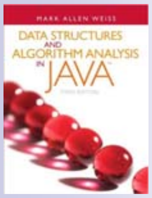
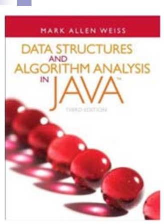


FLORIDA INTERNATIONAL UNIVERSITY

Sorting
(II)

Dr. Antonio L. Bajuelos

FIU School of Computing & Information Sciences



COP-3530 - Data Structures

Module #6: Sorting (part II)

<u>Outline:</u>

- **Shell Sort**
- **Heap Sort**
- **Examples**
- **Complexity analysis and Java code**

Remember… Sorting - the main problem

- Assume we have n **comparable objects** in an **array** and we want to rearrange them to be in **increasing order**

- **Input:**
  - An **array A of n** comparable objects
  - A **key value** in each data object
  - A **comparison function** (consistent and total)

- **Output:**
  - **Reorganize** the objects of A such that:

$$\forall \, \mathbf{i,j \; (if \; i < j \; then \; A[i] \leq A[j])}$$

Classification of Sorting Algorithms

Comparison based **sorting algorithms** may be classified as follows:

- **Elementary** sorting algorithms:
  - Insertion
  - Selection
  - Shell Sort
- **Divide and Conquer** sorting algorithms:
  - Merge-Sort
  - Quick-Sort
- **Priority queues** based sorting algorithm:
  - Heap-Sort

Shell Sort (also called - “Gap” sort).

- Created in **1959** by **Donald Shell****
- **Motivation:**
  - **Shell sort** is a generalization of **insertion sort**, with two observations in mind:
    1. Insertion sort is **efficient** if the input is "almost sorted".
    2. Insertion sort is **inefficient**, on average, because it moves values just one position at a time.

    ⇓ ⇓

    segmented insertion sort.
  - **Shell sort** is easy to develop an intuitive sense of how this algorithm works, but is <u>very difficult to analyze its running time complexity</u>.

****Donald Shell**: “A High-Speed Sorting Procedure”, Communications of the ACM Vol 2, No. 7, 1959, pp. 30-32.

5

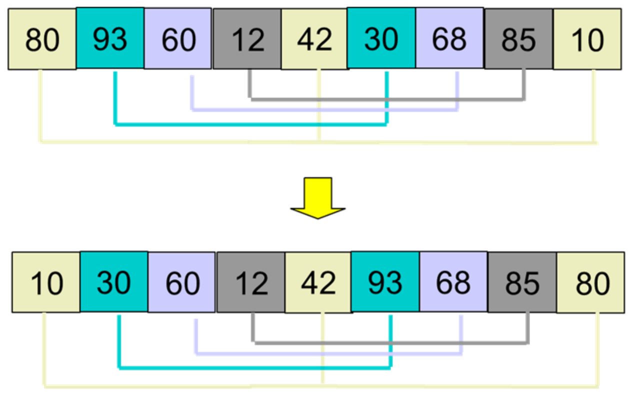
Shell Sort.

- **Algorithm. Main ideas:**
  - Divides an array into several smaller **non-contiguous segments**.
  - The **distance** between successive elements in one segment is called a **gap**.
  - Each segment is sorted within itself using **Insertion sort**.
  - Then re-segment into larger segments (**smaller gaps**) and repeat (insert) sort.
  - Continue until only one segment (**gap = 1**).
  - When the **gap = 1** $\Rightarrow$ **Shell Sort** $\equiv$ **Insertion Sort**,
    - It will be able to work very fast, since **Insertion Sort** is fast when the array is almost in order.

