BCC202 - Estrutura de Dados I

Aula 13: Ordenação: MergeSort

Reinaldo Fortes

Universidade Federal de Ouro Preto, UFOP
Departamento de Computação, DECOM

Website: **www.decom.ufop.br/reifortes**
Email: **reifortes@iceb.ufop.br**

Material elaborado com base nos slides do Prof. Túlio Toffolo (curso de 2013/01).

2013/02

Conteúdo

Conteúdo

Motivação

- 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.

Motivação

- 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

- 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:
  - **Uso eficiente da memória cache**: ao final da fase de divisão grande parte dos dados necessários para a fase de combinação já estão disponíveis na cache.
    - Com isso, 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

```
1 divisao_e_conquista(x):
2   if x pequeno ou simples:
3     return resolve(x)
4   else:
5     decompor x em n conjuntos menores x[0], x[1],..., x[n-1]
6     for i in [0,1,...,n-1]:
7         y[i] = divisao_e_conquista(x[i])
8     combinar y[0],y[1],...,y[n-1] em y
9     return y
```

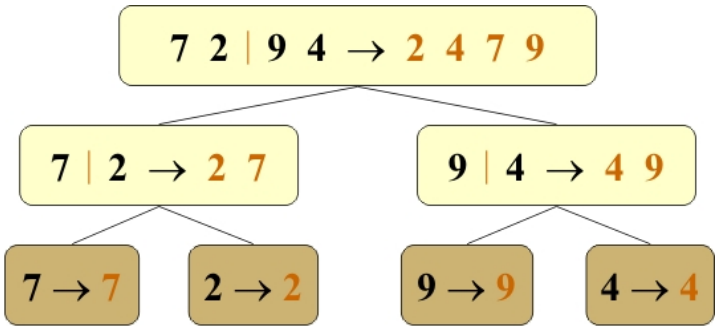
Utilização na ordenação

- Métodos de ordenação que utilizam-se da **Divisão e Conquista**:
  - **MergeSort**: aula de hoje.
  - **QuickSort**: próxima aula teórica.
- Principal diferença:
  - **MergeSort**: sempre divide o problema de forma balanceada (subproblemas de mesmo tamanho).
  - **QuickSort**: utiliza o conteito de **pivô** para dividir o problema em subproblemas (subproblemas de tamanhos diferentes).

Conteúdo

Funcionamento

- 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 folha são vetores de 1 ou 2 números (casos base).



Animação

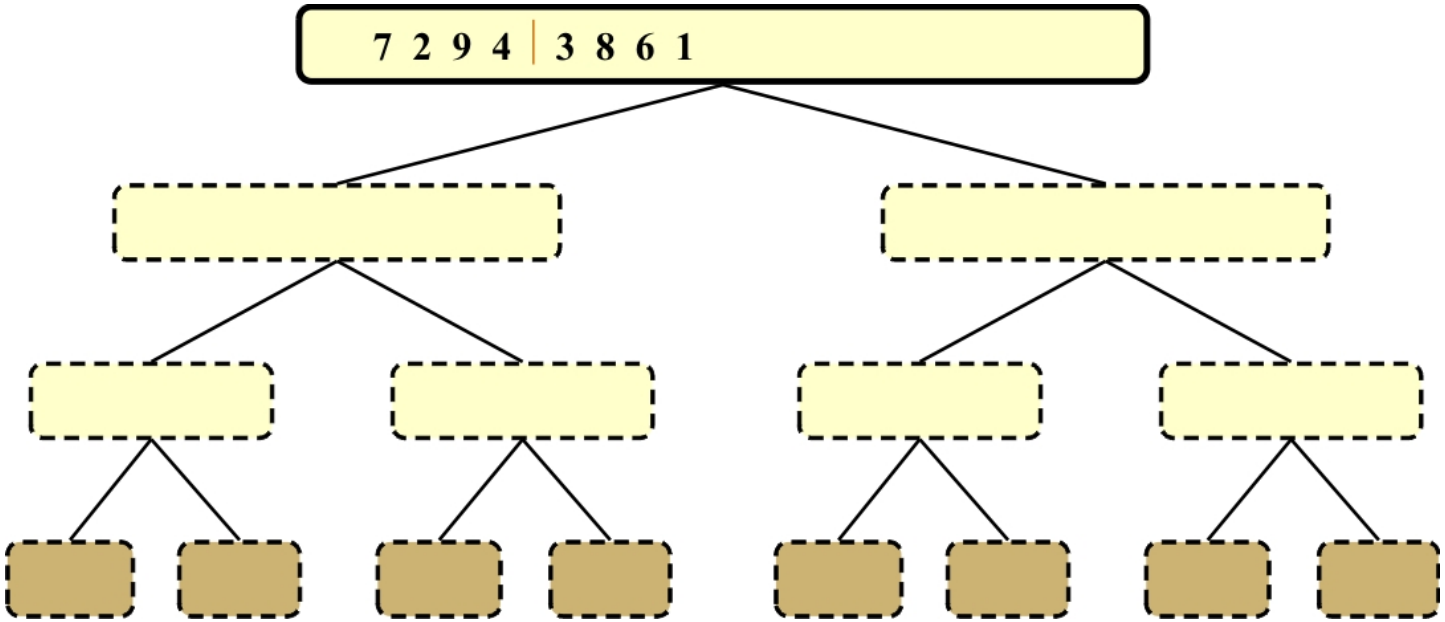
Algoritmo

- Divida o algoritmo em duas partes.
- Ordene as duas partes usando chamadas recursivas.
- Intercale as duas partes, obtendo um conjunto ordenado de todos os elementos.


