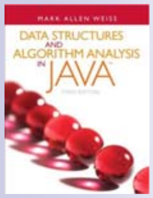
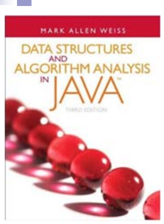


FLORIDA INTERNATIONAL UNIVERSITY

Sorting (III)

Dr. Antonio L. Bajuelos

School of Computing & Information Sciences



COP-3530 - Data Structures

Module #6: Sorting (part III)

Outline:

- **Divide and conquer (D&C) methods**
- **D&C sorting algorithms:**
  - **Merge-Sort**
  - **Examples**
  - **Complexity analysis and Java code**

2

Sorting. The Main Problem (remember)

- **Input:**
  - An array A of n data records (n comparable elements)
  - A key value in each data record
  - A comparison function (consistent and total)
- **Output:**
  - Reorganize the elements of A such that

    $\forall$ i,j, if i<j $\Rightarrow$ A[i] $\leq$ A[j]
- **Alternate way of saying this:**
  - **Given:** An unsorted Array
  - **Goal:** Sort it

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 <u>input order of equal elements in the sorted output</u>.

Divide and Conquer Methods

- **Divide and Conquer** is a very important **technique** in **algorithm design**.
- **Main Idea:**
  - **Divide** problem into sub-problems.
  - **Conquer** by solving sub-problems recursively. If the sub-problems are small enough, solve them in brute force approach.
  - **Combine** the solutions of sub-problems into a solution of the original problem.

5

Divide and Conquer Sorting Algorithms

- **Merge-Sort**
  - **Sort** the left half of the elements (recursively)
  - **Sort** the right half of the elements (recursively)
  - **Merge** the two sorted halves into a sorted whole
- **Quick-Sort**
  - Pick a "**pivot**" element
  - **Divide** elements into less-than pivot and greater-than pivot
  - **Sort** the two divisions (recursively on each)
  - **Combine** by doing nothing. Once the conquer step recursively sorts, we are done. All elements to the left of the pivot, are less than or equal to the pivot and are sorted, and all elements to the right of the pivot are greater than the pivot and are sorted.

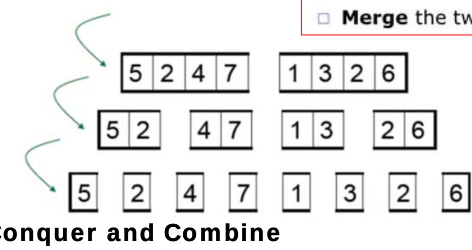
6

Merge-Sort. Example

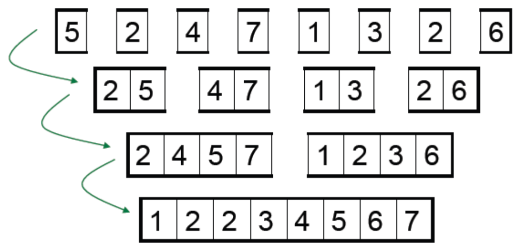
5 2 4 7 1 3 2 6

Mergesort
- **Sort** the left half of the elements (recursively)
- **Sort** the right half of the elements (recursively)
- **Merge** the two sorted halves into a sorted whole

- **Divide**

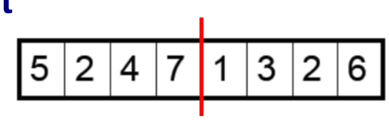


- **Conquer and Combine**



7

Merge-Sort



- **Formalizing the algorithm:**
  - To **sort the array** from position **left** to position **right**:

    if range is 1 element long, it is already sorted! (Base case)

    else:
    - **Sort** from **left** to **(right + left) / 2**
    - **Sort** from **(right + left) / 2** to **right**
    - **Merge** the two halves together

8

Merge-Sort. The Merge step

- The fundamental operation is **merging two sorted lists.**
- **Merge-Sort** uses extra space proportional to n.

