



Ordenação: MergeSort

Prof. Túlio Toffolo

<http://www.toffolo.com.br>

BCC202 – Aula 14

Algoritmos e Estruturas de Dados I

DIVISÃO E CONQUISTA

- É preciso revolver um problema com uma entrada grande
- Para facilitar a resolução do problema, a entrada é quebrada em pedaços menores (**DIVISÃO**)
- Cada pedaço da entrada é então tratado separadamente (**CONQUISTA**)
- Ao final, os resultados parciais são combinados para gerar o resultado final procurado

A técnica de Divisão e Conquista

A técnica de divisão e conquista consiste de 3 passos:

- **Divisão**: Dividir o problema original em subproblemas menores
- **Conquista**: Resolver cada subproblema recursivamente
- **Combinação**: Combinar as soluções encontradas, compondo uma solução para o problema original

A técnica de Divisão e Conquista

- Algoritmos baseados em divisão e conquista são, em geral, **recursivos**.
- A maioria dos algoritmos de divisão e conquista divide o problema em subproblemas da mesma natureza, de tamanho **n/b** .
- Vantagens:
 - Requerem um número menor de acessos à memória.
 - São altamente paralelizáveis. Se existirem vários processadores disponíveis, a estratégia propiciará eficiência.

Quando utilizar?

- Existem três condições que indicam que a estratégia de divisão e conquista pode ser utilizada com sucesso:
 - Deve ser possível decompor uma instância em sub-instâncias
 - A combinação dos resultados dever ser eficiente (trivial se possível)
 - As sub-instâncias devem ser mais ou menos do mesmo tamanho

Algoritmo Genérico

```
def divisao_e_conquista(x):  
    if x é pequeno ou simples:  
        return resolve(x)  
    else:  
        decompor x em n conjuntos menores  $x_0, x_1, \dots, x_{n-1}$   
        for i in  $[0, 1, \dots, n-1]$ :  
             $y_i = \text{divisao\_e\_conquista}(x_i)$   
        combinar  $y_0, y_1, \dots, y_{n-1}$  em y  
        return y
```

ORDENAÇÃO POR INTERCALAÇÃO

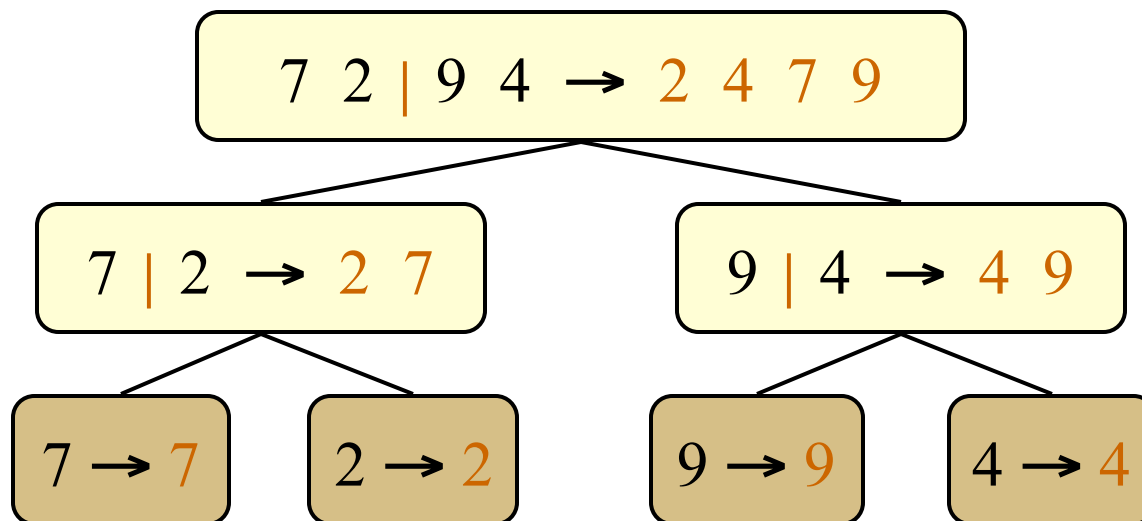
MERGESORT

Abordagem com Balanceamento

- Métodos de ordenação que fazem divisão e conquista
 - QuickSort (pior caso?)
 - MergeSort
- Principal diferença:
 - QuickSort utiliza o conceito de elemento pivô para dividir o problema em subproblemas
 - MergeSort sempre divide o problema de forma balanceada (gerando subproblemas de mesmo tamanho)

