Sorting

CO7005 Software Development Techniques

Dr Stuart Cunningham
s.cunningham@chester.ac.uk

“In order to use a computer properly, it is important to acquire a good understanding of the structural relationships present within data, and of the techniques for representing and manipulating such structure within a computer.”

(Knuth 1973)

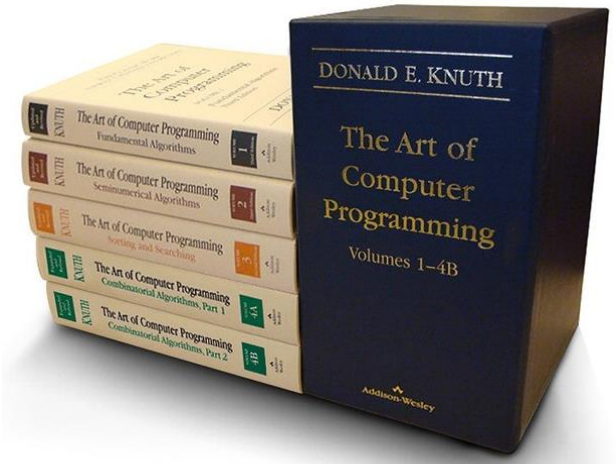


Image: https://www.informit.com/store/art-of-computer-programming-volumes-1-4b-boxed-set-9780137935109

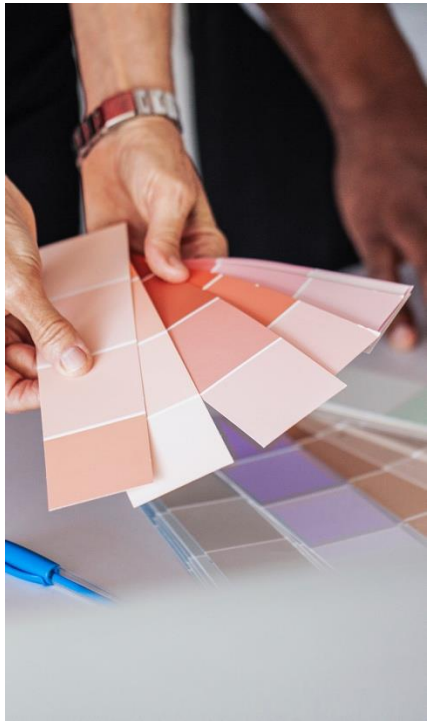
Sorting Problems

- A common computational operation is sorting data
- For example:
  - Putting names in alphabetical order
  - Sorting a file system from oldest to newest
  - Creating a league table based on points
  - Ordering items waiting in a queue
- There are many approaches to sorting
- In performing a sort, elements in a set or array are *comparable*
  - e.g., one element is larger or smaller than another

How do we sort things?

- Suppose we have three sets of numerical data we want to sort each from lowest to highest
  - `A = {6, 3, 11}`
  - `B = {8, 5, 10, 4}`
  - `C = {2, 9, 5, 3, 6, 7}`
- How do we approach this as humans?

Selection Sort



- Runs through data from beginning to end
- Utilises the *length* of the data and an *index*
  - Scans data for largest (or smallest) value
  - Swap largest value with last (first) value
  - Reduces effective length of data by 1
  - Process repeats
  - Ends when length of data reduces to size of 1

Selection Sort

A = {6, 3, 11}

{6, 3, **11**}
{3, **6**, 11}
{**3**, 6, 11}

B = {8, 5, 10, 4}

{8, 5, 4, **10**}
{4, 5, **8**, 10}
{4, **5**, 8, 10}
{**4**, 5, 8, 10}

Bubble Sort

- Similar approach to selection sort
  - Compares adjacent pairs of values
  - Determines largest in the pair
  - Values swapped if first value bigger (or smaller)
  - Continues for each pair in the set until end reached
  - Process repeats
- Pushes largest (or smallest) value to ‘end’ of the data (hence ‘bubble’ sort)

Bubble Sort

A = {6, 3, 11}

{6, 3, 11}
{3, **6, 11**}
{3, 6, 11}

B = {8, 5, 10, 4}

{8, 5, 10, 4}
{5, **8, 10**, 4}
{5, 8, **10, 4**}
{5, 8, 4, 10}
{5, **8, 4,** 10}
{5, 4, **8, 10**}
{5, 4, 8, 10}
{4, **5, 8,** 10}
{4, 5, **8, 10**}