Animação

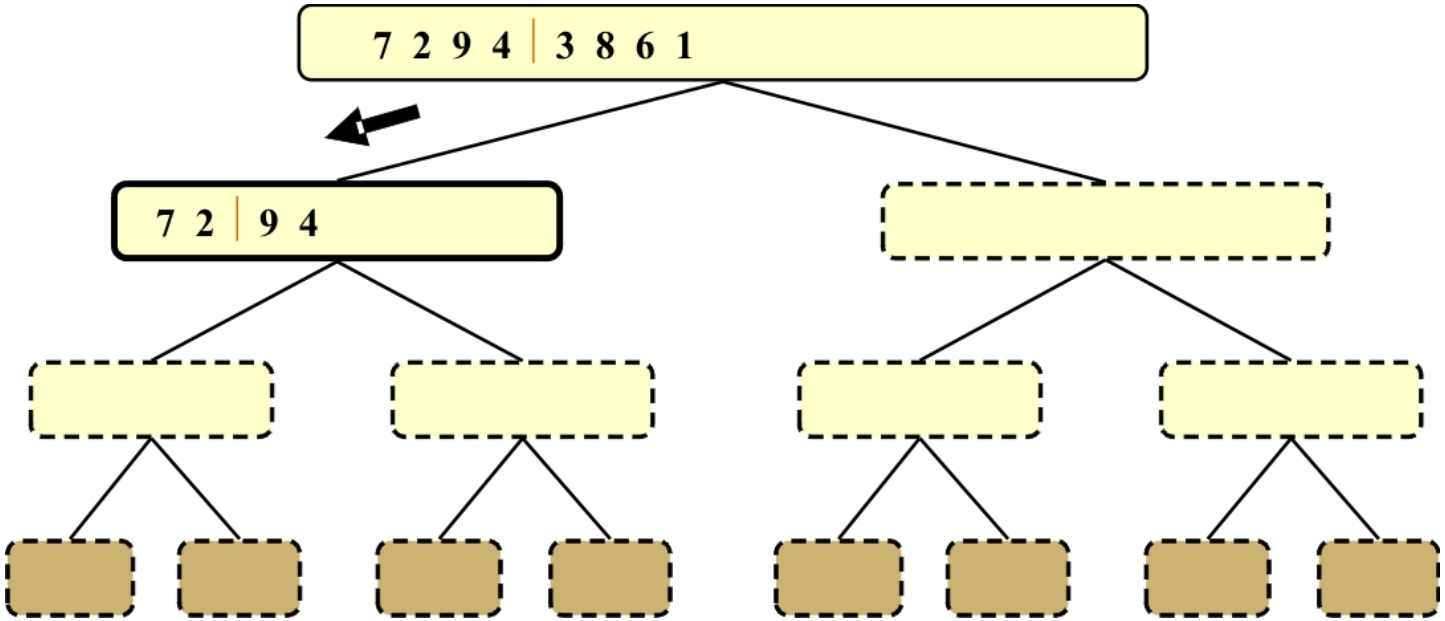

Vídeo

Execução



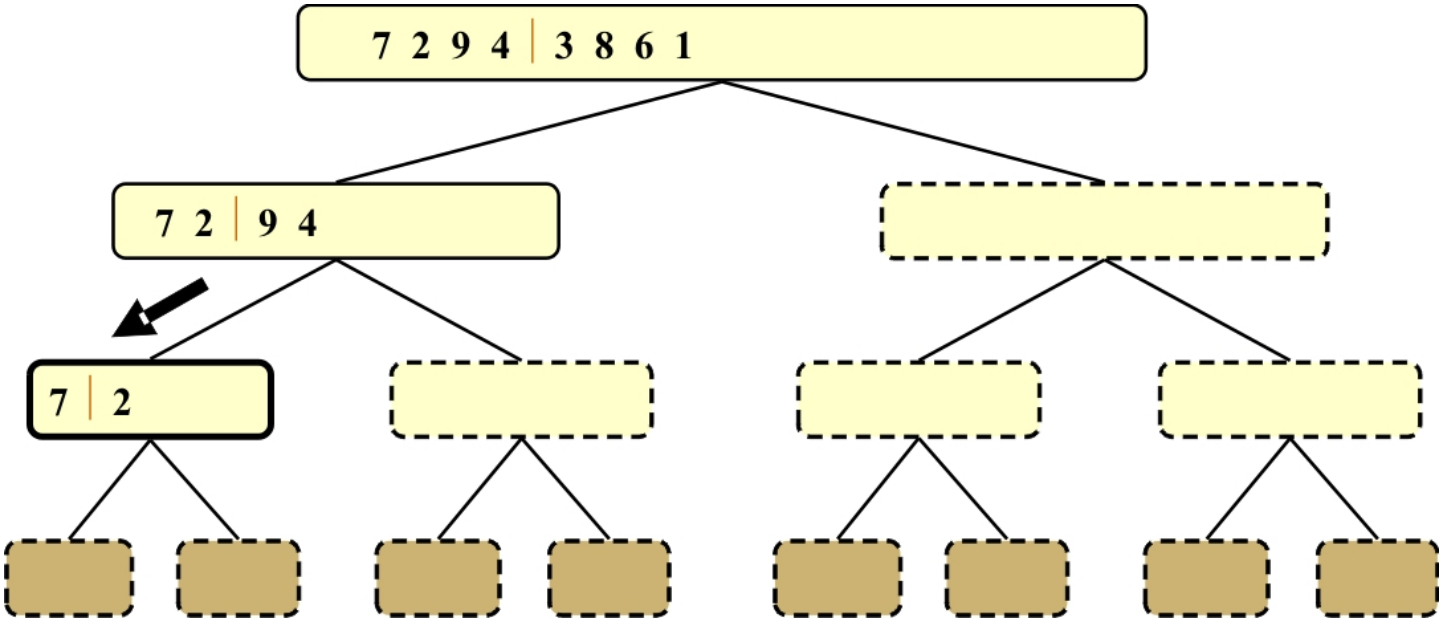