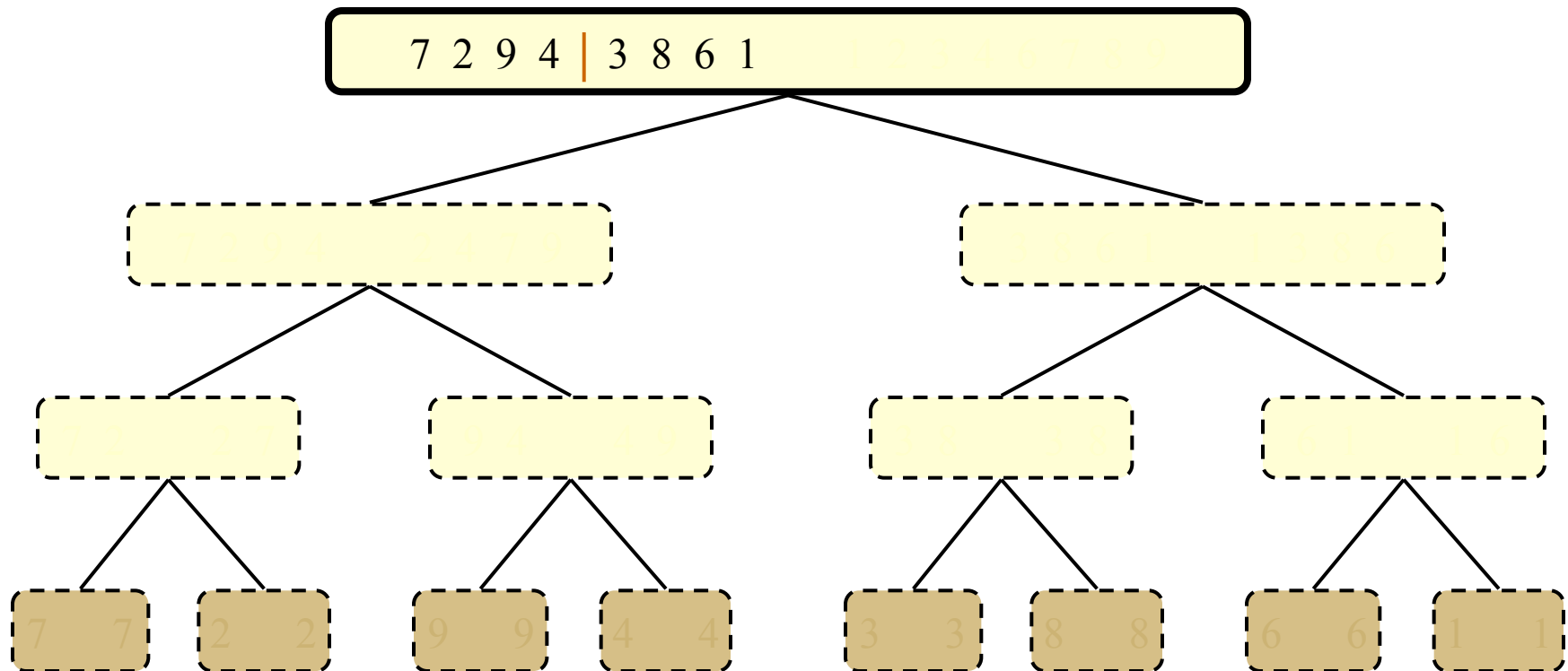
MergeSort: Execução

- A execução do MergeSort pode ser facilmente descrita por uma árvore binária
 - Cada nó representa uma chamada recursiva do MergeSort
 - O nó raiz é a chamada inicial
 - Os nós folhas são vetores de 1 ou 2 números (casos bases)



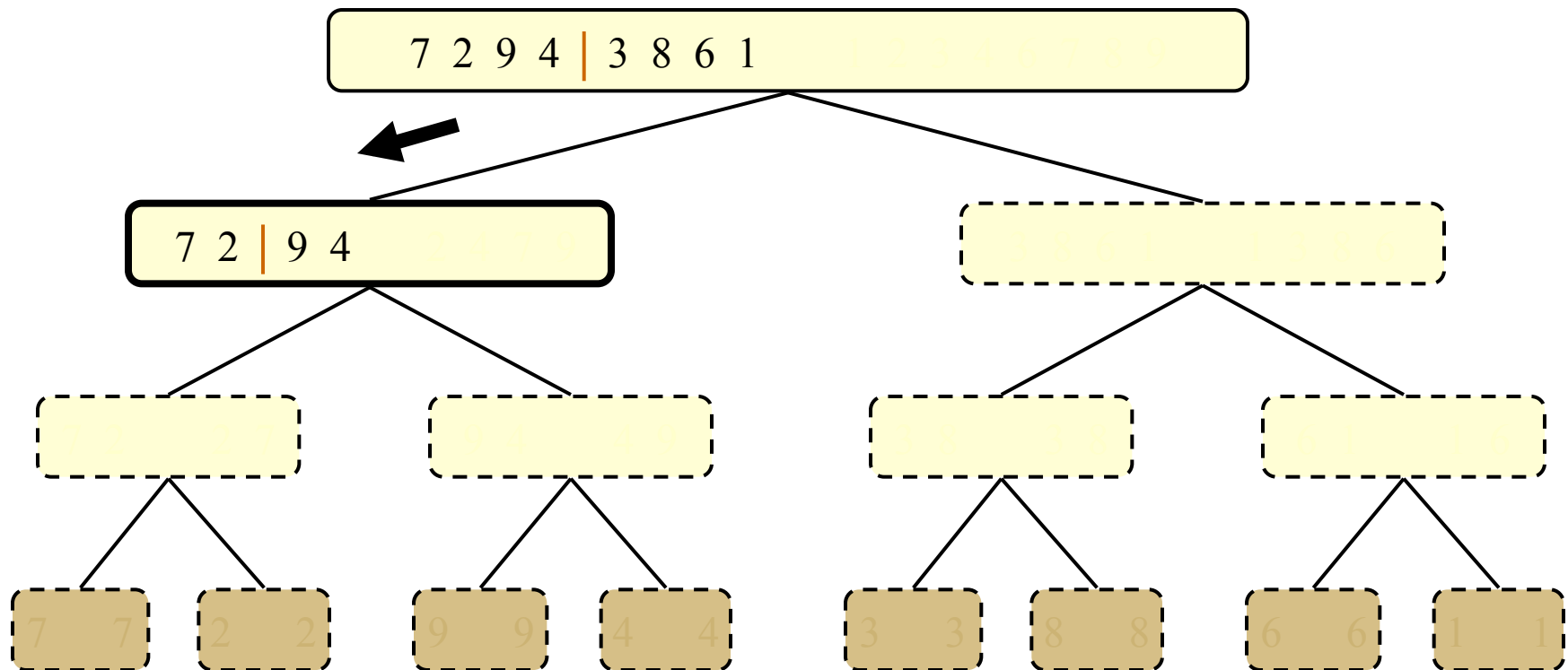
MergeSort: Exemplo de Execução

- Partição do problema (sempre no meio do vetor)



