UNISINOS

Estruturas Avançadas de Dados I

(Merge Sort)

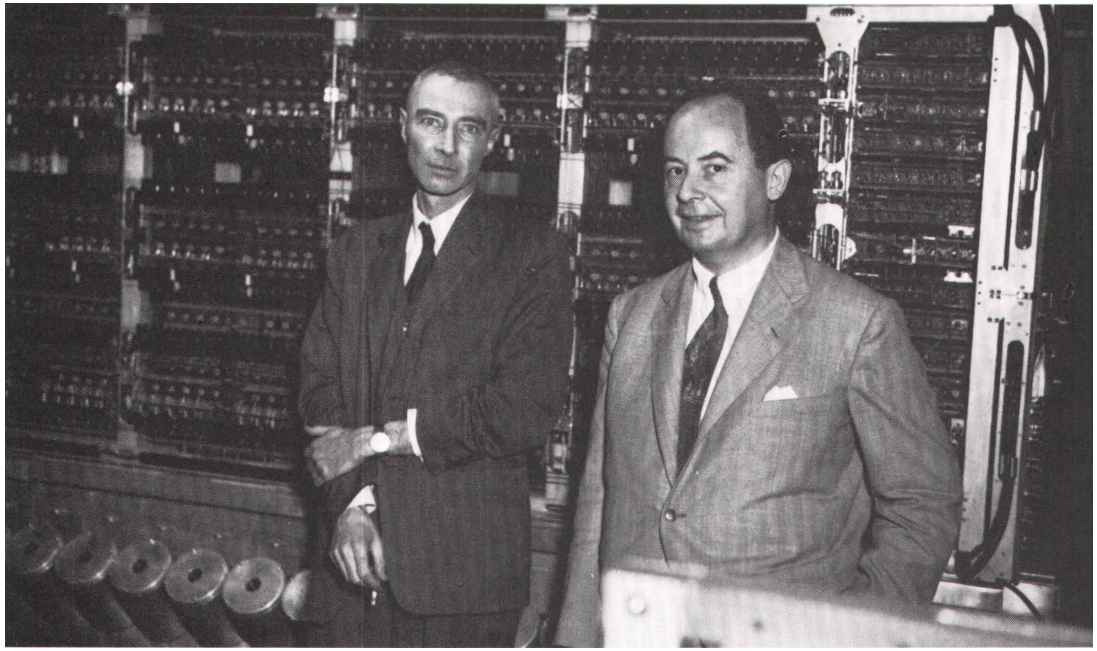
Prof. Gilberto Irajá Müller

Última atualização 05/06/2018

Introdução

- Proposto por John Von Neumann em 1945;
- Constante no artigo "First Draft of a Report on the EDVAC;
  - Desenvolvido para o EDVAC (sucessor do ENIAC);
  - Proposta de arquitetura de um computador que é similar aos computadores modernos (Arquitetura de Von Neumann).
- Projeto Manhattan;

J. Robert Oppenheimer e John Von Neumann



Introdução (cont.)

- Usa uma estratégia muito importante em Ciência da Computação: divisão (divide) e conquista (conquer);

- Do ponto de vista do Merge Sort:
  - **Divisão**: se o tamanho da entrada é muito grande para se lidar computacionalmente, divide-se em duas partes iguais e distintas;
  - **Recursão**: usado para dividir e conquistar de forma a resolver os subproblemas; ordena-se o subproblema!
  - **Conquistar**: pega-se os subproblemas resolvidos e "une-os" em uma solução que representa o problema original.

- Duas abordagens: top-down (base deste material) e bottom-up.

Introdução (cont.)

- Se S tem pelo menos dois elementos, **divida-os** em duas subsequências, S_1 e S_2, onde cada uma contém metade dos elementos de S. S_1 contém [0, middle] e S_2 contém [middle + 1, n – 1]; sendo que middle é n / 2;

- **Recursivamente** ordene as sequências S_1 e S_2;

- Coloque os elementos constantes em S_1 e S_2 novamente em S **unindo-os** (colocando sempre o menor elemento de S_1 ou S_2) de forma ordenada.