Partição do problema (sempre no meio do vetor).

Execução



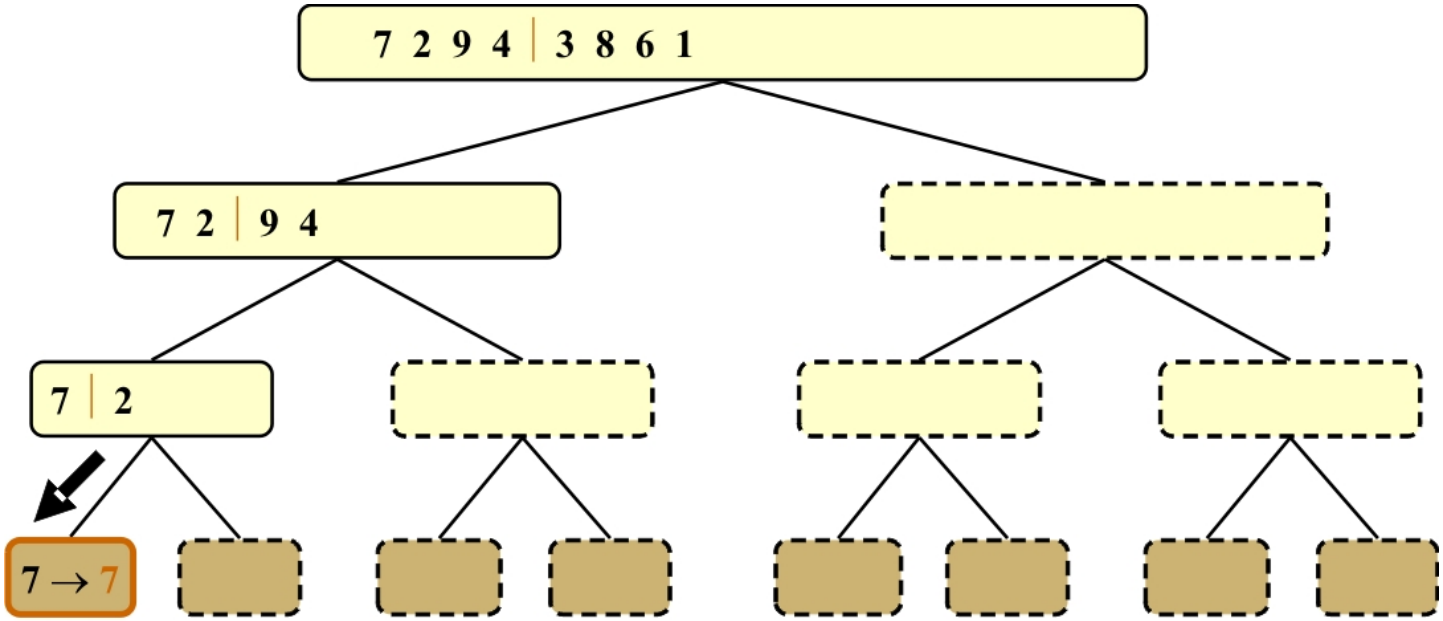
Chamada recursiva para primeira partição.