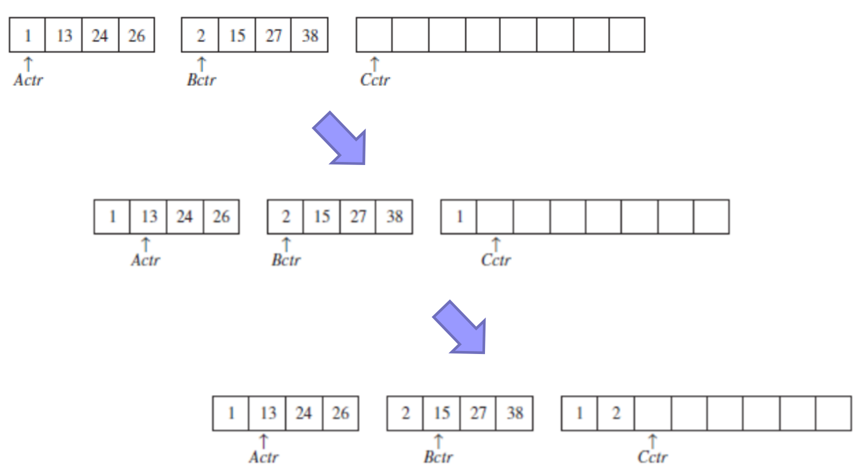
  Some facts:
  - (not hard). Use raux[] array of length ~ ½ N instead of N.
  - (very hard). in-place version of Merge-Sort.

- **How to merge two sorted array?**
  - The basic merging algorithm takes two input arrays ***A*** and ***B***, an **output** array ***C***, and three counters: ***Actr***, ***Bctr***, and ***Cctr***
  - ***Actr***, ***Bctr*** *and* ***Cctr*** *are i*nitially set to the beginning of their respective arrays.
  - The smaller of ***A*[*Actr*]** and ***B*[*Bctr*]** is copied to the **next entry** in ***C***, and the appropriate counters are advanced.
  - When either input list is exhausted, the **remainder of the other list is copied to *C***.

