Chapter 9

Sorting and searching arrays

Lecture slides for:

Java Actually: A Comprehensive Primer in Programming

Khalid Azim Mughal, Torill Hamre, Rolf W. Rasmussen

Cengage Learning, 2008.

ISBN: 978-1-844480-933-2

`http://www.ii.uib.no/~khalid/jac/`

Modified: 18/2/18

Overview

Pseudocode

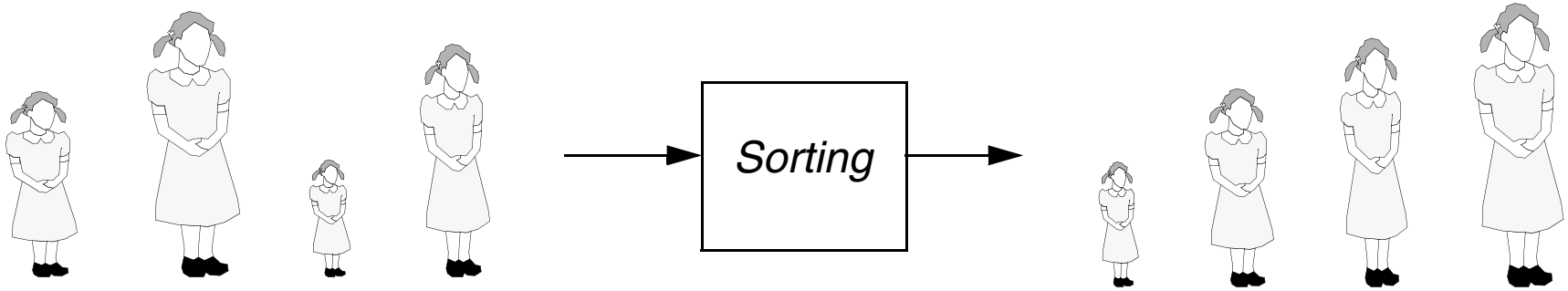
Natural order

- Relational operators for primitive data types
- Comparing floating-point values

Sorting arrays of integers

- Selection sort
- Insertion sort
- Bubble sort

- Sorting arrays of objects
  - The `Comparable` interface
- Searching
  - Linear search
  - Binary search
- Sorting and searching using the Java standard library



Pseudocode

- Pseudocode describes an algorithm (i.e. steps for solving a problem) in *plain text*.
- Control flow is specified by *reserved words*, e.g. `if` to start a selection statement.
- The structure of the actions in the algorithm is shown by *indentation*.
- Example: bubble sort (see Programming Exercise 8)

```
For each pass of the array:
    For each pair of neighbouring values in the unsorted part of the array:
        If values in the pair are in the wrong order
            Swap the values in the pair
```

- Pseudocode is written *by* and *for* programmers.
- It is a useful tool when developing algorithms.
- Complicated steps can be further refined until the problem is sufficiently well understood to begin coding.
- Besides, pseudocode can be written in the source code, and thereby be a part of the documentation of the program.

Natural order

- Values of numerical primitive data types have a *natural order*.
- Hence, we can always *rank* two values `a` and `b` of type `byte`, `short`, `int`, `long`, `float` and `double`. I.e. we can determine if value `a` is
  - *smaller than*,
  - *equal to*, or
  - *greater than*

  value `b`.
- Characters (type `char`) also have a natural order.
- Boolean values, however, have no natural order. We can compare two boolean values, `b1` and `b2`, for equality, but is doesn't make sense to say that `b1` is smaller than or greater than `b2`.

Relational operators for primitive data types

- Numerical values and characters can be compared using *relational operators*:

  == (equal to)

  != (not equal to)

  < (less than)

  > (greater than),

  <= (less than or equal to)

  >= (greater than or equal to)

- Examples:

```
int score;
... // initialise score
assert score >= 0;   // verify that score is valid
assert score <= 100;
if (score < 40) {
    grade = 'F';
} else if (score <= 50) {
    grade = 'E';
} ...
```

Comparing floating-point values

- Comparing integer values and characters for equality poses no problems:

```
int score;
char grade;
... // initialise score and grade
if (score == 0) {
    System.out.println("You REALLY have to study harder!!!");
}
if (grade == 'A') {
    System.out.println("Excellent! You got the highest grade.");
}
```

- But, take care when comparing floating-point values for equality:

```
final double ONE_THIRD = 0.333333333333;
double ratio = 1.0/3.0;
assert ratio == ONE_THIRD :                                    // (1)
    "ERROR: " + ratio + " is different from " + ONE_THIRD;
```

