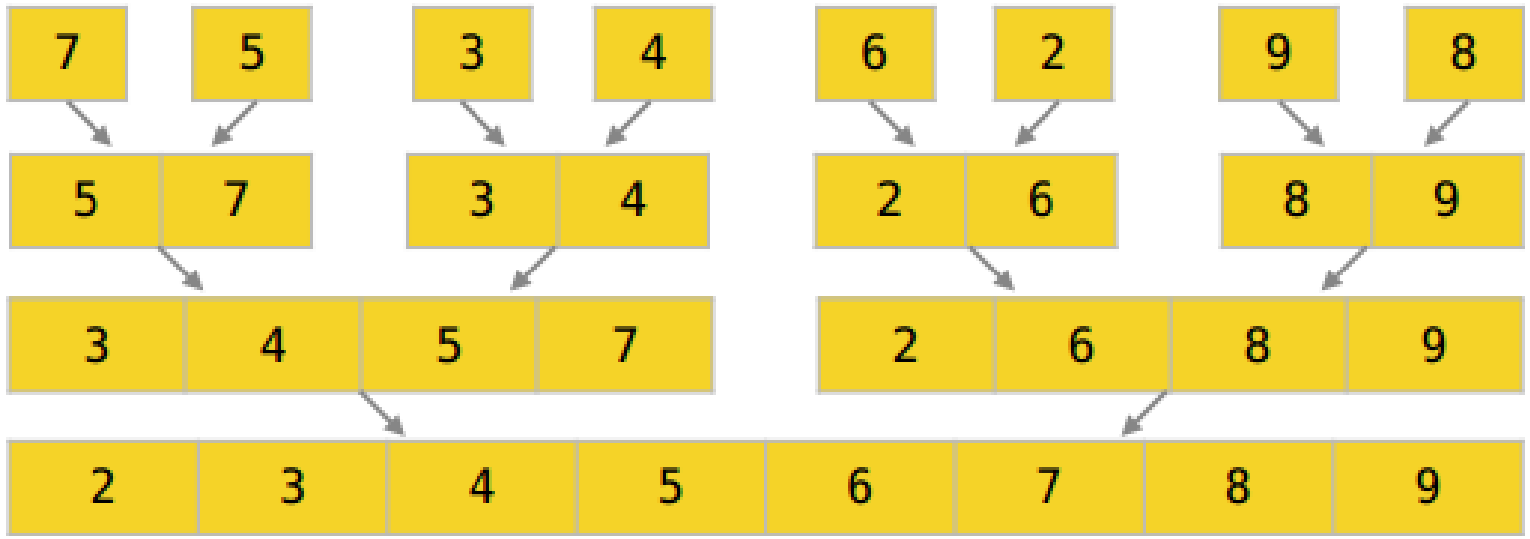
Deste modo, temos que o número de passos k para encerrar o algoritmo é log n



