INF 112: Programação II

Aula 09

- **Algoritmos de ordenação – parte 3**

Fábio R. Cerqueira, UFV, DPI, frcerqueira@gmail.com

Algoritmos de ordenação - QuickSort

- *QuickSort* é outro importante algoritmo de ordenação baseado na estratégia dividir-para-conquistar.

- Foi proposto por C.A.R Hoare em 1960. É um algoritmo de ordenação interna muito utilizado pois possui o melhor tempo médio de resposta.

- Diferentemente do *MergeSort*, que particiona a lista de elementos de acordo com suas posições, o *QuickSort* divide os elementos baseado em seus valores.

- Em particular, os elementos sofrem um rearranjo para que a partição seja realizada.

Algoritmos de ordenação - QuickSort

- A partição, no caso do *QuickSort*, representa uma situação onde todos os elementos antes de uma posição *s* são menores ou iguais a *A*[*s*] e todos os elementos após a posição *s* são maiores ou iguais a *A*[*s*]:

$$\underbrace{A[0]\ \ldots\ A[s-1]}_{\leq A[s]}\quad A[s]\quad \underbrace{A[s+1]\ \ldots\ A[n-1]}_{\geq A[s]}$$

- Desta forma, após a partição ter sido realizada, considera-se que *A*[*s*] já se encontra em sua posição final.

- Pode-se então prosseguir com a ordenação dos dois subarranjos compostos pelos elementos que precedem e sucedem *A*[*s*]. A ordenação de ambos é feita de modo independente e aplicando-se a mesma ideia acima.

Algoritmos de ordenação - QuickSort

→ Passos em alto nível do algoritmo *QuickSort*:

- Selecione um *pivô* (elemento de partição) – escolheremos aqui o primeiro elemento;

- Rearranje a lista tal que todos os elementos antes da posição de partição *s* (subarranjo 1) sejam menores ou iguais ao pivô e todos os elementos das posições posteriores a *s* (subarranjo 2) sejam maiores ou iguais ao pivô;

- Troque o pivô com o elemento da posição *s*: o pivô encontra-se agora em sua posição final;

- Ordene os dois subarranjos resultantes aplicando a mesma ideia acima.

Algoritmos de ordenação - QuickSort

- Segue o pseudocódigo:

ALGORITHM *Quicksort*($A[l..r]$)
//Sorts a subarray by quicksort
//Input: A subarray $A[l..r]$ of $A[0..n-1]$, defined by its left and right indices
// l and r
//Output: Subarray $A[l..r]$ sorted in nondecreasing order
if $l < r$
$s \leftarrow Partition(A[l..r])$ //s is a split position
$Quicksort(A[l..s-1])$
$Quicksort(A[s+1..r])$