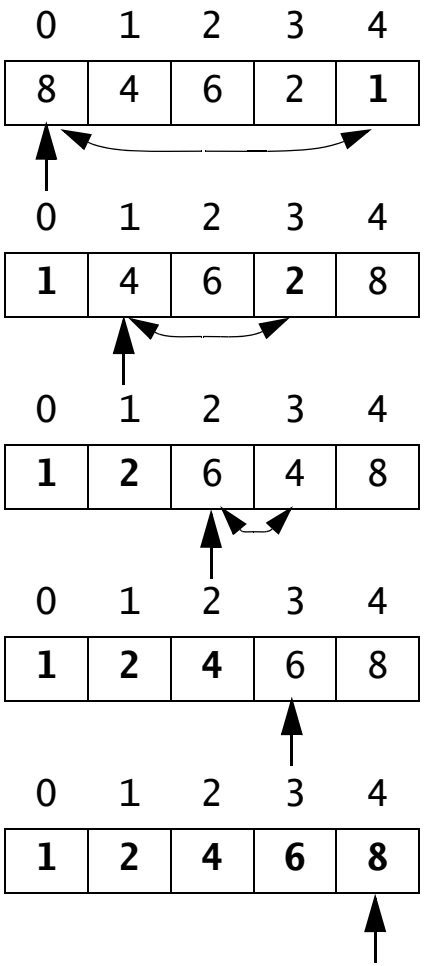
  – When run with assertions enabled, a similar error message will be printed:

```
Exception in thread "main" java.lang.AssertionError: ERROR: 0.3333333333333333
is different from 0.333333333333
        at TestDecimalEquality.main(TestDecimalEquality.java:7)
```

  – See the book for a solution to such problems.

Selection sort

Array a:



- find smallest in the whole array.
- swap a[0] with the smallest.

- find smallest from a[1] to a[4].
- swap a[1] with the smallest.

- find smallest from a[2] to a[4].
- swap a[2] with the smallest.

etc.

- continue until the whole array is sorted.

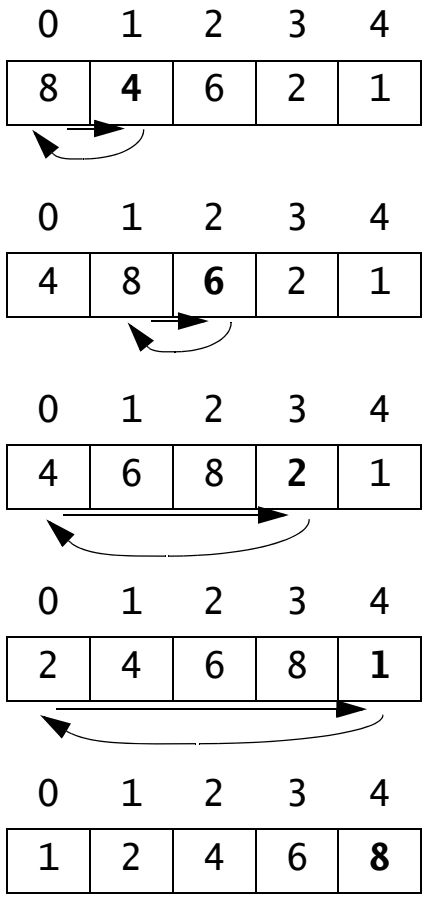
```
int a[] = {8,4,6,2,1};

// For each pass:
for (int ind = 0;
     ind < a.length - 1;
     ind++) {

    // Finn smallest value in unsorted
    // part of the array
    int ind_of_smallest = ind;
    for (int k = ind + 1; k < a.length; k++)
        if (a[k] < a[ind_of_smallest])
            ind_of_smallest = k ;
    // Swap the smallest value and the first
    // element in usorted part of the array
    int temp = a[ind_of_smallest];
    a[ind_of_smallest] = a[ind];
    a[ind] = temp;
}
```

Insertion sort

Array a:



Place value a[1] in correct position, so that a[0]≤ a[1].

Place value a[2] in correct position, so that a[0]≤ a[1] ≤ a[2].

Place value a[3] in correct position so that

a[0]≤ a[1] ≤ a[2] ≤ a[3].

