INF 112: Programação II

Aula 08

- **Algoritmos de ordenação – parte 2**

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

Algoritmos de ordenação - ShellSort

- O algoritmo de ordenação *shellsort* foi proposto por Ronald L. Shell em 1959 como uma extensão do algoritmo de ordenação por inserção.
- Ele observou que no método de inserção os elementos muito distantes da posição correta podiam levar muitas iterações para chegar até a mesma.
- Ele propôs então, uma forma de acelerar este posicionamento permitindo a troca de posição de elementos que estão distantes.

Algoritmos de ordenação - ShellSort

- Primeiramente é definida uma distância inicial alta *h* e todos os elementos *h*-distantes são ordenados usando o método de inserção.
- Veja que isto é equivalente a ordenar sublistas de tamanhos bem menores que a lista original.
- Neste caso, o algoritmo de inserção é adequado pois o mesmo se sai bem para volumes de dados pequenos.

Algoritmos de ordenação - ShellSort

- Após ordenar as sublistas de elementos *h*-distantes, um novo valor é calculado para *h*, desta vez menor, e o processo de ordenação por inserção de elementos *h*-distantes é reiniciado.

- Note que, como *h* agora é menor, o tamanho das sublistas a serem ordenadas vai aumentar. No entanto, as mesmas já estarão ligeiramente mais ordenadas neste ponto e o algoritmo de inserção, portanto, tente a ser um pouco mais eficiente.

- Segue-se aplicando o raciocínio acima, diminuindo o valor de *h* a cada nova iteração, até que se chegue ao valor 1, onde é feita uma ordenação por inserção comum.

- Neste ponto, aplica-se o algoritmo de inserção para a lista inteira. No entanto, a lista estará com uma ordenação bastante próxima da ordenação final e, neste caso, o algoritmo de inserção é bastante eficiente.

Algoritmos de ordenação - ShellSort

- Para calcular o valor de *h*, Knuth propôs em 1973 a seguinte fórmula obtida experimentalmente:

  $h(s) = 3h(s\text{-}1)+1$, para $s > 1$,

  $h(s) = 1$, para $s = 1$.

- Ainda sobre a sequência de valores para *h*, um valor não deve ser múltiplo do anterior.

Algoritmos de ordenação - ShellSort

- Segue o algoritmo:

```
ALGORITHM   ShellSort(A[0..n - 1])
   h ← 1
   do
      h ← h*3+1
   while h<n

   do
      h ← h DIV 3
      for i ← h to n - 1 do
         v ← A[i]
         j ← i - h
         while j ≥ 0 and A[j] > v do
            A[j+h] ← A[j]
            j ← j - h
         A[j+h] ← v
   while h ≠ 1
```

- Compare o algoritmo de inserção com o trecho acima que vai da linha do *for* até a penúltima linha.

Algoritmos de ordenação - ShellSort

- Exemplo. Seja o arranjo inicial:

  45 78 66 99 24 57 5 15 1 5 28 10 81 8 34

- Valor final de h usando a fórmula de Knuth: 40

  Fazendo ordenação por inserção em subarranjos utilizando um salto (valor de h) por vez:

  $h = 13$

  8 34 66 99 24 57 5 15 1 5 28 10 81 45 78

  $h = 4$

  1 5 5 10 8 34 28 15 24 45 66 99 81 57 78

  $h = 1$

  1 5 5 8 10 15 24 28 34 45 57 66 78 81 99

Algoritmos de ordenação - ShellSort

- A complexidade do *Shellsort* não foi até hoje determinada matematicamente. Para os valores de *h* gerados pela recorrência de Knuth, existem duas conjecturas para o número de comparações realizadas, obtidas empiricamente:
  - $C(n) \in O(n^{1,25})$
  - $C(n) \in O(n(\ln n)^2)$

- Características:
  - O método é relativamente simples de se implementar;
  - O desempenho é relativamente eficiente, próximo dos melhores métodos, embora geralmente um pouco mais lento;
  - Não é estável;
  - Sensível à ordem inicial dos elementos.

Algoritmos de ordenação - MergeSort

- O algoritmo *MergeSort* é um bom exemplo da estratégia denominada como *dividir-para-conquistar*.

- O algoritmo divide o arranjo $A[0..n-1]$ em duas partes $A[0..\lfloor n/2 \rfloor -1]$ and $A[\lfloor n/2 \rfloor..n-1]$, ordena as partes obtidas recursivamente e, ao final, junta (*merge*) as duas partes ordenadas, obtendo a ordenação do arranjo original.

Algoritmos de ordenação - MergeSort

→ Passos em alto nível do algoritmo MergeSort:

- Quebre o arranjo $A[0..n-1]$ em duas partes mais ou menos iguais e copie cada parte para os arranjos *B* e *C*;
- Ordene os arranjos *B* e *C* recursivamente;
- Junte (merge) os arranjos *B* e *C* no arranjo *A* da seguinte forma:
  - Repita os seguintes passos até que um dos arranjos (*B* ou *C*) tenha todos os seus elementos processados:
    - Compare os primeiros elementos nas porções ainda não processadas dos arranjos *B* e *C*;
    - Copie o menor dos dois para a próxima posição em *A* e incrementente o indíce relativo ao arranjo de onde veio o menor valor, que indica também a porção ainda não processada do arranjo que originou o valor.
  - Uma vez que todos os elementos em um dos arranjos foram processados, copie para *A* o restante de elementos não processados do outro arranjo.