Execução



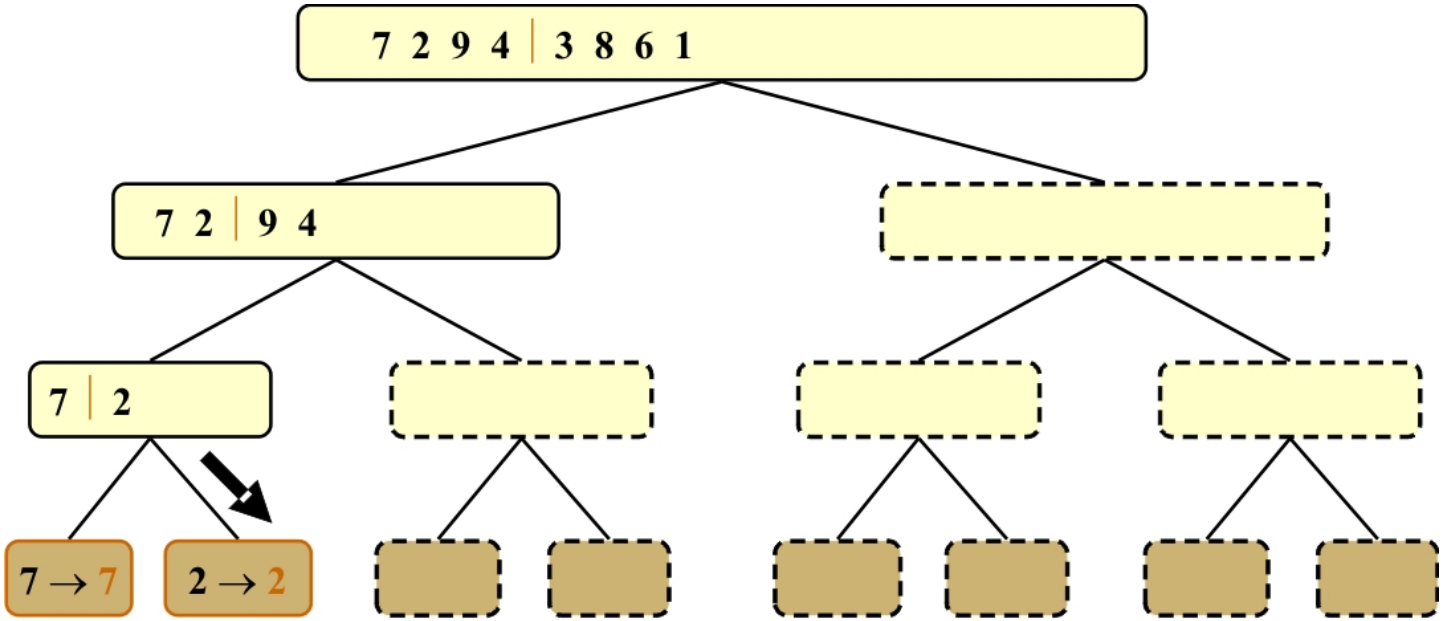
Chamada recursiva.

Execução



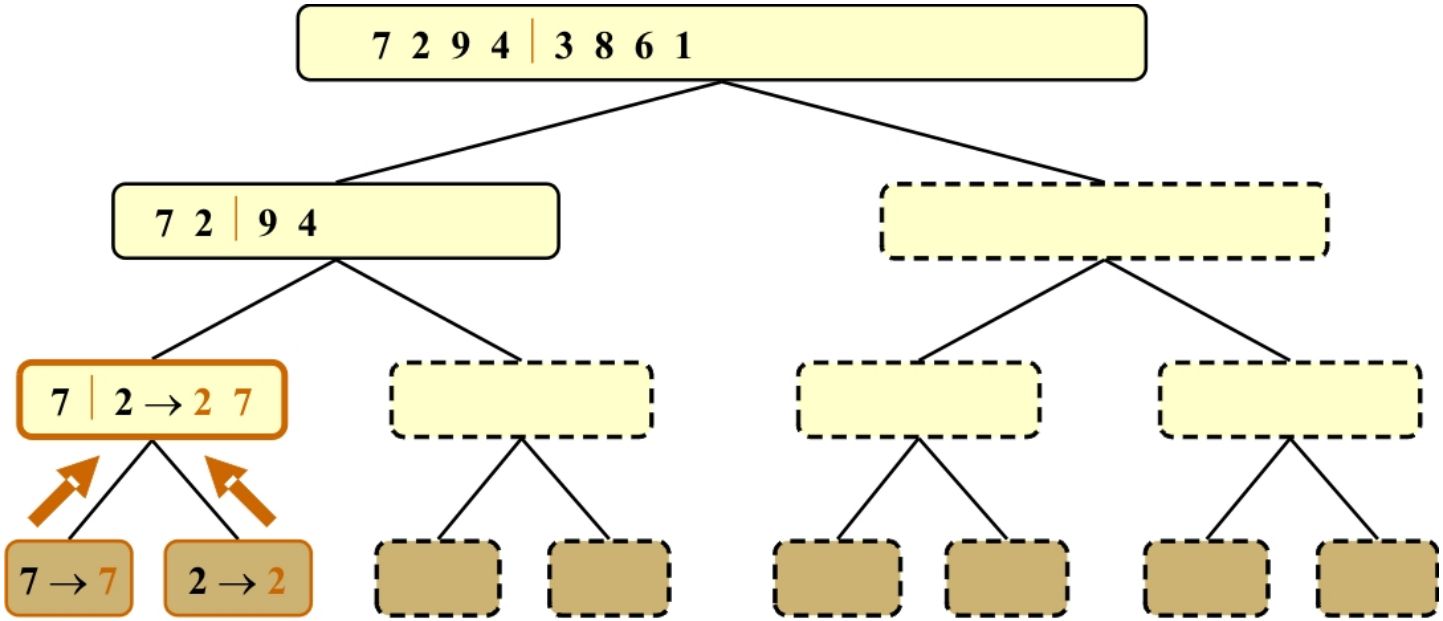
Chamada recursiva: caso base encontrado.