MergeSort – Análise

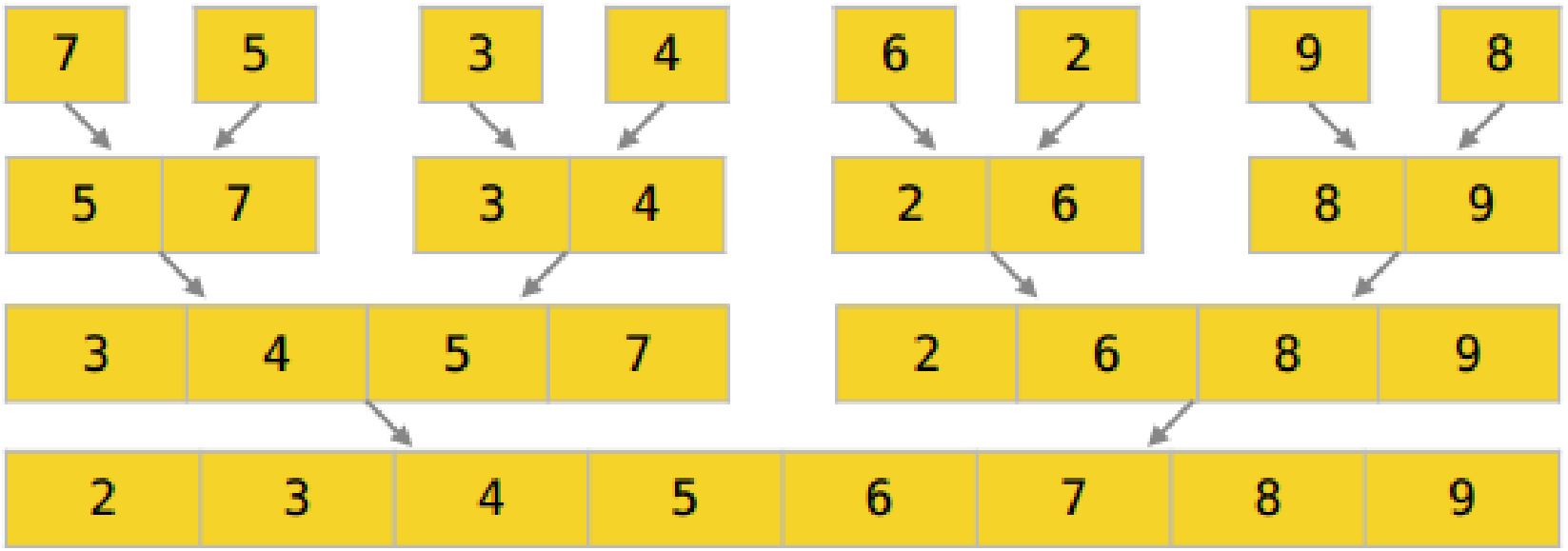
- Sendo k = log n, o custo total do algoritmo é então

O(nk) = O(n log n)



MergeSort – Análise

- Como vimos anteriormente, algoritmos típicos da classe O(n log n) são os que quebram o problema em problemas menores, fazem uma operação em cada um dos elementos e depois combinam as soluções



MergeSort – Análise

- Assim, a complexidade do método é O(n log n)

- Uma vantagem do método é sua complexidade constante para o pior caso e melhor caso

MergeSort – Análise

- A maior desvantagem é a necessidade de memória extra na fase de intercalação e nas chamadas recursivas da função

- Assim, este é o algoritmo a ser usado quando queremos uma ordem de complexidade baixa, constante, mas memória não seja um problema

Exercício

- Criar um arranjo com 10 elementos aleatórios desordenados entre 1 e 50
- Desenhar o arranjo várias vezes demonstrando os passos de uma ordenação com o método:
  - MergeSort (intercalar elementos a cada passo)
- Colocar um círculo nos elementos movimentados. Colocar um traço entre os elementos ordenados e desordenados

Algoritmos e Estruturas de Dados II

- Bibliografia:
  - Básica:
    - CORMEN, Thomas, RIVEST, Ronald, STEIN, Clifford, LEISERSON, Charles. Algoritmos. Rio de Janeiro: Elsevier, 2002.
    - EDELWEISS, Nina, GALANTE, Renata. Estruturas de dados. Porto Alegre: Bookman. 2009. (Série livros didáticos informática UFRGS,18).
    - ZIVIANI, Nívio. Projeto de algoritmos com implementação em Pascal e C. São Paulo: Cengage Learning, 2010.
  - Complementar:
    - ASCENCIO, Ana C. G. Estrutura de dados. São Paulo: Pearson, 2011. ISBN: 9788576058816.
    - PINTO, W.S. Introdução ao desenvolvimento de algoritmos e estrutura de dados. São Paulo: Érica, 1990.
    - PREISS, Bruno. Estruturas de dados e algoritmos. Rio de Janeiro: Campus, 2000.
    - TENEMBAUM. Aaron M. Estruturas de dados usando C. São Paulo: Makron Books. 1995. 884 p. ISBN: 8534603480.
    - VELOSO, Paulo A. S. Complexidade de algoritmos: análise, projeto e métodos. Porto Alegre, RS: Sagra Luzzatto, 2001

Algoritmos e Estruturas de Dados II