Algoritmos de ordenação - QuickSort

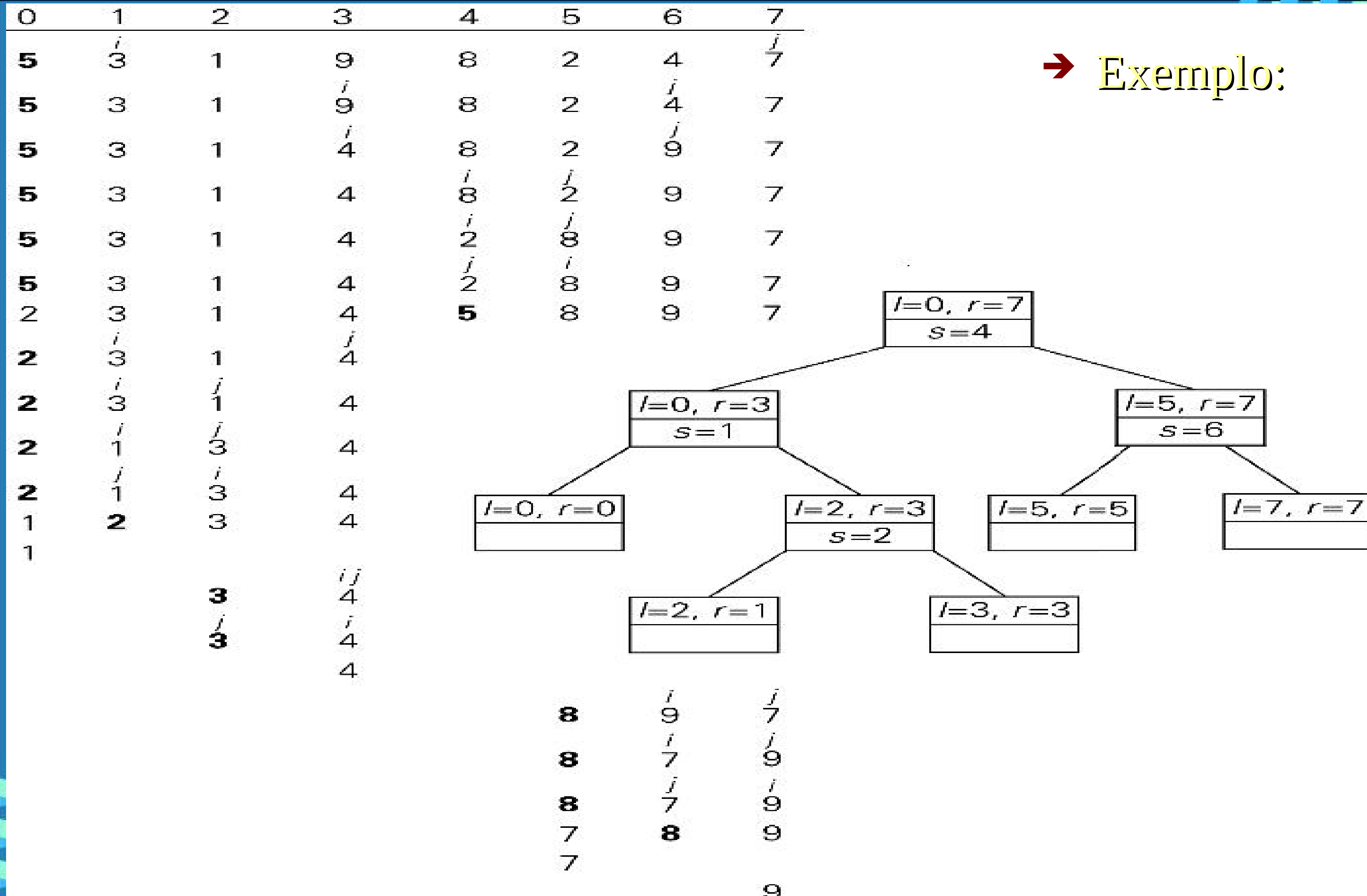
ALGORITHM *Partition*$(A[l..r])$

```
//Partitions a subarray by using its first element as a pivot
//Input: A subarray A[l..r] of A[0..n − 1], defined by its left and right
//       indices l and r (l < r)
//Output: A partition of A[l..r], with the split position returned as
//        this function's value
p ← A[l]
i ← l;   j ← r + 1
repeat
    repeat i ← i + 1 until A[i] ≥ p
    repeat j ← j − 1 until A[j] ≤ p
    swap(A[i], A[j])
until i ≥ j
swap(A[i], A[j])   //undo last swap when i ≥ j
swap(A[l], A[j])
return j
```

- Note que, desta forma, o índice *i* pode atingir valor inválido (> *r*). Pode-se resolver isto checando-se o valor de *i* a cada iteração ou então anexando-se um valor sentinela ao arranjo *A*.

Algoritmos de ordenação - QuickSort

- Exemplo:



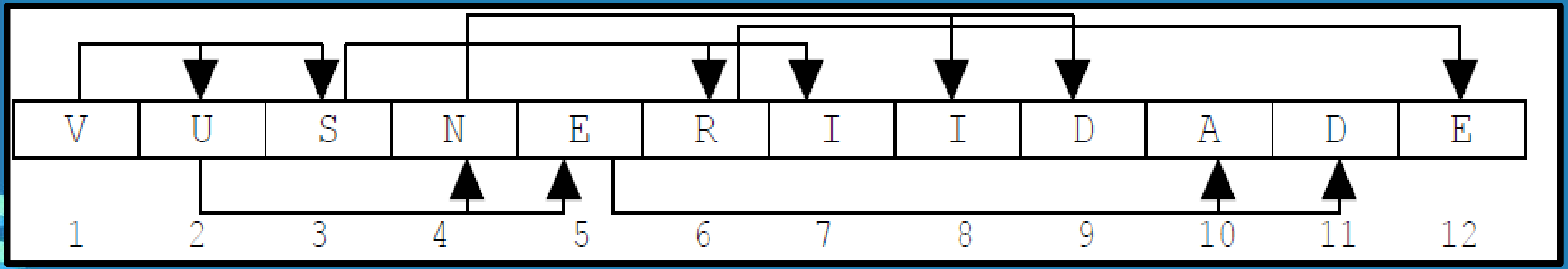
Algoritmos de ordenação - QuickSort

- A parte crucial do algoritmo *QuickSort* é o particionamento do arranjo. A escolha do elemento pivô deve ser feita de tal forma que em média ele se posicione na parte central do arranjo que está sendo ordenado.
- Se o pivô tender a se posicionar nas extremidades do arranjo, o desempenho do algoritmo cairá.
- Com relação ao número de comparações que realiza, o *Quicksort* é $O(n \log n)$ para o melhor caso e para o caso médio.
- Já para o pior caso possui complexidade $O(n^2)$.
- Para evitar o pior caso uma sugestão é escolher aleatoriamente três elementos e usar como pivô a mediana.