Execução



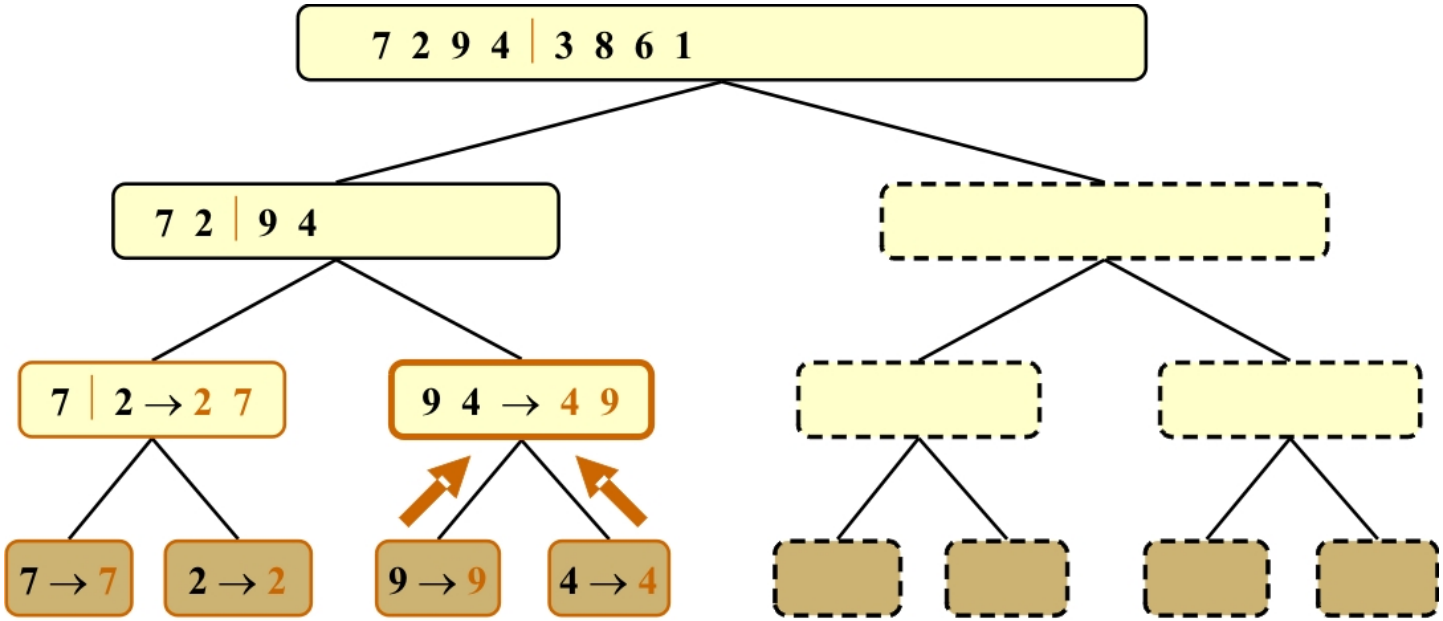
Chamada recursiva: caso base encontrado.