Place value a[4] in correct position so that

a[0]≤ a[1] ... ≤ a[4].

for (i = 1; i < a.length; i++)

- place value *a[i]* in correct position *j* among the elements *a[0]...a[i].*

---

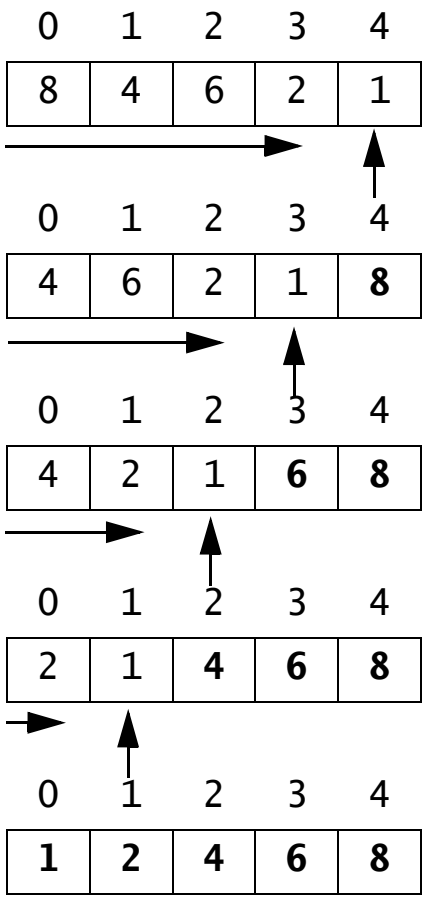
for (i = 1; i < a.length; i++)

- find correct position *j*, $0 \le j \le i$, for value *a[i]* by shifting values of *a[j]...a[i-1]* one position to the right.
- place value *a[i]* in position *j*.

```
int a[] = {8,4,6,2,1};
// For each pass:
for (int i = 1; i < a.length; i++) {
    // Keep next value to be inserted
    int value = a[i];
    // Shift all values in the sorted part
    // of the array greater than this
    // one position backwards in the array.
    int j;
    for (j = i;
         (j > 0 && value < a[j-1]);
         j--)
        a[j] = a[j-1];  // shift backwards!
    a[j] = value; // insert value
                  // in correct position!
}
```

Bubble sort

Array a:



Given n = number of elements -1, i.e. 4.

Pass 1:
for k = 0(1)n-1:
if $element_k$ is greater than $element_{k+1}$,
swap these elements.

Pass 2:
for k = 0(1)n-2:
if $element_k$ is greater than $element_{k+1}$,
swap these elements.

Pass 3:
for k = 0(1)n-3:
if $element_k$ is greater than $element_{k+1}$,
swap these elements.

Pass 4:
for k = 0(1)n-4:
if $element_k$ is greater than $element_{k+1}$,
swap these elements.

Number of passes equals n, i.e. number of elements -1.

$a[0] \leq a[1] \leq \ldots \leq a[n]$

```
int a[] = {8,4,6,2,1};

// For each pass:
for (int i = a.length -1;
     i > 0;
     i--) {
    // For each neighbouring pair
    // in usorted part of array
    for (int k = 0; k < i; k++)
        // If values in
       // neighbouring pairs are
        // in the wrong order:
        // Swap values in
        // neighbouring pairs
        if (a[k] > a[k+1]) {
            int temp = a[k];
            a[k] = a[k+1];
            a[k+1] = temp;
        }
}
```

This algorithm can be improved.

Natural order for objects

- *Natural order* for values of type `T`:
  - Two values `a` and `b` of a type `T` can be compared to determine if `a` is *less than*, *equal to* or *greater than* `b`.
- Objects are *compared* for equality using the == operator (reference equality) and the `equals()` method (object equality).
  - The `equals()` method must be *overridden* to provide a suitable implementation for testing object equality.
  - Take care not to *overload* the `equals()` method!
- The *ranking* of objects is determined by the `compareTo()` method, which is specified in the `Comparable` interface.

The `Comparable` interface

- The *ranking* of objects is determined by the `compareTo()` method, which is specified in the `Comparable` interface.

```
interface Comparable {
    int compareTo(Object objRef);
}
```

- The call `obj1.compareTo(obj2)` will return an integer that is:
  - == 0 if `obj1` and `obj2` are equal
  - < 0 if `obj1` is before `obj2`
  - > 0 if `obj1` is after `obj2` in the natural order.
- All classes whose objects need a *natural order* must implement this interface.
- The implementation of the `compareTo()` method must be consistent with the implementation of the `equals()` method in the class.

