KIT101 Notes — 15 — Sorting & Searching (Arrays)

James Montgomery

Revised: 2020-02-16

References: L&L 8.1–8.3, 10.4, 10.5

And later: utility methods in `java.util.Arrays` class:
http://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html

Refer to the Part 10 Managing Collections with Arrays of these notes for an introduction to arrays. Also, be aware that these notes don't go into as much detail about some of the algorithms as the Week 9 lecture slides do, and the slides also have diagrams (so refer to them too).

Processing lots of data

Data structures such as arrays enable us to implement algorithms for processing large amounts of data efficiently (and sometimes not so efficiently) because there is a relationship between one piece of data and another. In the case of arrays, each element is adjacent to one or two other elements, so we can:

- traverse (or iterate over, or loop over) all the elements in an array; or
- go directly to any particular location in the array.

Common tasks with arrays

There are three broad tasks that apply to any data structure, including arrays:

- Process every element (for example, to display, fill with a particular value, or add up the values already present)
- Rearrange the existing elements, which includes tasks such as
  - Moving elements along within the array (to make space to insert another value), and
  - Sorting the elements
- Search (for a particular value, or a value with a particular property)

Where to find array-processing algorithms and implementations

Algorithms can be found in

- text books and their references
- many places on the web, including Wikipedia (its coverage of sorting and searching algorithms is very good)

Implementations of these algorithms could be:

1. written by you each time they're needed;
2. prepared in reusable libraries (such as the `ArrayRoutines` class distributed with the lecture); or
3. found in the Java standard library.

Option (1) is good for practice and understanding, (2) is good for saving you time later and, in the future, unless you're doing something unusual, (3) should be your preferred source for efficient and thoroughly-tested implementations.

Some useful Java standard library array processing methods (all are class methods) include:

Package.Class	Method header	Notes
`java.lang.System`	`public static arraycopy(Object source, int srcPos, Object dest, int destPos, int size)`	`source` and `dest` must be arrays to use this
`java.util.Arrays`	`public static int binarySearch(int[] array, int key)`	Implements binary search
	`public static char binarySearch(char[] array, char k)`	There are variants of this method for all data types
	`public static boolean equals(int[] a1, int[] a2)`	Are `a1` and `a2` the same length and do they contain the same values in the same positions?
	`public static void sort(double[] array)`	Sorts the array (variants exist for all types that can be ordered)
	`public static void fill(int[] a, int val)`	Fills the array with the given value
	`public static void fill(int[] a,`	As above but only between

	`int from, int to, int val)`	positions `from` and `to`

Sorting

Sorting is the process of arranging a list of items into a particular order (which requires that there be some way to compare two items to know which belongs before the other). There are many algorithms for sorting a list of items, which vary in their efficiency. An inefficient algorithm may take *thousands* of times longer to process a large amount of data than an efficient algorithm.

An example sorting algorithm: Insertion Sort

The basic approach of insertion sort is to:

- pick any item and insert it into its proper place in a sorted sublist
- repeat until all items have been inserted

In practice this means the following high-level algorithm:

1. consider the first item to be a sorted sublist (of one item)
2. insert the second item into the sorted sublist, shifting items as necessary to make room to insert the new addition
3. repeat until all values are inserted into their proper position

See the lecture for an example of this in action and the accompanying `ArrayRoutines.java` for an example implementation.

Sorting objects

Sorting primitive types (other than `boolean`) is easy because they have a *natural* order: 1 is less than 2, which is less than 5, and so on. Sorting objects can be more difficult because not all classes of object have a natural order. In fact, unless the author of a class implements a particular method called `compareTo()` then it can be impossible to sort objects of that class.

`Strings` do have a `compareTo()` method, which we looked at earlier in semester. Given two `Strings` `s1` and `s2`, `s1.compareTo(s2)` returns a value less than zero if `s1` belongs before `s2` (in a dictionary sense, but also where upper case letters are considered to belong before lower case ones), more than zero if it belongs after, and zero if they represent the same text. When implementing a sorting algorithm we only have to consider one of these cases (which can be either `s1` belongs before `s2` or `s1` belongs after `s2`).

Tip: If you want to compare strings in a dictionary sense, ignoring capitalisation, then use the `compareToIgnoreCase()` method instead.