Implementação do Merge Sort

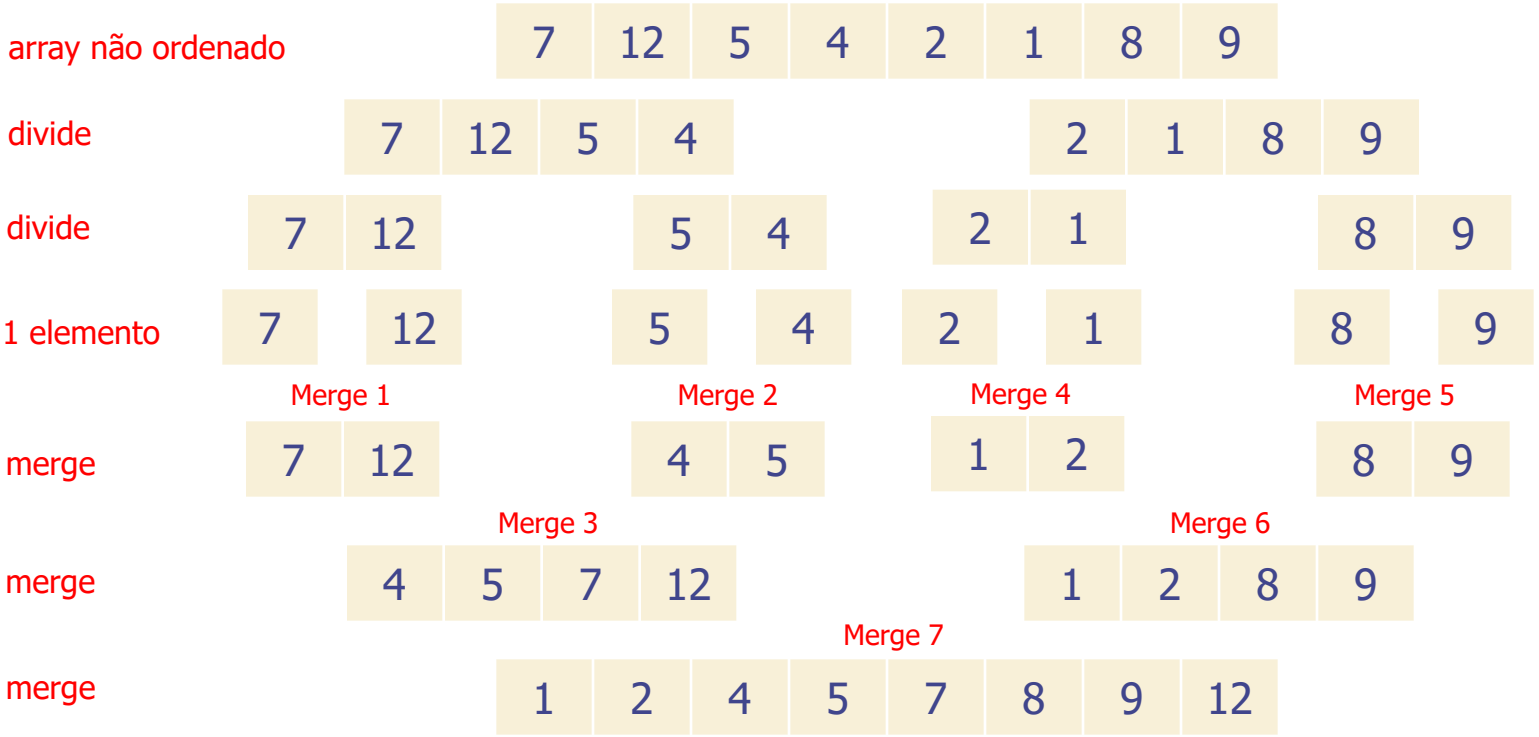
```
public static <T extends Comparable<? super T>> void mergeSort(T[] a) {
  T[] aux = (T[]) new Comparable[a.length];
  divide(a, aux, 0, a.length - 1);
}
```

```
private static <T extends Comparable<? super T>> void divide(T[] a, T[] aux, int low, int high) {
  if (low >= high) return;

  int middle = (low + high) / 2;
  divide(a, aux, low, middle);
  divide(a, aux, middle + 1, high);
  conquer(a, aux, low, middle, high);
}
```