9

Merge-Sort. The Merge step

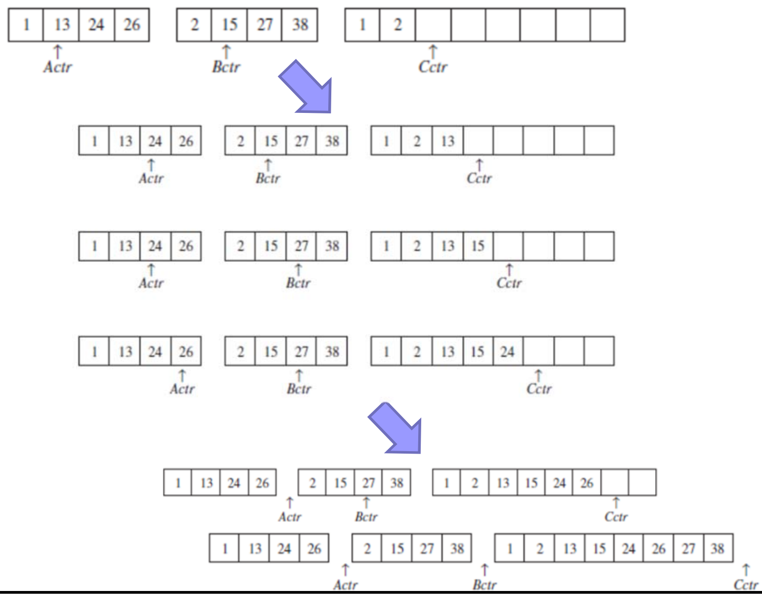
- **Example #1:**



10

Merge-Sort. The Merge step

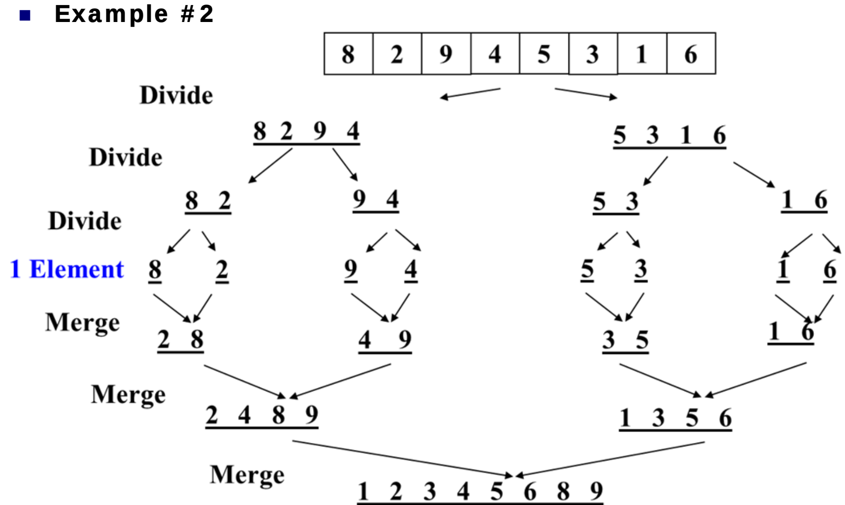
- **Example #1 (cont…):**



11

Merge-Sort. The Merge step

- **Example #2**



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

12

Merge-Sort. Algorithm Analysis

- To sort an array of n elements, we have:
  - **Step 1:** If the problem size is small, solve this problem directly; otherwise, **split the original problem into 2 sub-problems with equal sizes**.
  - **Step 2: Recursively** solve these **2 sub-problems** by applying this algorithm.
  - **Step 3: Merge** the solutions of the **2 sub-problems** into a solution of the original problem.
- So, our recurrence relation is:
  - $\mathbf{T(1) = c_1}$
  - $\mathbf{T(n) = 2T(n/2) + c_2 n}$

Merge-Sort. Analysis (cont...)

- Our recurrence relation for the **Mergesort** is:
  - $\mathbf{T(1) = c_1}$
  - $\mathbf{T(n) = 2T(n/2) + c_2 n}$
- Assume that $c_1 = c_2 = 1$ (not affect the asymptotic behavior):

$T(1) = 1$

$T(n) = 2T(n/2) + n$

$= 2(2T(n/4) + n/2) + n$

$= 4T(n/4) + 2n$

$= 4(2T(n/8) + n/4) + 2n$

$= 8T(n/8) + 3n$

...

$= 2^k T(n/2^k) + kn$

$T(n) = 2^k T(n/2^k) + kn$

Assume than $n = 2^k$ then

$n/2^k = 1$, i.e., $\log n = k$

$T(n) = 2^{\log_2 n} T(1) + n \log n$

$= n + n \log n$

$\mathbf{T(n) = O(n \log n)}$

- Recall that we have assumed $N = 2^k$. The analysis can be refined to handle cases when N is not a power of 2. The answer turns out to be almost identical.

Insertion Sort vs Merge-Sort*

- **Assume that:**
  - **home** PC executes 10^8 compares/second.
  - **super**computer executes 10^{12} compares/second

	insertion sort (N^2)			mergesort (N log N)		
computer	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min
super	instant	1 second	1 week	instant	instant	instant

Good algorithms are better than supercomputers!

*from http://algs4.cs.princeton.edu/lectures/22Mergesort.pdf

15

Merge-Sort. Java Code

- See **Java Code** of **Merge-Sort** in:

  http://users.cis.fiu.edu/~weiss/dsj2/code/Sort.java
  (Author: Mark Weiss)

- **Recommended material:**

  http://algs4.cs.princeton.edu/lectures/22Mergesort.pdf

16