Execução



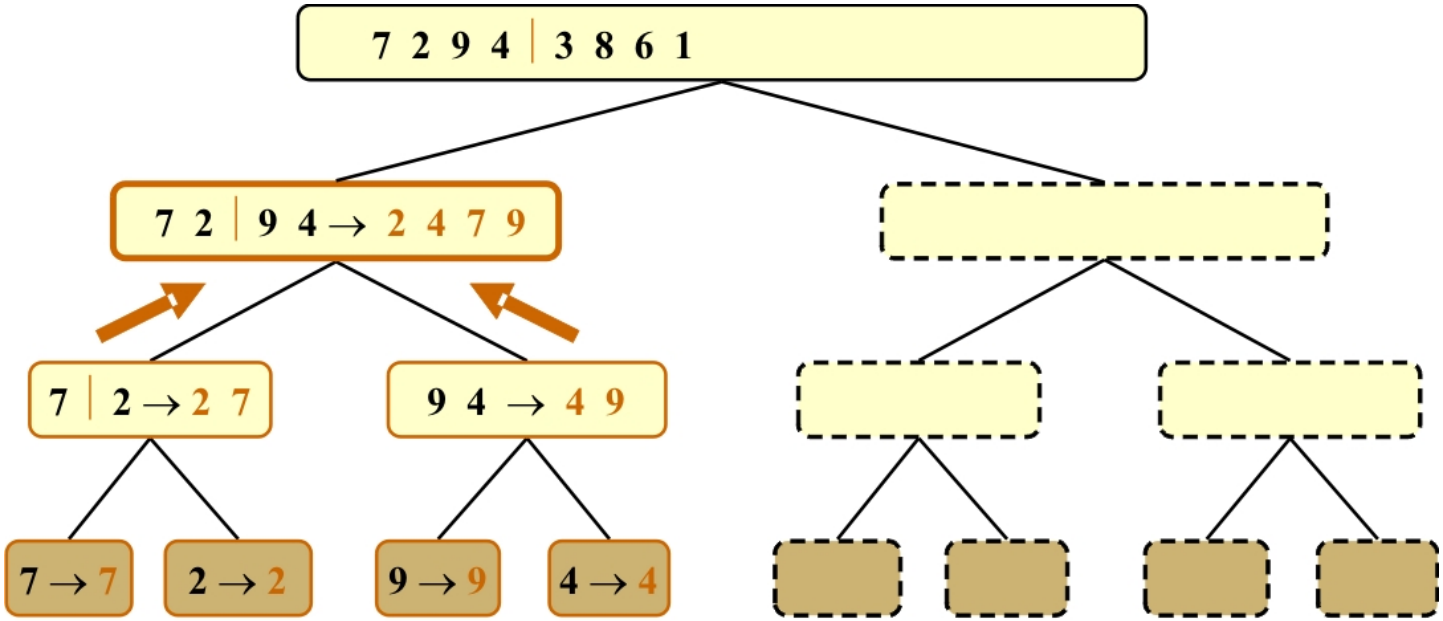
Operação de merge (intercalação).

Execução



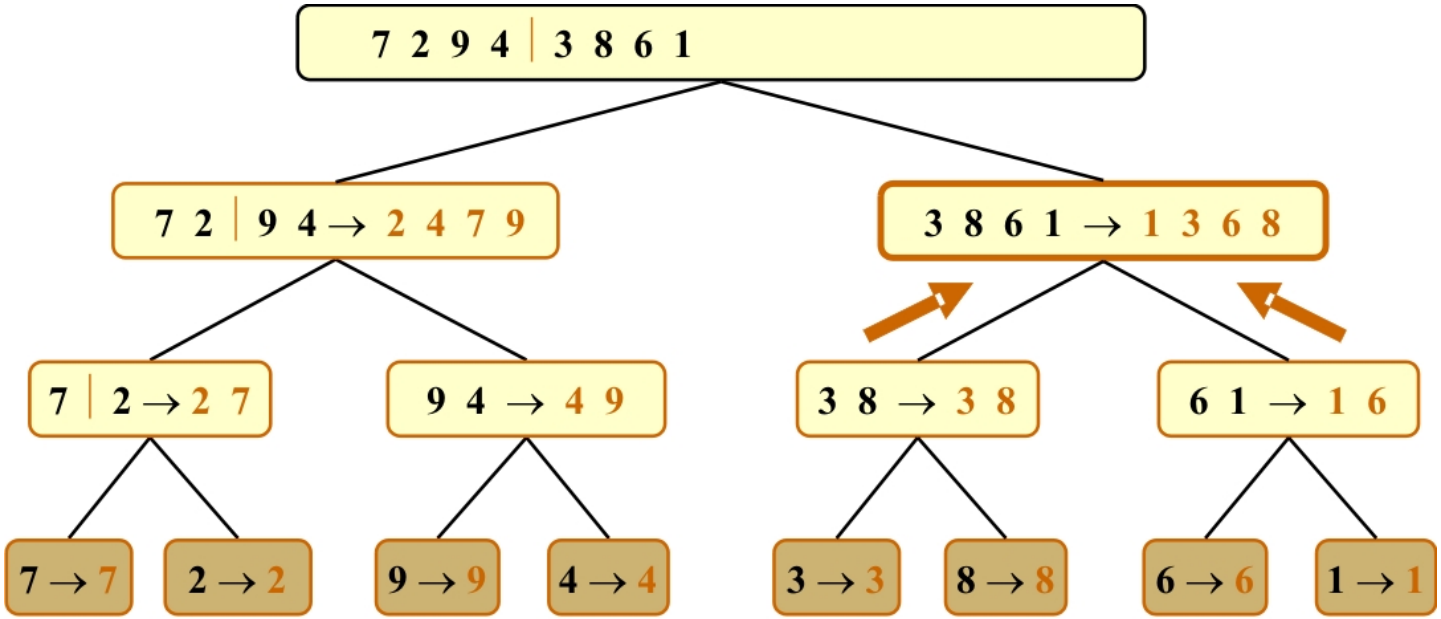
Chamadas recursivas, casos bases e merge (intercalação).