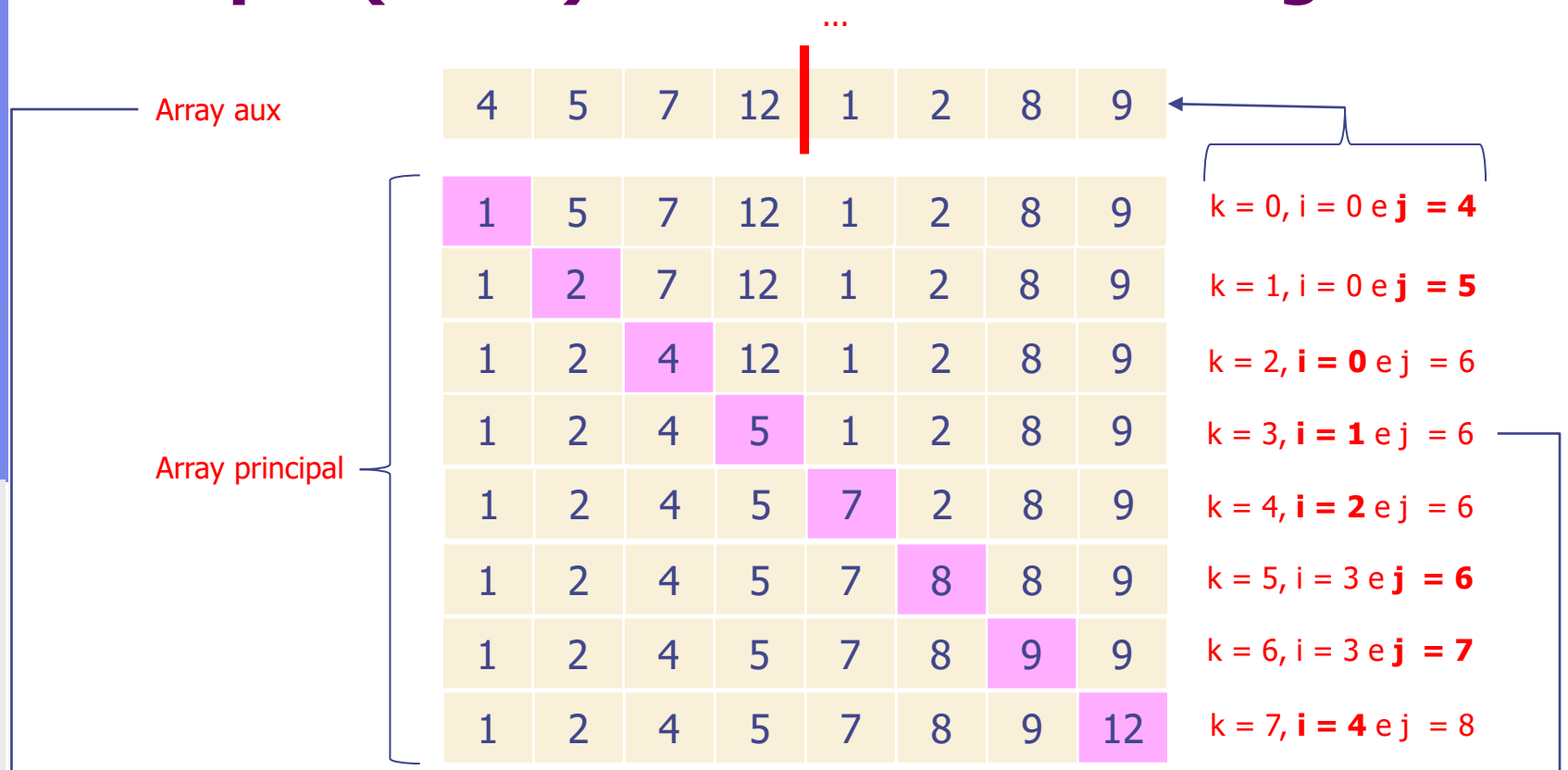
```
private static <T extends Comparable<? super T>> void conquer(T[] a, T[] aux, int
low, int middle, int high) {
  for (int k = low; k <= high; k++) {
    aux[k] = a[k];
  }
  int i = low, j = middle + 1;
  for (int k = low; k <= high; k++) {
     if      (i > middle)                    a[k] = aux[j++];
     else if (j > high)                      a[k] = aux[i++];
     else if (aux[j].compareTo(aux[i]) < 0)  a[k] = aux[j++];
     else                                    a[k] = aux[i++];
  }
}
```

