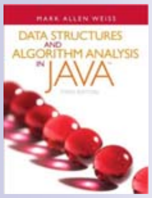
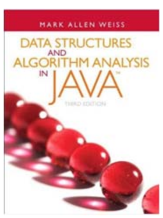


FLORIDA INTERNATIONAL UNIVERSITY

Sorting
(I)

Dr. Antonio L. Bajuelos

School of Computing & Information Sciences



COP-3530 - Data Structures

Module #6: Sorting (part I)

Outline:

- **Why sorting?**
- **Some applications of sorting**
- **Classification of the sorting algorithms**
- **Insertion Sort**
- **Selection Sort**
- **Complexity analysis and Java code**

Why sorting?

"When in doubt, sort"

- **Sorting** – one of the principles of algorithm design. Sorting used as a subroutine in many of the in many of the algorithms.
- **Sorting is ordering a list of objects.**
- Two **types of sorting**:
  - **internal sorting** - if the number of objects is small enough to fits into the main memory;
  - **external sorting** - if the number of objects is so large that some of them reside on external storage during the sort

3

Application of Sorting?

- **Searching** - **Binary search** lets you test whether an item is in a dictionary/array in O(log n) time.
- **Closest pair problem** - Given n numbers, find the pair which are closest to each other. Once the numbers are sorted, the closest pair will be next to each other in sorted order, so an O(n) linear scan completes the job.
- **Duplicates?** - Given a set of n items, are they all unique or are there any duplicates? Sort them and do a linear scan to check all adjacent pairs. This is a special case of closest pair above.
- . . .

4

The Problem of Finding the Maximun

- Consider, for example, the problem of **finding the maximum** item in an **array** of items.
- To determine order, we can use (in **Java**) the **compareTo** method that we know must be available for all **Comparables.**

$$x.compareTo(y) = \begin{cases} < 0; & if\ x < y \\ = 0; & if\ x = y \\ > 0; & if\ x > y \end{cases}$$

```
public static Comparable findMax( Comparable[] arr )
{
        int maxIndex = 0;
        for( int i = 1; i < arr.length; i++ )
           if( arr[i].compareTo( arr[maxIndex] ) > 0 )
                maxIndex = i;
        return arr[maxIndex];
}
```

5

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:

    $\forall$ i,j (if i < j then A[i] $\leq$ A[j])

6

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

7

Insertion Sort

- **Main idea:**
  - At step k, put the k^{th} element in the correct position among the first k elements

- Alternate way of saying this:
  - Sort first two elements
  - Now insert 3^{rd} element in order
  - Now insert 4^{th} element in order
  - . . .
- The **insert** operation means:
  - Insert the element A[i] into sorted array A[0: i-1] by pairwise swap down to the correct position

Original	34	8	64	51	32	21	Positions Moved
After $p = 1$	8	34	64	51	32	21	1
After $p = 2$	8	34	64	51	32	21	0
After $p = 3$	8	34	51	64	32	21	1
After $p = 4$	8	32	34	51	64	21	3
After $p = 5$	8	21	32	34	51	64	4

8

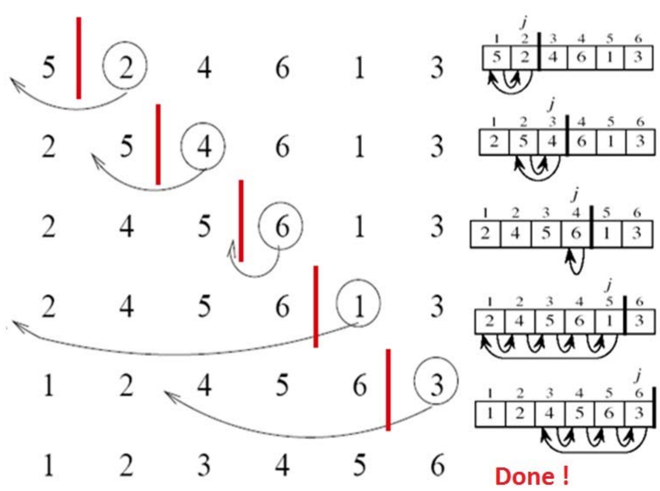
Insertion Sort. Java Implementation

```
public static <AnyType extends Comparable<? super AnyType>>
      void insertionSort( AnyType [] a )
{
     int j;
     for( int p = 1; p < a.length; p++ )
     {
        AnyType tmp = a[p];
        for( j = p; j > 0; j-- )
            if (tmp.compareTo( a[j - 1] ) >= 0)    // Comparison
               break;
            else
               a[j] = a[j - 1];                    // Flip or Inversion
        a[j] = tmp;
      }
}
```

Note that in **insertion sort algorithm**, the elements are inserted into the sorted section, while in **bubble sort** the maximums are bubbled out of the unsorted section.