Execução



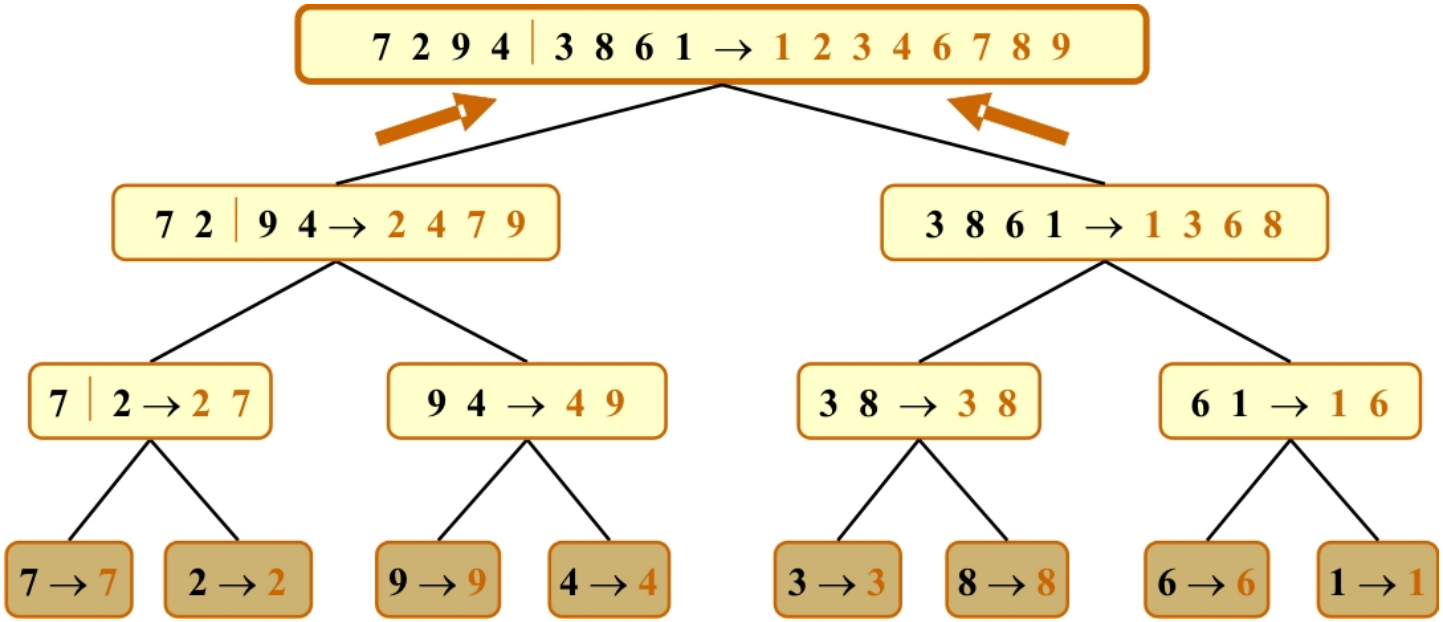
Operação de merge (intercalação).

Execução



Execução do MergeSort para a outra partição.