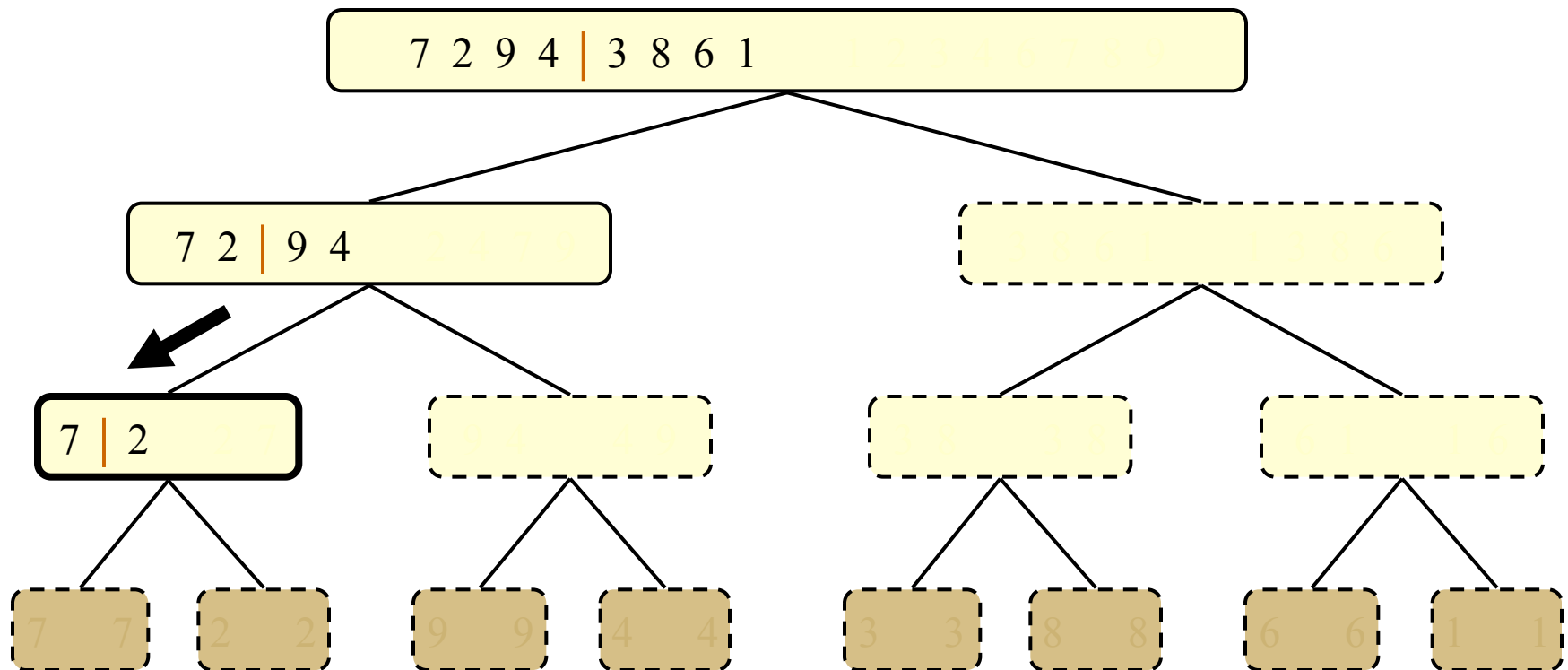
MergeSort: Exemplo de Execução (cont.)

- Chamada recursiva para primeira partição



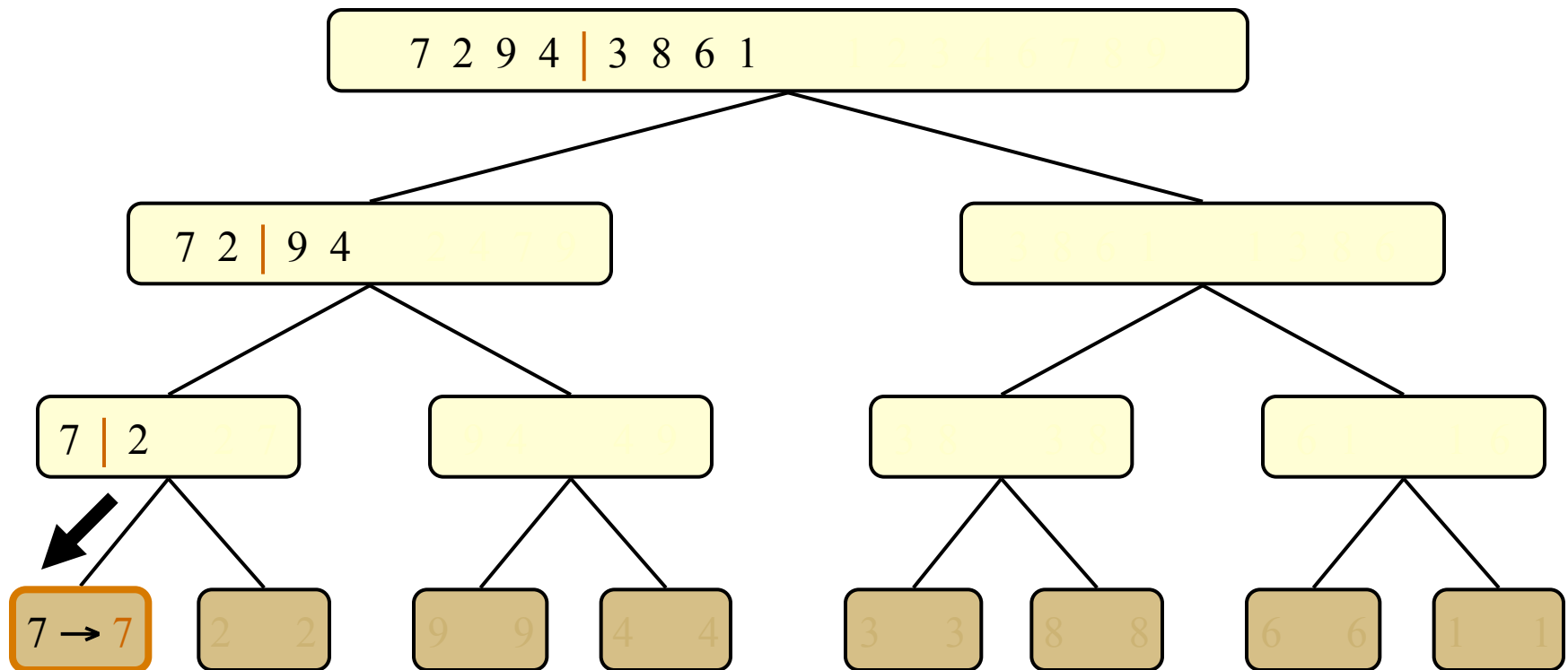
MergeSort: Exemplo de Execução (cont.)