9

Insertion Sort. Example



10

Insertion Sort. Analysis

- **Number of comparisons:**

$B_C(n) = n$; $W_C(n) = \sum_{k=0}^{n-1} k = \frac{n(n-1)}{2}$; $A_C(n) = \frac{1}{2}\sum_{k=0}^{n-1} k = \frac{n(n-1)}{4}$;

- **Number of exchanges** (**inversions or flips**) between adjacent elements:
  - **Best case** (pre-sorted array):

    $B_E(n) = 0$
  - **Worst case** (reverse order):

    $W_E(n) = \sum_{k=0}^{n-1} k = \frac{n(n-1)}{2}$
  - **Average case**:

    $A_E(n) = \frac{1}{2}\sum_{k=0}^{n-1} k = \frac{n(n-1)}{4}$

- Best-case: **O(n)** Worst-case**: O(n²)** Average-case: **O(n²)**

Visualization:

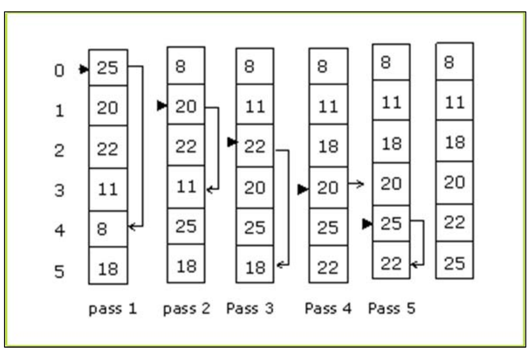
http://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

11

Selection Sort.

- **Main idea:**
  - At step k, find the smallest/largest element among the not-yet-sorted elements and put it at position k

- Alternate way of saying this:
  - Find smallest element, put it 1st
  - Find next smallest element, put it 2nd
  - Find next smallest element, put it 3rd ...



12

Selection Sort. Example Java Code

```
public static void swapReferences(Comparable[] a, int ind1, int ind2)
{
        int tmp = a[ind1];
        a[ind1] = a[ind2];
        a[ind2] = tmp;
}
public static void selectionSort(Comparable[] A )
{
    int i;
    for( i = A.length-1; i > 0; i-- ) {
      // find maximum value in A[0..i]
      int maxIndex = 0;
      int j;
      for( j = 1; j < i; j++ ) {
        /*  inner loop invariant: for all k < j, A[maxIndex] >= A[k]  */
        if (A[maxIndex].compareTo(A[j]) < 0 )
            maxIndex = j;
      }
      /*  swap largest (A[maxIndex]) into A[i]  */
      swapReferences( A, i, maxIndex );
    }
  }
```

13

Selection Sort. Analysis

- **Number of comparisons:**

  $B_C(n) = W_C(n) = A_C(n) = \sum_{k=0}^{n-1} k = \frac{n(n-1)}{2}$

- **Number of exchanges** (**inversions**) between adjacent elements:

  $B_E(n) = W_E(n) = A_E(n) = n-1$

- Best-case: **$O(n^2)$** Worst-case: **$O(n^2)$** Average-case: **$O(n^2)$**

- **Important point:**
  - For **selection sort** method that makes recursive calls:

    $T(1) = 1$ and **$T(n) = n + T(n-1)$**

Visualization:

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

14

Insertion vs. Selection Sort

- Two different algorithms to solve the same problem.
- They have the same **worst-case** and **average-case** asymptotic complexity
- **Insertion sort** has better best-case complexity; preferable when input is "mostly sorted".
- **Insertion sort** may do well on small arrays.

- Some Animations:

  Insertion Sort

  https://www.youtube.com/watch?v=ROalU379l3U

  Selection Sort

  https://www.youtube.com/watch?v=Ns4TPTC8whw