# CSIS 3103

## Ch 8: Sorting

## Sorting

- Probably the most extensively studied problem in computer science
- Many sorting algorithms exist
- Applications range from
  - simple in-memory sorting of a small collection of integers
  - sorting massive sets of records in databases involving external storage and multiple processors
- We will look at a small sample of the known sorting algorithms

## An Invariant for Sorting

A list of elements, *A,* is sorted (in ascending order) if

For all *i, j* in 0..*A.length* – 1: $i < j \Rightarrow A[i] \leq A[j]$

## Using Java Sorting Methods

- Java API provides a class `Arrays` with several overloaded `sort` methods for different array types
- The `Collections` class provides similar sorting methods
  - Sorting methods for arrays of primitive types are based on the quicksort algorithm
  - Sorting methods for arrays of objects and `Lists` are based on mergesort

## Java Sorting Methods

| Method sort in Class Arrays | Behavior |
|---|---|
| public static void sort(int[] items) | Sorts the array items in ascending order. |
| public static void sort(int[] items, int fromIndex, int toIndex) | Sorts array elements items[fromIndex] to items[toIndex] in ascending order. |
| public static void sort(Object[] items) | Sorts the objects in array items in ascending order using their natural ordering (defined by method compareTo). All objects in items must implement the Comparable interface and must be mutually comparable. |
| public static void sort(Object[] items, int fromIndex, int toIndex) | Sorts array elements items[fromIndex] to items[toIndex] in ascending order using their natural ordering (defined by method compareTo). All objects must implement the Comparable interface and must be mutually comparable. |
| public static <T> void sort(T[] items, Comparator<? super T> comp) | Sorts the objects in items in ascending order as defined by method comp.compare. All objects in items must be mutually comparable using method comp.compare. |
| public static <T> void sort(T[] items, int fromIndex, int toIndex, Comparator<? super T> comp) | Sorts the objects in items[fromIndex] to items[toIndex] in ascending order as defined by method comp.compare. All objects in items must be mutually comparable using method comp.compare. |
| **Method sort in Class Collections** | **Behavior** |
| public static <T extends Comparable<T>> void sort(List<T> list) | Sorts the objects in list in ascending order using their natural ordering (defined by method compareTo). All objects in list must implement the Comparable interface and must be mutually comparable. |
| public static <T> void sort (List<T> list, Comparator<? super T> comp) | Sorts the objects in list in ascending order as defined by method comp.compare. All objects must be mutually comparable. |

## Declaring a Generic Method

SYNTAX **Declaring a Generic Method**

FORM:

*methodModifiers* <*genericParameters*> *returnType* *methodName*(*methodParameters*)

EXAMPLE:

```
public static <T extends Comparable<T>> int binarySearch(T[] items,
                                                          T target)
```

MEANING:

To declare a generic method, list the *genericParameters* inside the symbol pair <> and between the *methodModifiers* (e.g., public static) and the return type. The *genericParameters* can then be used in the specification of the *methodParameters*.

## Selection Sort

A relatively simple algorithm that sorts an array by making passes through the array, selecting the smallest remaining item and placing it where it belongs in the array

– Efficiency is $O(n^2)$

## Selection Sort

Basic rule: on each pass select the smallest remaining item and place it in its proper location


## Selection Sort Algorithm

1. **for** fill = 0 to n – 2 do
2. Set `posMin` to the subscript of the smallest item in the subarray starting at subscript `fill`
3. Exchange the item at `posMin` with the one at `fill`

<u>Refining Step 2</u>

**2.1** **for** next = fill + 1 to n – 1 do

**2.2** **if** the item at `next` is less than the item at `posMin`

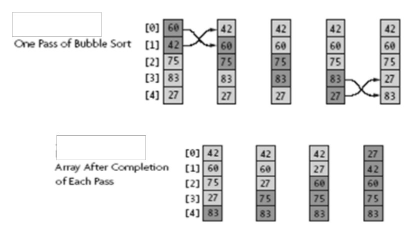
2.3 Reset `posMin` to next

Number of comparisons is $O(n^2)$

Number of exchanges is $O(n)$

## Bubble Sort

Compares adjacent array elements and exchanges their values if they are out of order




## Analysis of Bubble Sort

- Very poor performance in most cases
- Works best when array is nearly sorted to begin with
- Worst case number of comparisons is $O(n^2)$
- Worst case number of exchanges is $O(n^2)$
- Best case occurs when the array is already sorted: $O(n)$ comparisons and $O(1)$ exchanges

## Insertion Sort

Based on the technique commonly used to arrange a hand of cards

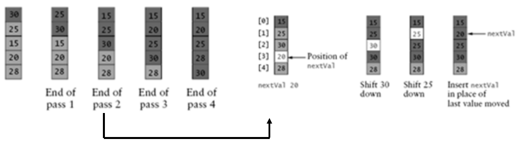
– Player keeps the cards that have been picked up so far in sorted order

– When the player picks up a new card, he makes room for the new card and inserts it in its proper place

## Insertion Sort Algorithm

For each array element from the second (`nextPos = 1`) to the last

- Insert the element at `nextPos` where it belongs in the array, increasing the length of the sorted subarray by 1




## Analysis of Insertion Sort

- Maximum number of comparisons is $O(n^2)$
- Best case number of comparisons is $O(n)$
- The number of shifts performed during an insertion is one less than or the same as the number of comparisons
- A shift in insertion sort requires the moving only one item whereas in bubble or selection sort an exchange involves a temporary item and requires the movement of three items

## Comparison of Quadratic Sorts

| | Number of Comparisons | | Number of Exchanges | |
|---|---|---|---|---|
| | Best | Worst | Best | Worst |
| Selection sort | $O(n^2)$ | $O(n^2)$ | $O(n)$ | $O(n)$ |
| Bubble sort | $O(n)$ | $O(n^2)$ | $O(1)$ | $O(n^2)$ |
| Insertion sort | $O(n)$ | $O(n^2)$ | $O(n)$ | $O(n^2)$ |

| $n$ | $n^2$ | $n \log n$ |
|---|---|---|
| 8 | 64 | 24 |
| 16 | 256 | 64 |
| 32 | 1,024 | 160 |
| 64 | 4,096 | 384 |
| 128 | 16,384 | 896 |
| 256 | 65,536 | 2,048 |
| 512 | 262,144 | 4,608 |

## Comparison of Quadratic Sorts

- Insertion sort
  - gives the best performance for most arrays
  - takes advantage of any partial sorting in the array and uses less costly shifts
- Bubble sort generally gives the worst performance—unless the array is nearly sorted
- None of the quadratic search algorithms are very good for large arrays ($n > 1000$)
- The best sorting algorithms provide $n \log n$ average case performance

## Comparisons versus Exchanges

- In Java objects, an exchange requires a switch of two object references using a third object reference as an intermediary
- A comparison requires an execution of a `compareTo` method
  - The cost of a comparison depends on its complexity, but is generally more costly than an exchange
- For some languages (and primitives in Java), an exchange may involve physically moving information rather than swapping object references. In these cases, an exchange may be more costly than a comparison