Sorting arrays of objects

- A class must implement the `Comparable` interface, if its objects are to have a natural order.
- The `String` class implements the `Comparable` interface, so do the wrapper classes for primitive values.

```
// Utility class that sorts arrays of Comparable objects.
  public static void bubbleSort(Comparable[] a) {
    boolean sorted = false;
    for (int i = a.length -1; (i > 0) && (!sorted); i--) {
      sorted = true;
      System.out.println("Pass: " + (a.length - i));
      for (int k = 0; k < i; k++)
        if (a[k].compareTo(a[k+1]) > 0) {
          Comparable temp = a[k]; // swap neighbouring elements
          a[k] = a[k+1];
          a[k+1] = temp;
          sorted = false;
        }
    }
  }
```

Sorting arrays of objects

- A client using the utility class to sort an array of integer objects:

```
public class BubbleSortClient {
  public static void main(String args[]) {
    Integer intArray[] = {8,4,6,2,1};

    System.out.println("Before sorting:");
    for (int number : intArray)
      System.out.print(number + " ");
    System.out.println();

    GenericUtility.bubbleSort(intArray);

    System.out.println("After sorting:");
    for (int number : intArray)
      System.out.print(number + " ");
    System.out.println();
  }
}
```

Sorting arrays of objects

- Running the program

```
Before sorting:
8 4 6 2 1
Pass: 1
Pass: 2
Pass: 3
Pass: 4
After sorting:
1 2 4 6 8
```

Sorting arrays of objects

- Insertion sort for an array of objects:

```
public static void sortByInsertion(Comparable[] compArray) {
  // For each pass:
    for (int next=1; next<compArray.length; next++) {  // outer loop
        // Store the next element to be inserted in an auxiliary variable
        Comparable value = compArray[next];
        // Iterate through the sorted subarray backwards
        //   If the current element is larger than the next element
       //    Shift the current element one position towards the end of the array
        int current;
        for (current=next;                                  // inner loop
             current>0 && value.compareTo(compArray[current-1])<0;
             current--) {
            compArray[current] = compArray[current-1]; // shift backwards
        }
        compArray[current] = value;
        printArray("Pass #" + next + ": ", compArray);
    }
}
```

- Exercise: Write a client program that uses the method above to sort the array

```
String strArray[] = {"joe", "albert", "john", "bart", "donald" };
```

Linear search

Problem: Find a given *value* (if it exists!) in an array. The value we are looking for is called a *key*.

Pseudocode:

Repeat while there are elements left to compare in the array:

If the current element equals the search key

Return the index of the current element

If the search key is not found after searching the entire array

Return a negative index to indicate that the key was not found

Linear search

- Example: Searching in an array of integers.

```
int a[] = {8,4,6,2,1};
int key = 6;
boolean found = false;
int counter = 0;
while (counter < a.length && !found) {
    if (a[counter] == key)
        found = true;   // key is found
    else
        counter++;      // otherwise, continue with next element in the array
}
assert (counter >= a.length || found) : "Error in linear search";
if (found)
    System.out.println("Value " + key +
                       " found in element with index " + counter);
else
    System.out.println("Value " + key + " is not in the array!");
```

- The key can be in any element since the array is unsorted;
  this demands (N_ELEMENTS/2) comparisons on average.

Linear search for objects

- Linear search method from the `GenericUtility` class:

```
static Comparable linearSearch(Comparable[] objects, Comparable key) {
boolean found = false;
int i = 0;
// Repeat while more elements in array:
while (i < objects.length && !found) {  // (1)
  // If the current element contains the key:
  //   Return the index of the current element
  if (objects[i].compareTo(key) == 0) {
    found = true;
  }
  else {
   i++;      // Move to next element
  }
}
assert (i >= objects.length || found) : "Error in linear search.";
if (found)
  return i;  // The key is in element objects[i]
else
  return -1; // The key is not found
}
```

Binary search

Problem: Find a given value (if it exists!) in a *sorted array*

Pseudocode:

Repeat while there are elements left to compare in the array:

If the middle element is equal to the key

Return the index of the middle element

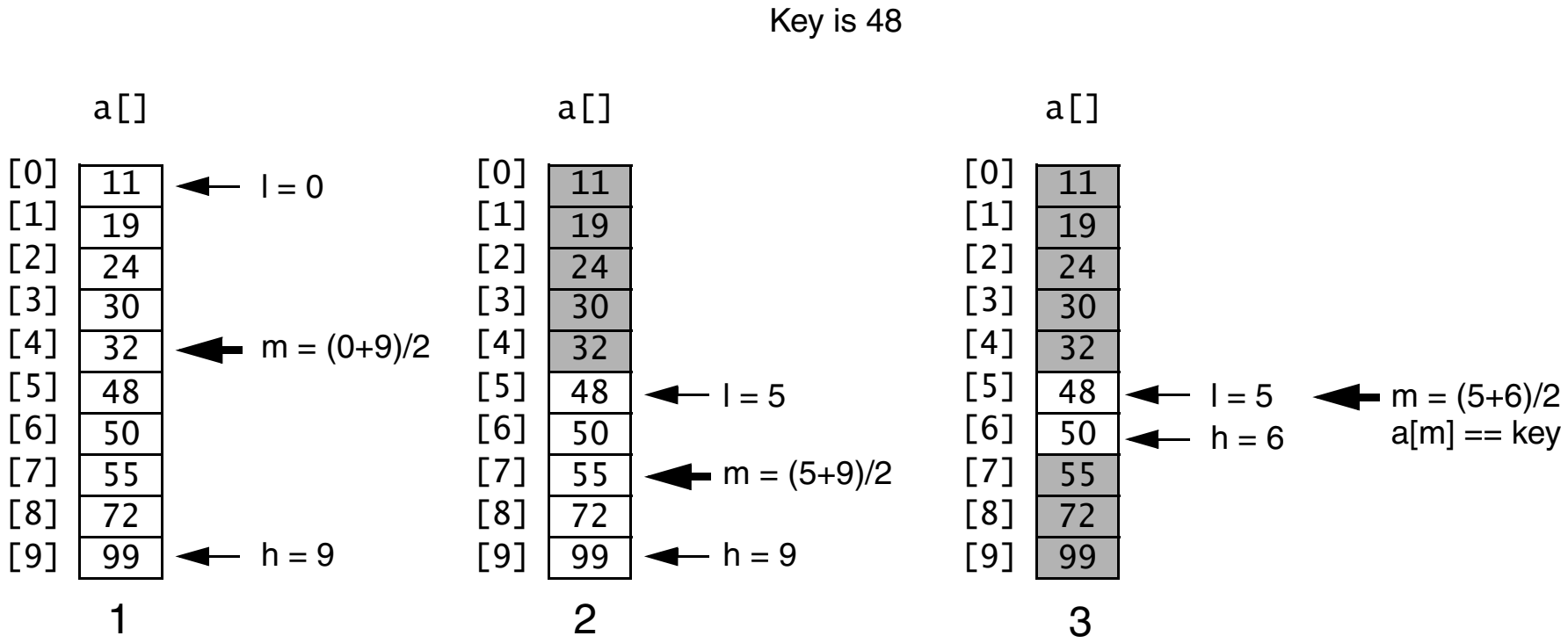
If the middle element is smaller than the key

Continue search in the upper half

If the middle element is greater than the key

Continue search in the lower half

Binary search (cont.)



Maximum number of comparisons is given by $2^{\texttt{comparisons}} = \texttt{N\_ELEMENTS}$, i.e. $\texttt{comparisons} = \lceil \log_2 \texttt{N\_ELEMENTS} \rceil$.

E.g. for `N_ELEMENTS` equal to 9, `comparisons` is equal to 3.

Binary search (cont.)

- Example: Searching in a sorted array of integers.

```
int a[] = {11,19,24,30,32,48,50,55,72,99}; // values in ascending order.
int key = 48;

int low = 0, high = a.length - 1, middle = -1;
boolean found = false, moreElements = true;
while (moreElements && !found) {
    middle = (low + high) / 2;                // middle of the (sub)array
    if (low > high) moreElements = false;     // key is not in array
    else if (a[middle] == key) found = true;  // key is in a[middle]
    else if (a[middle] < key) low = middle + 1;  // if key in array,
                                                 // it must be in upper half
    else high = middle - 1; // if key in array, it must be in lower half
}
// !(moreElements && !found) <=> (!moreElements || found)
assert (!moreElements || found) : "Error in binary search";
if (found)
    System.out.println("Value " + key + " found in element with index " + middle);
else
    System.out.println("Value " + key + " is not in array!");
```