6

Shell Sort. Example

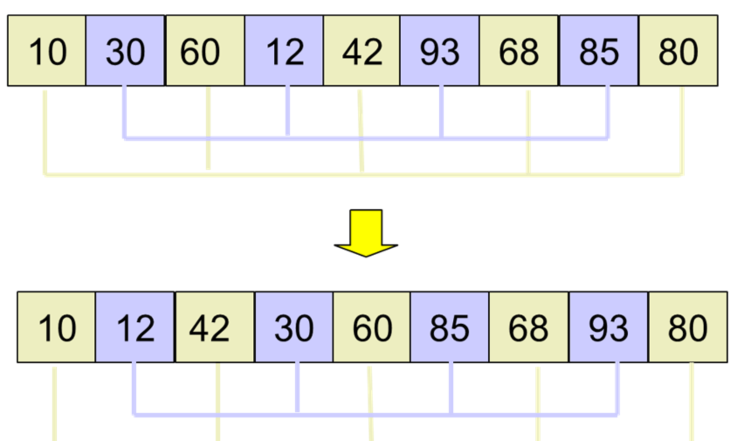
- Initial **Gap = 4**



7

Shell Sort. Example

- **Gap = 2**



8

Shell Sort. Example

- **Gap = 1**

10	12	42	30	60	85	68	93	80

10	12	30	42	60	68	80	85	93

Shell Sort. Gap sequence

- **Important points:**
  - The sequence h_1, h_2, h_3,..., h_t is a sequence of increasing integer values which will be used as a sequence (from right to left) of **gap values**.
  - Any sequence will work as long as it is increasing and $h_1 = 1$.
  - For any **gap value h_k** we have **$A[i] \leq A[i + h_k]$**
  - Best practical results are obtained when all values in the **gap sequence** are **relatively prime** (sequence does not share any divisors).

Shell Sort. Practical Gap sequence

- **Important points (cont…):**
  - Three Methods (for the **gap sequence**):
    1) **Shell's suggestion** - first gap is $n/2$ - successive gaps are previous value divided by 2.
    2) **Odd gaps only** - like **Shell** method except if division produces an even number add 1.
    3) **2.2 method** - like Odd gaps method (add 1 to even division result) but use a divisor of 2.2 and truncate. Best performance of all - most nearly a relatively prime sequence.

Visualization:

https://www.youtube.com/watch?v=CmPA7zE8mx0

http://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

11

Shell Sort. Analysis and Java Code

- **Running Time**:
  - Worst case: $O(n^{1.5})$ **sub-quadratic!**
  - Best case: $O(n \log n)$
  - Average: $O(n^{1.25})$ **conjectured!**
- **Java** code for **Shell Sort**

```
public static void shellSort(Comparable[ ] theArray, int n)
{
    for( int gap = n/2; gap > 0; gap = gap/2 )
      for( int i = gap; i < n; i++ ) {
        Comparable tmp = theArray[ i ];
        int j = i;
        for(; j >= gap && tmp.compareTo(theArray[ j - gap ]) < 0 ; j -= gap)
              theArray[ j ] = theArray[ j - gap ];
        theArray[ j ] = tmp;
      }
}
```

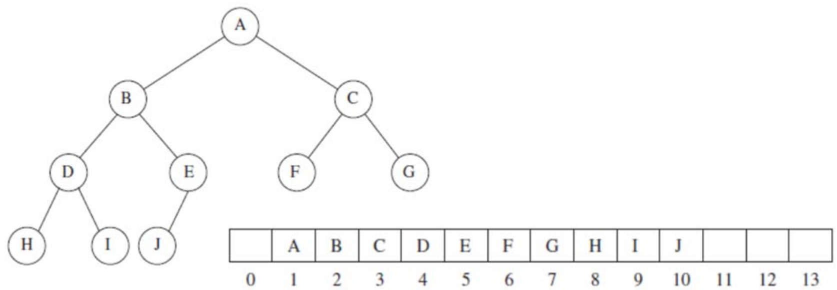
12

HeapSort

- **Heap sort** is a comparison based sorting technique based on **Binary Heap** data structure.

Remember that...

- A **binary min-heap** has:
  - **Structure property:** A complete binary tree – binary tree that is completely filled, with the possible exception of the bottom level, which is filled from left to right.
  - **(min) Heap property**: the key of a node ≤ the keys of the children



13

HeapSort. The algorithm.

- **Input:**
  - Unsorted array A[1..*n*]
- **Algorithm:**
  - **BuildHeap**

    for (i=0; i < A.length; i++)

    B[i]=insert(A[i])
  - **Sort procedure**

    for (i=0; i <B.length; i++)

    A[i]=deleteMin()
- **Output:**
  - Sorted array A[1..*n*]

- **Algorithm Analysis:**
  - **insert()** and **deleteMin()** are **O(log*n*)** then the **overall running time of HeapSort is O(*n*log*n*)**

14

HeapSort

- **Java implementation** (Author: Mark Weiss)
  https://users.cs.fiu.edu/~weiss/dsj2/code/Sort.java

- **Visualization** (in-place version)
  https://www.cs.usfca.edu/~galles/visualization/HeapSort.html

Summary and preliminary results:

algorithm	¿stable?	best time	average time	worst time	extra memory
selectionsort	no	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
insertionsort	yes	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
shellsort	no	$O(n*\log(n))$	$O(n^{1.25})^{\dagger}$	$O(n^{1.5})$	$O(1)$
heapsort	no	$O(n)$	$O(n*\log(n))$	$O(n*\log(n))$	$O(1)$

Non-trivial

- **Stable sorting algorithm** – mean that the algorithm preserves the input order of equal elements in the sorted output.

17