Algoritmos de ordenação - MergeSort

- Segue o pseudocódigo:

ALGORITHM *Mergesort*$(A[0..n-1])$

//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
Mergesort$(B[0..\lfloor n/2 \rfloor - 1])$
Mergesort$(C[0..\lceil n/2 \rceil - 1])$
Merge(B, C, A)

Algoritmos de ordenação - MergeSort

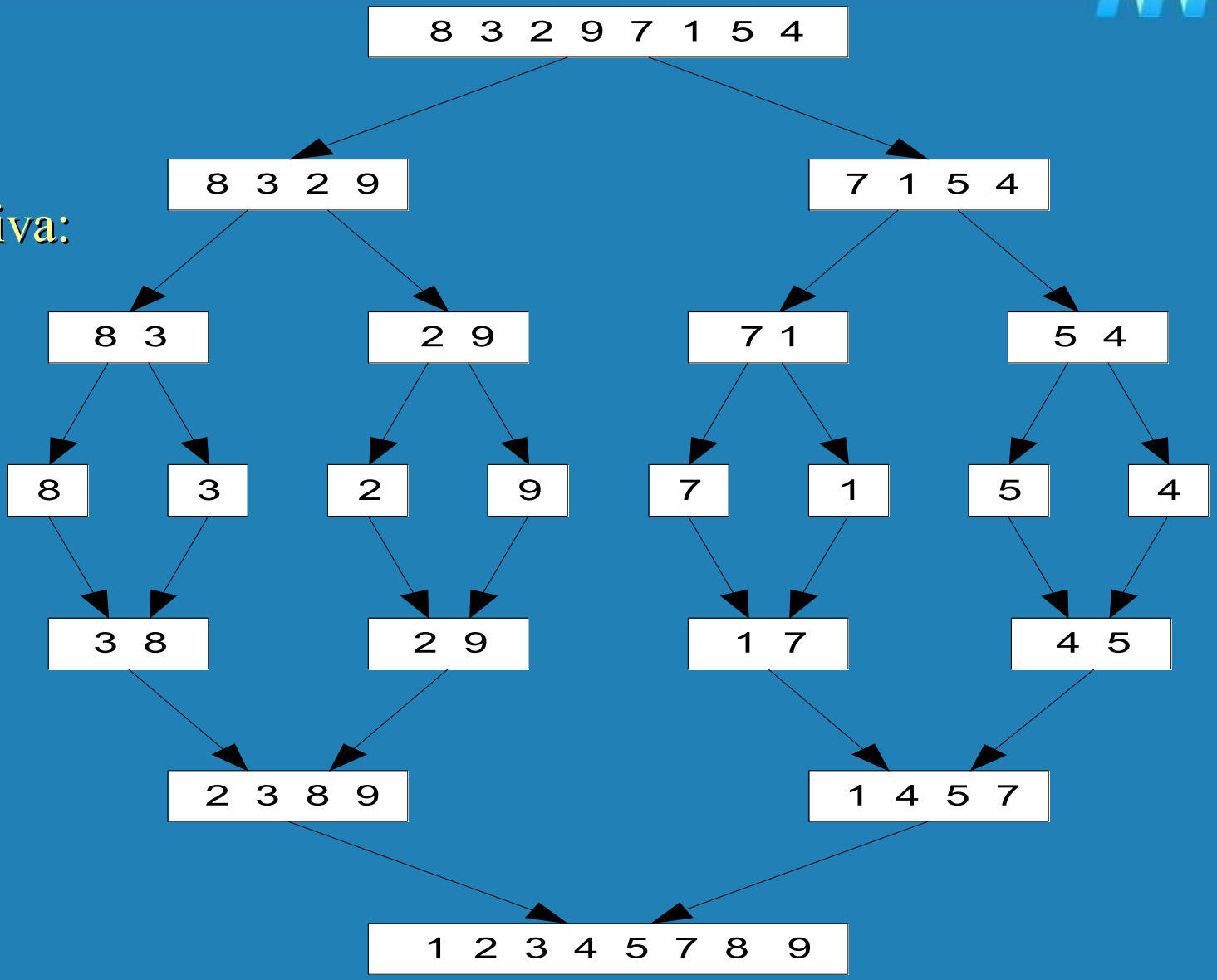
- Pseudocódigo para realizar o "merge":

ALGORITHM $Merge(B[0..p-1], C[0..q-1], A[0..p+q-1])$

//Merges two sorted arrays into one sorted array
//Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
//Output: Sorted array $A[0..p+q-1]$ of the elements of B and C
$i \leftarrow 0;\ j \leftarrow 0;\ k \leftarrow 0$
while $i < p$ **and** $j < q$ **do**
    if $B[i] \leq C[j]$
        $A[k] \leftarrow B[i];\ i \leftarrow i+1$
    else $A[k] \leftarrow C[j];\ j \leftarrow j+1$
    $k \leftarrow k+1$
if $i = p$
    copy $C[j..q-1]$ to $A[k..p+q-1]$
else copy $B[i..p-1]$ to $A[k..p+q-1]$

Algoritmos de ordenação - MergeSort

- Árvore ilustrativa:



Algoritmos de ordenação - MergeSort

- Ainda não aprendemos a analisar algoritmos recursivos. Por enquanto, basta saber que o algoritmo *MergeSort* é $O(n \log n)$ para o número de comparações que realiza.

- Um problema do *MergeSort* é o espaço extra que exije: $O(n)$ (não é *in-place*).

- O *MergeSort* pode ser executado de modo *in-place*, evitando este custo extra de espaço. No entanto, esta versão tem uma constante multiplicativa significativamente maior se comparada ao algoritmo visto.

Exercícios

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

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

3) Faça a mesma análise mostrada no exemplo dado para o *ShellSort*, utilizando a mesma lista de números, mas desta vez para valores de *h* iguais a 7, 5, 3, 1.

4) Por que se recomenda que um valor de *h* não deve ser múltiplo do valor anterior?

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

Exercícios

6) O *MergeSort* é estável?

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