- Chamada recursiva



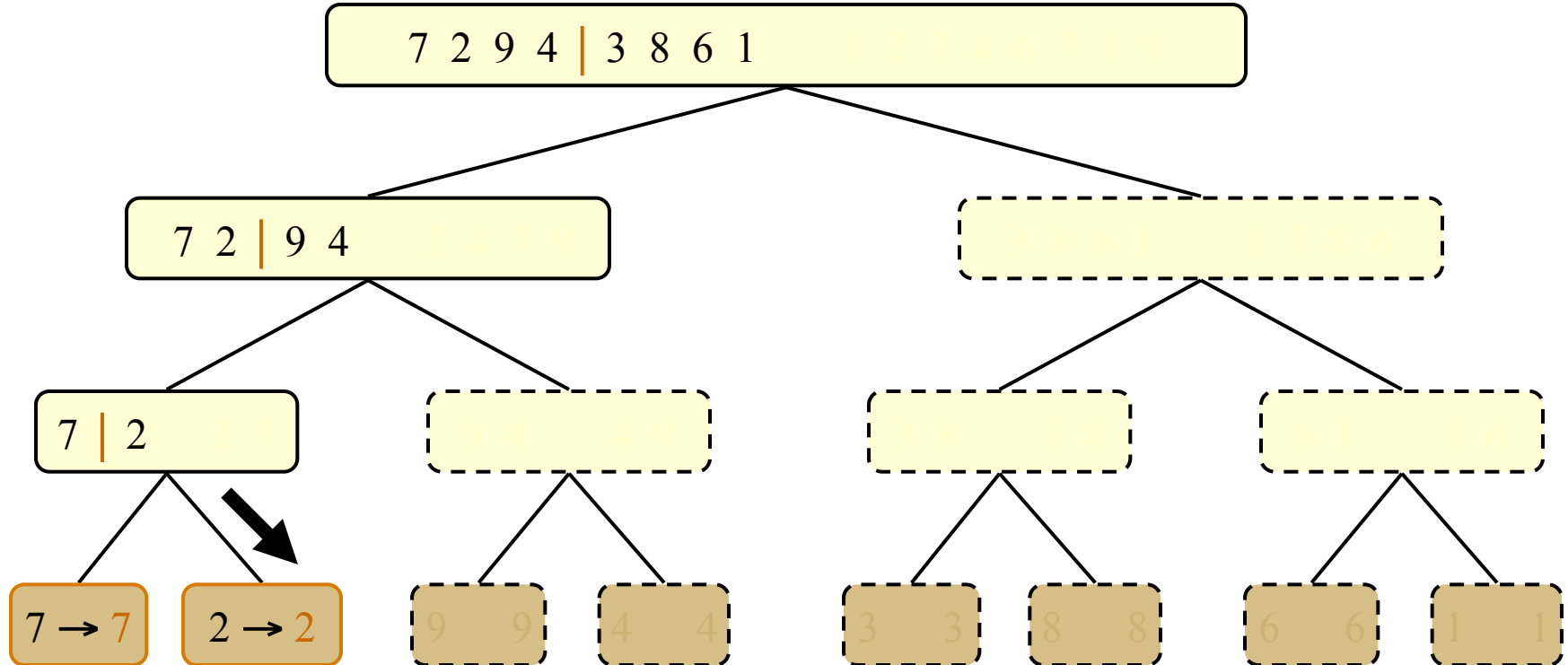
MergeSort: Exemplo de Execução (cont.)