Algoritmos de ordenação - HeapSort

- O algoritmo *HeapSort* ordena um arranjo de elementos através de reorganizações sucessivas do mesmo. A reorganização consiste em fazer com que os elementos do arranjo obedeçam uma regra conhecida como *condição de heap*. Esta condição estabelece que o elemento na *i*-ésima posição deve ser maior ou igual aos elementos nas posições $2*i$ e $2*i+1$, caso estas existam, que aqui chamaremos de *posições filhas*.

- Por exemplo, o arranjo abaixo obedece a condição de *heap* (note que se assume, por conveniência, que o arranjo começa com índice 1):



Algoritmos de ordenação - HeapSort

- Nesta organização, as posições *n* DIV 2 + 1 até *n* não possuem posições à frente para verificação da condição de *heap*. Então, basta verificar esta condição da posição *n* DIV 2 até 1 que chamaremos de *posições parentais*.

- Segue a ideia geral de como reorganizar o arranjo para que o mesmo satisfaça a condição de *heap*:

Passo 1: Comece com a posição parental *n* DIV 2;

Passo 2: Verifique a condição de *heap* para a posição parental atual. Se a condição não for satisfeita, siga trocando o elemento desta posição com o maior de seus filhos até que a condição seja satisfeita;

Passo 3: Repita o passo 2 para a posição parental seguinte (atual-1) até que não haja mais posições parentais para processar.

Algoritmos de ordenação - HeapSort

- Segue a mesma ideia em pseudocódigo:

Algorithm $Heap(H[1..n])$
//Constructs a heap from the elements of a given array
// by the bottom-up algorithm
//Input: An array $H[1..n]$ of orderable items
//Output: A heap $H[1..n]$
for $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1 **do**
  $k \leftarrow i; \quad v \leftarrow H[k]$
  $heap \leftarrow$ **false**
  while not $heap$ **and** $2 * k \leq n$ **do**
    $j \leftarrow 2 * k$
    if $j < n$ //there are two children
      if $H[j] < H[j+1] \quad j \leftarrow j + 1$
    if $v \geq H[j]$
      $heap \leftarrow$ **true**
    else $H[k] \leftarrow H[j]; \quad k \leftarrow j$
  $H[k] \leftarrow v$

Algoritmos de ordenação - HeapSort

- Note que após esta organização o maior elemento fica na primeira posição do arranjo. O que o algoritmo *HeapSort* faz é trocá-lo com o elemento da última posição e recomeçar o mesmo procedimento para o subarranjo 1..*n*-1.

- Ideia geral do *HeapSort*:

Passo 1: Estabeleça a condição de *heap* para a entrada contendo *n* elementos;

Passo 2: Repita ***n***-1 vezes a operação de remoção do maior elemento:

  - Troque o primeiro com o último elemento;
  - Considere agora o arranjo que vai de 1 até tamanho anterior – 1;
  - Estabeleça a condição de *heap* para o novo arranjo.

Algoritmos de ordenação - HeapSort

- Exemplo: ordenando a lista 2, 9, 7, 6, 5, 8 pelo *HeapSort*:

Passo 1 (condição de *heap*)

2 9 **7** 6 5 8

2 **9** 8 6 5 7

2 9 8 6 5 7

9 **2** 8 6 5 7

9 6 8 2 5 7

Passo 2 (repete: remoção do maior e reestabelecimento da condição de *heap* para novo arranjo)

9 6 8 2 5 7

7 6 8 2 5 | 9

8 6 7 2 5 | 9

5 6 7 2 | 8 9

7 6 5 2 | 8 9

2 6 5 | 7 8 9

6 2 5 | 7 8 9

5 2 | 6 7 8 9

5 2 | 6 7 8 9

2 | 5 6 7 8 9

- Obs.: O algoritmo *HeapSort* possui complexidade $O(n \log n)$ para qualquer cenário.

Algoritmos de ordenação

- Deem uma olhada na URL abaixo. Lá vocês poderão ver animações de alguns algoritmos que vimos em sala:

  `http://coderaptors.com/?All_sorting_algorithms`

Exercícios

1) Faça uma função C++ para implementar o *QuickSort*.

2) O algoritmo *QuickSort* não é estável. Descreva um cenário qualquer que prove esta afirmação.

3) Mostre o resultado do *QuickSort* (do mesmo modo como foi mostrado no exemplo passado) para a lista de números mostrada no exemplo do *ShellSort*.

4) Ainda não vimos relações de recorrência e como resolvê-las, mas mostre mesmo assim uma linha de raciocínio que permita concluir as complexidades mostradas para o melhor e o pior caso do *QuickSort*.

Exercícios

5) Faça uma função C++ para implementar o *HeapSort*.

6) O algoritmo *HeapSort* não é estável. Descreva um cenário qualquer que prove esta afirmação.

7) Mostre o resultado do *HeapSort* (do mesmo modo como foi mostrado no exemplo passado) para a lista de números mostrada no exemplo do *ShellSort*.

8) Pesquise e entenda o funcionamento dos algoritmos de ordenação *RadixSort* e *BucketSort*