Binary search for objects

- Binary search method from the `GenericUtility` class.

```
public static int binarySearch(Comparable[] a, Comparable key) {
    int l = 0, h = a.length - 1, m = -1;
    boolean found = false, moreElements = true;
    while (moreElements && !found) {
        m = (l + h) / 2;                          // middle of the array
        if (l > h) moreElements = false;          // key is not in the array
        else {
            int status = a[m].compareTo(key);     // Compare
            if (status == 0) found = true;        // key is in a[m]
            else if (status < 0) l = m + 1;       // if key is in array,
                                                  // it must be in the upper half
            else h = m - 1; // if key is in array, it must be in lower half
        }
    }
    // !(moreElements && !found) <=> (!moreElements || found)
    assert (!moreElements || found) : "Error in binary search";
    if (found)
        System.out.println(key + " found in element with index " + m);
    else
        System.out.println(key + " is not in the array!");
    return m;  // Returns the index where key is or should have been.
}
```

Example: sorting and searching with objects

```
public class Item implements Comparable {
  String itemName;   // name of item that can be purchased
  int stock;         // number of items in stock

  Item(String itemName, int stock) {
    assert stock >= 0: "Stock must be more than 0 items.";
    this.itemName = itemName;
    this.stock = stock;
  }

  public String getItemName() { return itemName; }
  public int getStock() { return stock; }
  public void setItemName(String itemName) {
    this.itemName = itemName;
  }
  public void settStock(int stock) {
    assert stock >= 0: "Stock must be more than 0 items.";
    this.stock = stock;
  }
```

```
  public int compareTo(Object obj) {
    assert (obj instanceof Item) : "Not an Item object";
    if (this == obj) return 0;
    Item item = (Item) obj;
    String itemName2 = item.getItemName();
    int stock2 = item.getStock();
    if (itemName.compareTo(itemName2) == 0) { // same itemName
      return stock -  stock2;
    }
    else
      return itemName.compareTo(itemName2);
  }

  public boolean equals(Object obj) {  // compatible with compareTo()
    if (!(obj instanceof Item))
      return false;
    return this.compareTo(obj) == 0;
  }

  public String toString() { // overrides toString() from the Object class
    return "\nItem name: " + itemName + "\tNumber in stock: " + stock;
  }
}
```

Example: sorting and searching with objects (forts.)

- Using the sorting method from the `GenericUtility` class.

```
public class ItemCatalogue {
  public static void main(String[] args) {
    Item[] itemCatalogue = {
        new Item("Diet Coke 0.5l", 30),
        new Item("Diet Coke 0.33l", 20),
        new Item("Pepsi Max 0.5l", 20),
        new Item("Schweppes 0.5l", 10),
        new Item("Pepsi Max 0.33l", 25)
    };
    GenericUtility.printArray("Unsorted item catalogue:", itemCatalogue);
    GenericUtility.bubbleSort(itemCatalogue);
    GenericUtility.printArray("Sorted item catalogue:", itemCatalogue);
  }
}
```

Output from the program:

```
Unsorted item catalogue:
Item name: Diet Coke 0.5l       Number in stock: 30
Item name: Diet Coke 0.33l      Number in stock: 20
Item name: Pepsi Max 0.5l       Number in stock: 20
Item name: Schweppes 0.5l       Number in stock: 10
Item name: Pepsi Max 0.33l      Number in stock: 25
Pass: 1
Pass: 2
Pass: 3
Sorted item catalogue:
Item name: Diet Coke 0.33l      Number in stock: 20
Item name: Diet Coke 0.5l       Number in stock: 30
Item name: Pepsi Max 0.33l      Number in stock: 25
Item name: Pepsi Max 0.5l       Number in stock: 20
Item name: Schweppes 0.5l       Number in stock: 10
```

Sorting and searching using the Java standard library

```
// Using the Java standard library for sorting and searching arrays.
import java.util.Arrays;
public class SortingByArrays {
  public static void main(String[] args) {
    int[] numbers = { 8, 4, 2, 6, 1 };
    Utility.printArray("Unsorted array: ", numbers);
    Arrays.sort(numbers);
    Utility.printArray("Array sorted by Java library: ", numbers);
    int key = 4;
    int index = Arrays.binarySearch(numbers, key);
    System.out.println("Key " + key + " has index " + index);
    key = 5;
    index = Arrays.binarySearch(numbers, key);
    System.out.println("Key " + key + " has index " + index);
  }
}
```

Program output:

```
Unsorted array: 8 4 2 6 1
Array sorted by Java library: 1 2 4 6 8
Key 4 has index 2
Key 5 has index -4
```