- Chamada recursiva: caso base encontrado



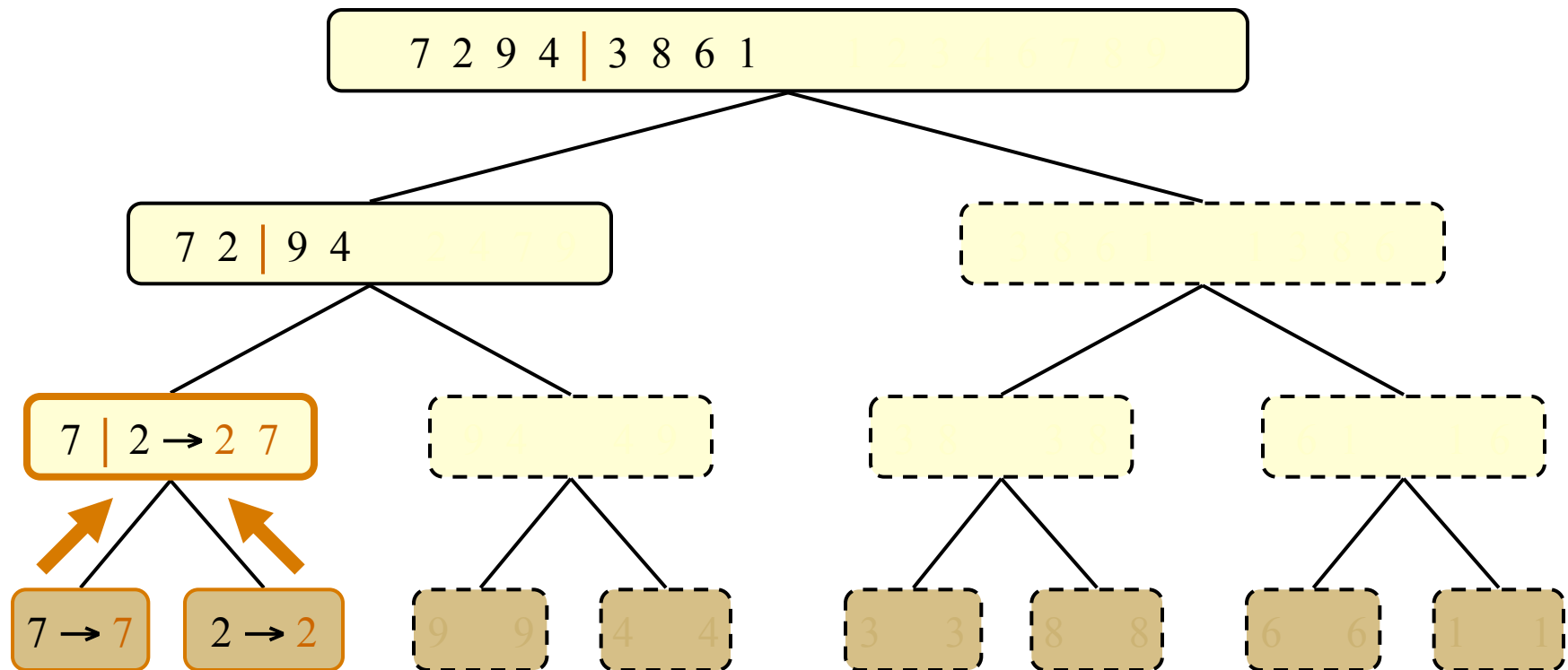
MergeSort: Exemplo de Execução (cont.)

- Chamada recursiva: caso base encontrado



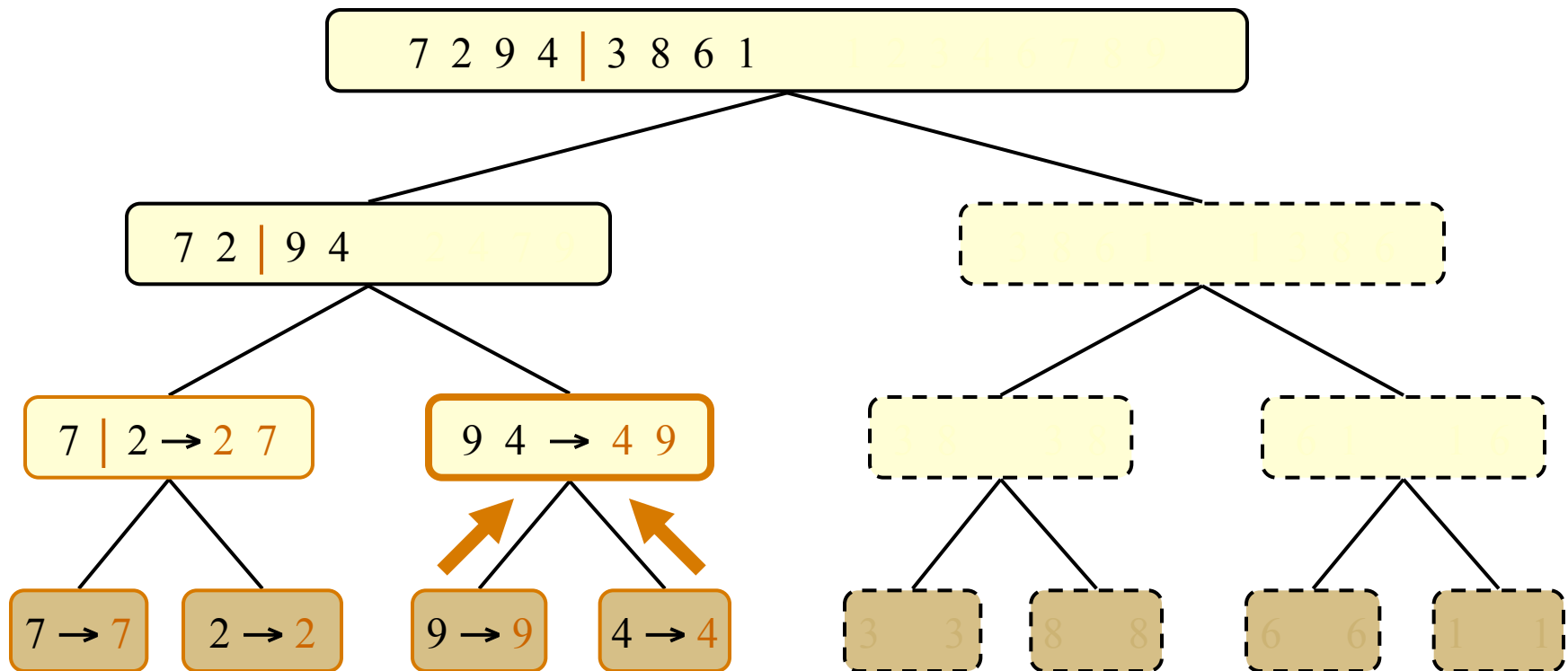
MergeSort: Exemplo de Execução (cont.)