Exemplo



- O Merge Sort necessita de um array auxiliar de tamanho *n* e, portanto, é um algoritmo **out-place**;
- Além disso, é um algoritmo **estável** (na maioria das implementações) dado que a ordem relativa das chaves iguais permanece a mesma.

Exemplo (cont.) – Passo último merge




```
private static <T extends Comparable<? super T>> void conquer(T[] a, T[] aux, int low, int middle, int high) {
  for (int k = low; k <= high; k++) {
    aux[k] = a[k];
  }
  int i = low, j = middle + 1;
  for (int k = low; k <= high; k++) {
     if      (i > middle)                    a[k] = aux[j++];
     else if (j > high)                      a[k] = aux[i++];
     else if (aux[j].compareTo(aux[i]) < 0)  a[k] = aux[j++];
     else                                    a[k] = aux[i++];
  }
}
```

Complexidade

Método	Caso médio	Melhor caso	Pior caso	Complexidade de Espaço	Estável	Interno	Recursivo	Comparação
Bubble Sort	$O(n^2)$	O(n)	$O(n^2)$	In-place = O(1)	Sim	Sim	Não	Sim
Insertion Sort	$O(n^2)$	O(n)	$O(n^2)$	In-place = O(1)	Sim	Sim	Não	Sim
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	In-place = O(1)	Sim	Sim	Não	Sim
Shell Sort	$O(n^{7/6})$ – depende do gap	O(nlog(n))	O(nlog(n)) a $O(n^{3/2})$	In-place = O(1)	Não	Sim	Não	Sim
Heap Sort	O(nlog(n))	O(nlog(n))	O(nlog(n))	In-place = O(1)	Não	Sim	Sim	Sim
Merge Sort	**O(nlog(n))**	**O(nlog(n))**	**O(nlog(n))**	**Out-place = O(n)**	**Sim**	**Sim (há implementações para Externo)**	**Sim**	**Sim**

Observa-se que o Merge Sort possui um bom desempenho nos três casos de O(nlog(n)); além disso, é um algoritmo que possui implementações para memória interna/externa;

Contudo, ao ser considerado a complexidade de espaço, utiliza uma estrutura auxiliar para ordenação de O(n). Há implementações in-place.

Implementação do Merge Sort – Bottom-Up

```
public static <T extends Comparable<? super T>> void mergeSortBottomUp(T[] a) {
  int n = a.length;
  T[] aux = (T[]) new Comparable[n];
  for (int len = 1; len < n; len *= 2) {
     for (int low = 0; low < n - len; low += len + len) {
       int middle = low + len - 1;
       int high = Math.min(low + len + len - 1, n - 1);
       conquer(a, aux, low, middle, high);
     }
  }
}
```

Exercícios Teóricos

- Exercício 1. Considerando o seguinte array:

a) Aplique o Merge Sort (passo-a-passo) mostrando o processo de divisão e merge.

Referências Bibliográficas

- CORMEN, Thomas H. et al. **Introduction to algorithms.** 3. ed. Cambridge: MIT, 2009. xix. 1292 p.

- https://algs4.cs.princeton.edu/22mergesort/. Acessado em 24/10/2017.

- http://www.cs.princeton.edu/courses/archive/spring14/cos226/lectures/22Mergesort.pdf. Acessado em 23/10/2017.