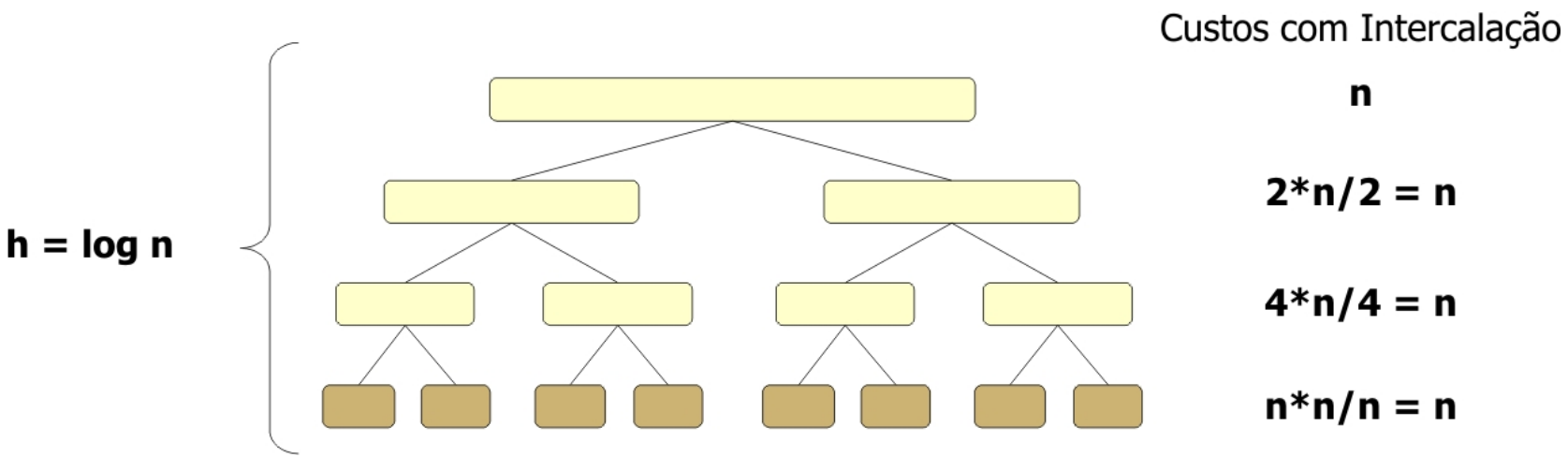
Execução



Finalmente o último merge (intercalação).

Ordem de Complexidade

- 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 $O(n\ log\ n)$.



Conteúdo

MergeSort RECURSIVO

```
1  // Ordena o vetor v[0..n-1]
2  void mergeSort(TItem *v, int n) {
3    mergeSort_ordena(v, 0, n-1);
4  }
5
6  // Ordena o vetor v[esq..dir]
7  void mergeSort_ordena(TItem *v, int esq, int dir) {
8    if  (esq == dir)
9      return;
10
11   int meio = (esq + dir) / 2;
12   mergeSort_ordena(v, esq, meio);
13   mergeSort_ordena(v, meio+1, dir);
14   mergeSort_intercala(v, esq, meio, dir);
15   return;
16 }
```

MergeSort RECURSIVO

```
18 // Intercala os vetores v[esq..meio] e v[meio+1..dir]
19 void mergeSort_intercala(TItem *v, int esq, int meio, int
       dir) {
20   int i, j, k;
21   int a_tam = meio-esq+1;
22   int b_tam = dir-meio;
23   TItem *a = (TItem*) malloc(sizeof(TItem) * a_tam);
24   TItem *b = (TItem*) malloc(sizeof(TItem) * b_tam);
25
26   for (i = 0; i < a_tam; i++) a[i] = v[i+esq];
27   for (i = 0; i < b_tam; i++) b[i] = v[i+meio+1];
28
29   for (i = 0, j = 0, k = esq; k <= dir; k++) {
30     if (i == a_tam) v[k] = b[j++];
31     else if (j == b_tam) v[k] = a[i++];
32     else if (a[i].chave < b[j].chave) v[k] = a[i++];
33     else v[k] = b[j++];
34   }
35   free(a); free(b);
36 }
```

MergeSort RECURSIVO: Características

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, em $O(n)$.
- Na prática é mais lento que **QuickSort** (próxima aula) no caso médio.

Conteúdo

MergeSort ITERATIVO

```
1  // Ordena o vetor v[0..n-1]
2  // A cada iteração, intercala dois "blocos" de b elementos:
3  // - o primeiro com o segundo, o terceiro com o quarto etc.
4  void mergeSort_iter(TItem *v, int n) {
5    int esq, dir;
6    int b = 1;
7    while (b < n) {
8      esq = 0;
9      while (esq + b < n) {
10       dir = esq + 2 * b;
11       if (dir > n) dir = n;
12       mergeSort_intercala(v, esq, esq+b-1, dir-1);
13       esq = esq + 2 * b;
14     }
15     b *= 2;
16   }
17 }
```

Conteúdo

Conclusão

- Nesta aula tivemos contato com o *algoritmo de ordenação* chamado **MergeSort**.
- Foram vistas duas versões: **Recursiva** e **Iterativa**.
- Apesar de mais simples de entender e implementar, a versão recursiva requer memória adicional.

Conclusão

- Quadro comparativo dos métodos de ordenação:

Algoritmo	Comparações			Movimentações			Espaço	Estável	In situ
	Melhor	Médio	Pior	Melhor	Médio	Pior			
Bubble	$O(n^2)$			$O(n^2)$			$O(1)$	**Sim**	**Sim**
Selection	$O(n^2)$			$O(n)$			$O(1)$	**Não***	**Sim**
Insertion	$O(n)$	$O(n^2)$		$O(n)$	$O(n^2)$		$O(1)$	**Sim**	**Sim**
Merge	$O(n \log n)$			–			$O(n)$	**Sim**	**Não**

* Existem versões estáveis.

Conclusão

- A tarefa de ordenação é muito importante, ela é uma necessidade básica para a solução de muitos problemas.
- *Próxima aula*: *QuickSort*.
- Dúvidas?

Conteúdo

Exercício 01

- Dada a sequência de números: `3 4 9 2 5 1 8`.
- Ordene em ordem crescente utilizando o algoritmo aprendido em sala (**MergeSort**), apresentando a sequência dos números a cada passo (Teste de Mesa).