- Operação de merge (intercalação)



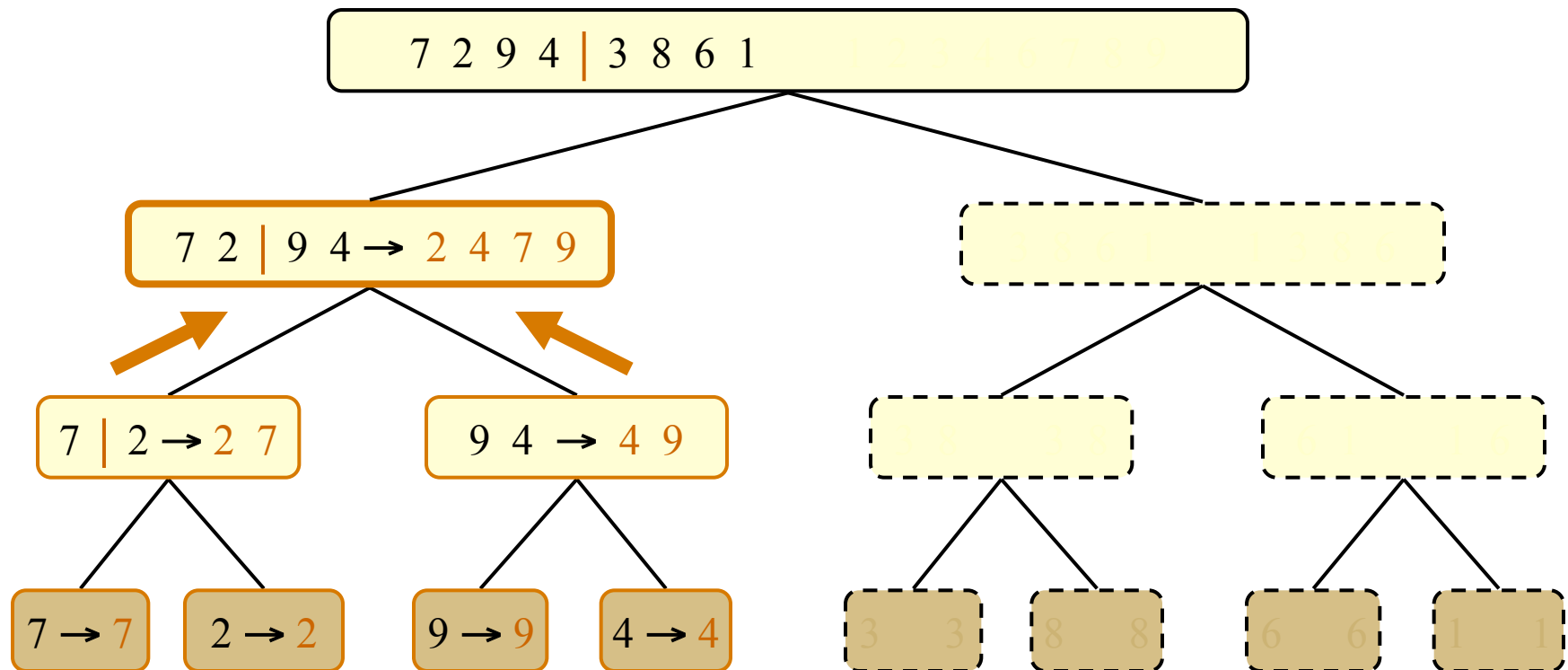
MergeSort: Exemplo de Execução (cont.)

- Chamadas recursivas, casos bases e merge (intercalação)



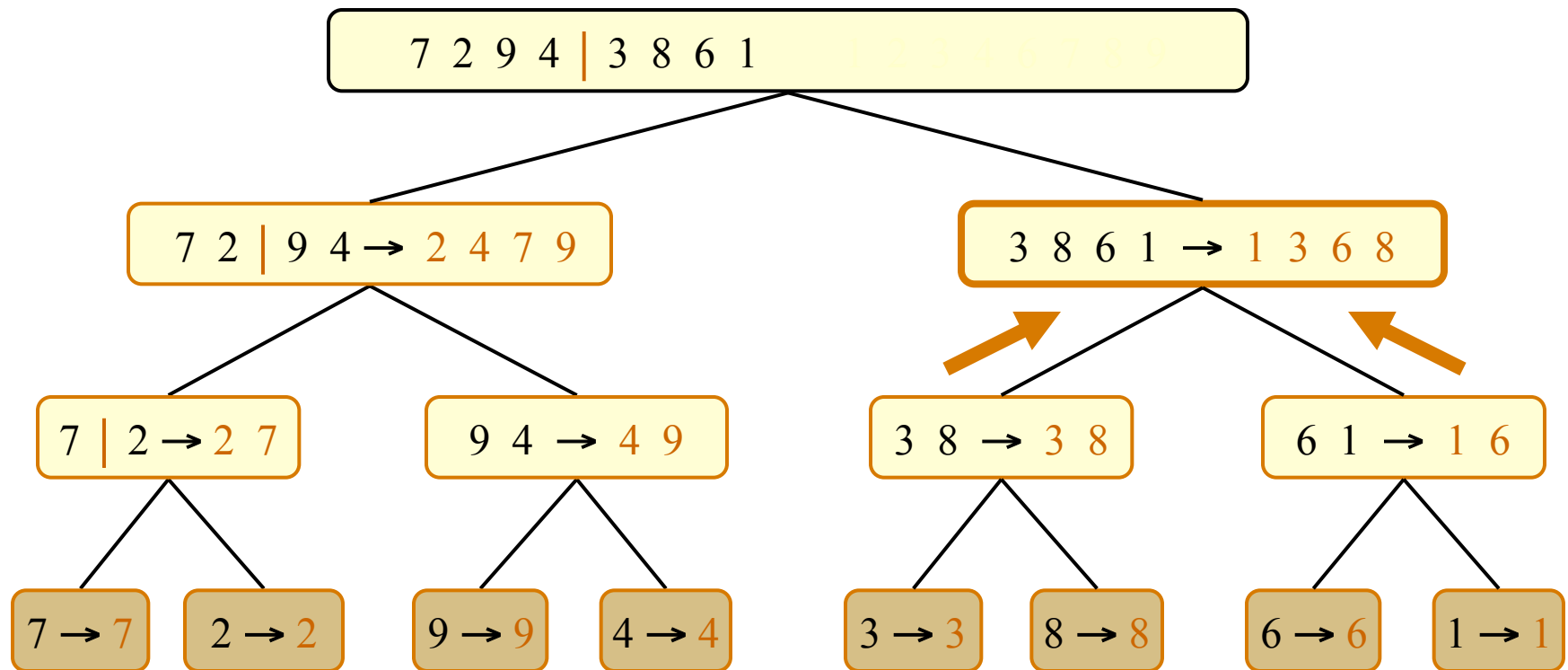
MergeSort: Exemplo de Execução (cont.)