Extension task: If you feel really confident with the material so far then take a look at the `Comparable` interface:

https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html. It's what Strings and other "orderable" classes implement.

Searching

The concept of "search" covers a wide range of different activities. Two key questions we need to ask ourselves (plus some examples) are:

- What are we searching for? Could be:
  - a particular value in the array
  - the largest value, or the smallest value
  - a value that matches some condition
- What do we want to know? Could be:
  - `boolean` (yes/no answer)
  - actual value
  - position of value

Implementing a method to search an array

To work out what to write when implementing a method to search array, consider the following three things:

- **Data in (parameters):**
  - Always the array
  - Some of
    - target (to search for)
    - start position, end position (if searching only part of the array)
- **Data out (return type):** depends on the task, e.g.,
  - `boolean` (found or not)
  - index of found value
  - the value found (being the largest or smallest)
- **The Algorithm:** The are *many* alternatives; we only look at two: linear search and binary search.

Linear search

Linear search is based on the algorithms you've implemented for processing the items in an array by traversing the array and doing something with each element (printing it, replacing its value, etc.). The key difference is that the loop can stop when the element is found (saving some time).

Linear search is appropriate when the elements of the array are not ordered.

```
Algorithm: Linear Search

Parameters:
an array
a target value to find
```

```
Returns:
the index of target in array, -1 if not found

Variables:
int index, the position of target, initialised to -1
boolean found, flags if target found, initialised to false
int current, the current position to examine

Steps:
Assign current the value of the starting position of search (0 is the de
fault)
While not at end of array and not found
|   If array[current] is the same as target then
|   |   Assign index the value of current
|   |   Assign found the value true
|   Else
|   |   Move to next element (increment current)
Return index
```

Binary search

When looking for someone in a phone book it's typical to take a guess how far through the book they appear based on their name and knowing that names appear in alphabetical order. Then, if that person isn't on the first page looked at, we can still work out which section of the phone book to look in, further back or further forward.

When the data in an array are ordered we can vastly speed up search by applying a systematic approach to identifying the most likely region of the array for the value we want to find. Unlike the phone book example, we probably can't make a guess as to which is the best starting point, since we don't know how values are distributed through the array (apart from knowing they are ordered), but that still won't slow us down much.

Binary search (which has nothing to do with the binary number system used by computers) starts by looking at the middle element in an array and, if it isn't a match for the target value, decides which half of the array the target will appear in (if present). It then repeats by looking at the middle of *that* half of the array, and so on.

It's much faster than linear search because it can eliminate half the elements to be searched at each step. But a downside is that the array *must* be sorted first.

```
Algorithm: Binary Search

Parameters:
an array
a target value to find

Returns:
the index of target in array, -1 if not found
```

```
Variables:
int index, the position of target, initialised to -1
boolean found, flags if target found, initialised to false
int low, the start of the current search area
int middle, the midpoint of the current search area
int high, the end of the current search area

Steps:
Assign low to first position and high to last position
While there are elements left to search and not found
|   Assign middle the value (low + high)/2 (integer division, of course)
|   If array[middle] is target then
|   |   Assign index the value of middle
|   |   Assign found the value true
|   Else
|   |   If target should appear before element at middle then
|   |   |   Adjust high to middle - 1 (to search lower half)
|   |   Else
|   |   |   Adjust low to middle + 1 (to search upper half)
Return index
```

For those who are interested

This is not assessable, and only for those looking for more information about the built-in sorting and searching algorithms in Java

`java.util.Arrays` has a large number of methods for processing arrays, including for sorting and searching them. The methods for sorting numerical primitives such as `int`s, `double`s and `char`s (we see them as characters, but the computer sees them as numbers) use a version of the algorithm known as Quicksort. The `Arrays` class's `sort()` method for dealing with objects like Strings uses a variant of Merge Sort. Away from the computer, you'll find that a mixture of insertion sort and merge sort is really handy when you need to order a larger number of sheets of paper: divide the original, unordered pile into a few piles; sort each pile as you normally would (you'll find yourself doing insertion sort naturally); then merge the sorted piles by taking the next piece of paper in sequence from the top of one of the piles until you're left with only your single sorted pile of paper.

The `Arrays` class also has a `binarySearch()` method (really several such methods for arrays with different element types). It implements exactly the algorithm described above.