- Operação de merge (intercalação)



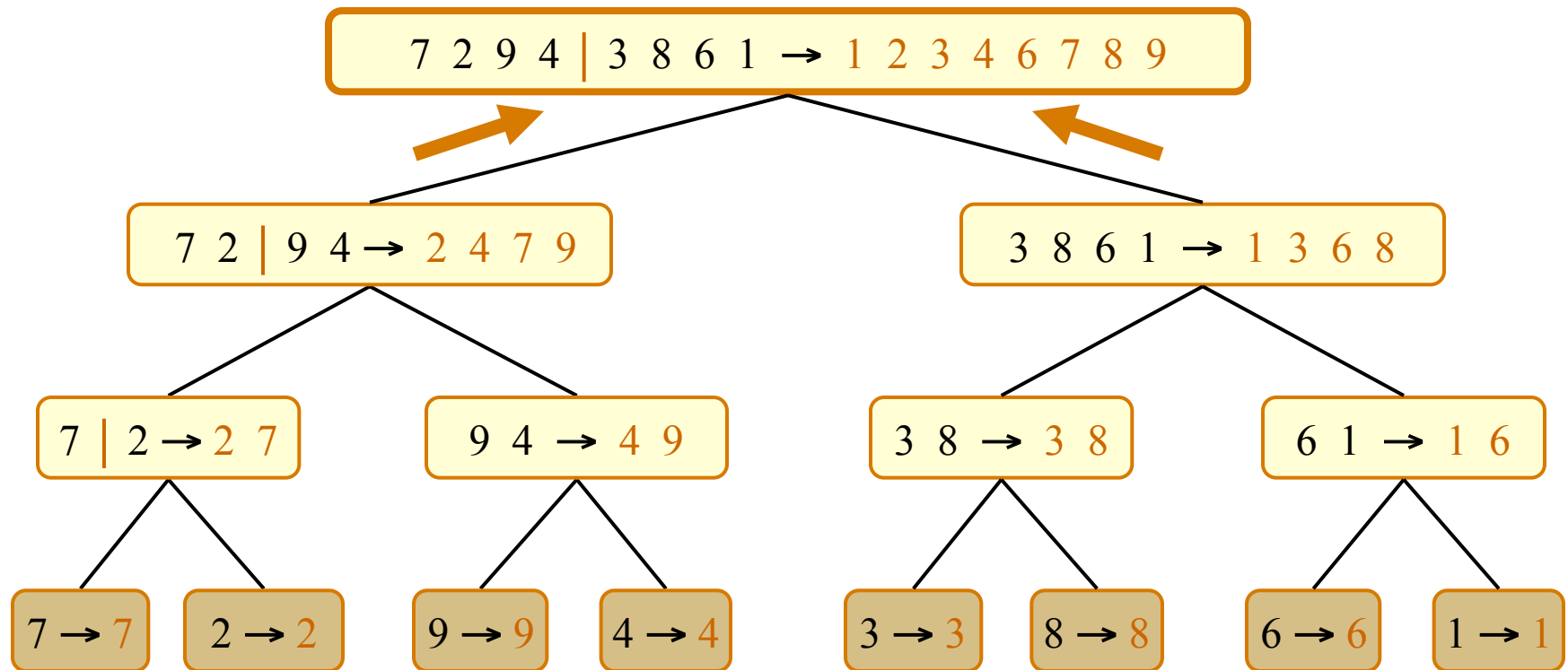
MergeSort: Exemplo de Execução (cont.)

- Execução do MergeSort para a outra partição



MergeSort: Exemplo de Execução (cont.)

- Finalmente o último merge (intercalação)

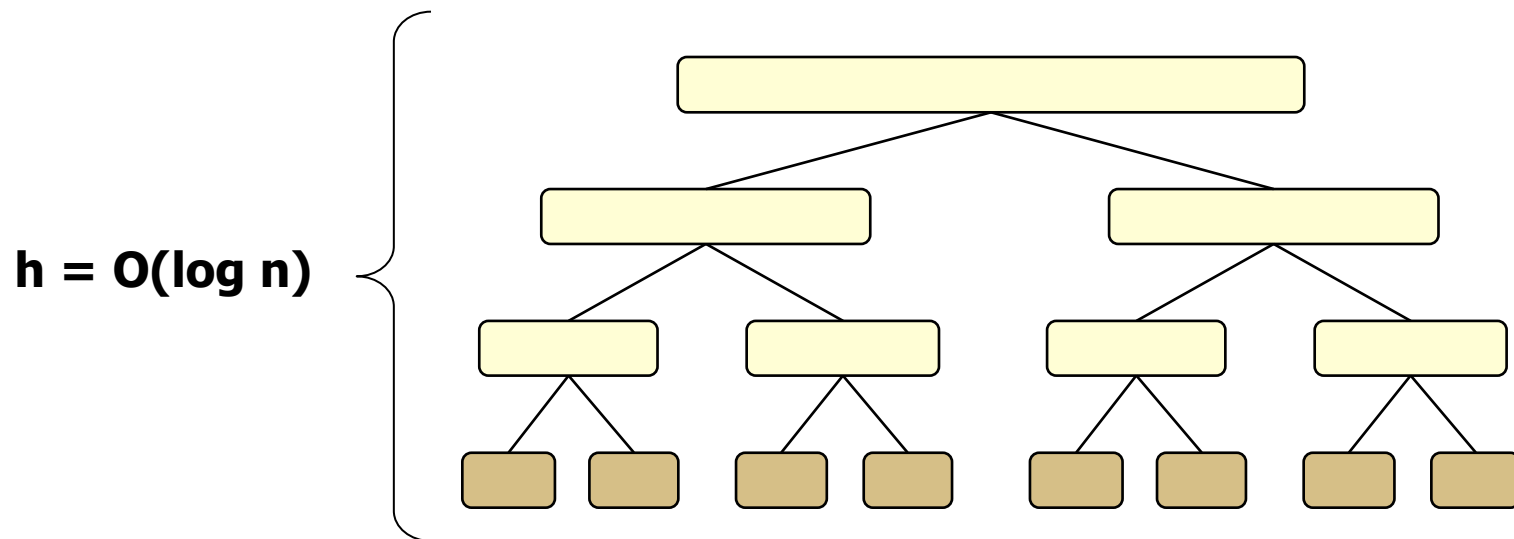


MERGESORT

ANÁLISE DO ALGORITMO

Análise do MergeSort

- A altura **h** da árvore de execução é **$O(\log n)$**
- A quantidade de operações em cada nível da árvore é assintoticamente igual a **$O(n)$**
- Logo: algoritmo é **$O(n \log n)$**



MERGESORT

IMPLEMENTAÇÃO RECURSIVA

Algoritmo MergeSort

```
/* ordena o vetor v[0..n-1] */  
void mergeSort(TItem *v, int n) {  
    mergeSort_ordena(v, 0, n-1);  
}  
  
/* ordena o vetor v[esq..dir] */  
void mergeSort_ordena(TItem *v, int esq, int dir) {  
    if (esq == dir)  
        return;  
  
    int meio = (esq + dir) / 2;  
    mergeSort_ordena(v, esq, meio);  
    mergeSort_ordena(v, meio+1, dir);  
    mergeSort_intercala(v, esq, meio, dir);  
    return;  
}
```


Algoritmo MergeSort

```
/* intercala os vetores v[esq..meio] e v[meio+1..dir] */
void mergeSort_intercala(TItem *v, int esq, int meio, int dir) {
    int i, j, k;
    int a_tam = meio-esq+1;
    int b_tam = dir-meio;
    TItem *a = (TItem*) malloc(sizeof(TItem) * a_tam);
    TItem *b = (TItem*) malloc(sizeof(TItem) * b_tam);

    for (i = 0; i < a_tam; i++) a[i] = v[i+esq];
    for (i = 0; i < b_tam; i++) b[i] = v[i+meio+1];

    for (i = 0, j = 0, k = esq; k <= dir; k++) {
        if (i == a_tam) v[k] = b[j++];
        else if (j == b_tam) v[k] = a[i++];
        else if (a[i].chave < b[j].chave) v[k] = a[i++];
        else v[k] = b[j++];
    }
    free(a); free(b);
}
```

Implementação do MergeSort

- O procedimento Intercala requer o uso de um segundo arranjo, B, para receber os dados ordenados.
- Note que no retorno de Mergesort com um arranjo de tamanho 1, a resposta encontra-se no arranjo A (o arranjo original de entrada).
- No próximo nível (arranjo de comprimento 2) o resultado da intercalação estará no arranjo B.

Implementação do MergeSort



- Podemos administrar este problema de duas maneiras:
 - Copiando a porção do arranjo referente ao resultado de volta para o arranjo A
 - Utilizando uma chave para indicar a “direção” dos movimentos de Intercala.

MERGESORT

VANTAGENS/DESVANTAGENS

- Vantagens
 - MergeSort é **$O(n \log n)$**
 - Indicado para aplicações que tem restrição de tempo (executa sempre em um determinado tempo para n)
 - Passível de ser transformado em estável
 - Tomando certos cuidados na implementação da intercalação
 - Fácil Implementação
- Desvantagens
 - Utiliza memória auxiliar – $O(n)$
 - Na prática é mais lento que QuickSort no caso médio

MERGESORT
IMPLEMENTAÇÃO
NÃO-RECURSIVA

MergeSort Não Recursivo

```
/* ordena o vetor v[0..n-1] - MergeSort iterativo */  
void mergeSort_iter(TItem *v, int n) {  
    int esq, dir;  
    int salto = 1;  
    while (salto < n) {  
        esq = 0;  
        while (esq + salto < n) {  
            dir = esq + 2*salto;  
            if (dir > n) dir = n;  
            mergeSort_intercala(v, esq, esq+salto-1, dir-1);  
            esq = esq + 2*salto;  
        }  
        salto = 2*salto;  
    }  
}
```



Perguntas?

MERGESORT
EXERCÍCIO

Exercício

- Dada a sequência de números:

3 4 9 2 5 1 8

Ordene em ordem crescente utilizando o algoritmo **MergeSort**, apresentado a sequência dos números a cada passo do algoritmo.