Insertion Sort

- Forms a *partially* sorted list over sort duration
- Partial list grows by inserting unsorted elements and locating them in the correct place in the sequence
  - Ignores ‘sorted’ elements in the array
  - Assumes initial first element is sorted (ignores it)
  - Compare current element *n* to *n-1*
  - If *n* < *n-1* swap them
  - Check *n* in previous elements of sorted list

Insertion Sort

A = {6, 3, 11}

{6, 3, 11}
{3, **6, 11**}

B = {8, 5, 10, 4}

{8, 5, 10, 4}
{5, 8, 10, 4}
{5, **8, 10,** 4}
{5, 8, **10, 4**}
{5, 8, 4, 10}
{5, **8, 4,** 10}
{5, 4, 8, 10}
{5, 4, 8, 10}
{4, 5, 8, 10}

Arrays Class

- Items to be sorted will often be held in a *data structure*
- A familiar example is the *array* - *e.g.* `int[] nums = {3, 7, 5, 1}`
- Java's *Arrays class* has *static* methods for working with arrays
- Including `Arrays.sort()`
  - Implements a *Quicksort*
  - *"…arguably the best general-purpose sorting algorithm…currently available"* (Schildt 2022)
- Changes array contents

Arrays Class

```java
import java.util.Arrays;

class ArraysSimple {

 public static void main(String[] args) {
  double [] sales = {2.797, 1.656, 5.638, 0.923, 8.108, 3.048, 1.107, 0.007};
  System.out.println("Unsorted Array");
  System.out.println(Arrays.toString(sales));
  // sort array and re-display
  Arrays.sort(sales);
  System.out.println("Sorted Array");
  System.out.println(Arrays.toString(sales));
 }
}
```

```
>> java ArraysSimple
Unsorted Array
[2.797, 1.656, 5.638, 0.923, 8.108, 3.048, 1.107, 0.007]
Sorted Array
[0.007, 0.923, 1.107, 1.656, 2.797, 3.048, 5.638, 8.108]
```

Java Collections Framework

“Though conceptually simple, collections are one of the most powerful parts of any programming language. Collections implement data structures that lie at the heart of managing complex problems. …It can also save you from reinventing the wheel.”

(Loy 2020)

Collections Framework / Class

- A set of *interfaces* for common tasks on groups of objects
- Useful for advanced tasks, such as reverse order sorting
- For arrays, can be used to sort according to various criteria

sort

```
public static <T> void sort(T[] a,
                            Comparator<? super T> c)
```

Sorts the specified array of objects according to the order induced by the specified comparator. All elements in the array must be *mutually comparable* by the specified comparator (that is, `c.compare(e1, e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the array).

Source: https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#sort-T:A-java.util.Comparator-

```java
import java.util.Arrays;
import java.util.Collections;

public class ArraysCollections {

public static void main(String[] args) {
  String [] names = {"Olivia", "Muhammad", "Theo", "Precious"};

  // output unsorted array
  System.out.print("Unsorted:\t");
  System.out.println(Arrays.toString(names));

  // sort and display - natural order
  Arrays.sort(names);
  System.out.print("Natural Order:\t");
  System.out.println(Arrays.toString(names));

  // sort and display - reverse order
  Arrays.sort(names, Collections.reverseOrder());
  System.out.print("Reverse Order:\t");
  System.out.println(Arrays.toString(names));
  }
}
```

```
>> java ArraysCollections
Unsorted:       [Olivia, Muhammad, Theo, Precious]
Natural Order:  [Muhammad, Olivia, Precious, Theo]
Reverse Order:  [Theo, Precious, Olivia, Muhammad]
```

The Comparable Class

- Useful for sorting complex data structures
- Such as data classes with multiple attribute types
- Use Comparable class and override `compareTo(object o)`
  - This becomes the class's *natural* ordering
- Implementing classes compare themselves with others
  - Using `this` instance and a given instance (`object o`)

People.java

```
>> java People

Unsorted Characters
---------------------
Blackadder      40
S Baldrick      51
Melchett        43
Mrs Miggins     38
Percy Percy     37
Elizabeth       32

Age Sorted Characters
---------------------
Elizabeth       32
Percy Percy     37
Mrs Miggins     38
Blackadder      40
Melchett        43
S Baldrick      51
```

- Uses Person (data) class that *implements* Comparable
  - Contains name (`String`) and age (`float`)
- People class creates instances and adds to an ArrayList
- `Characters` ArrayList sorts according to overridden `comapreTo()`
  - Age defined as the sorting field

The Comparator Class

- Custom sorting facilitated using Comparator *interface*
- A class can be defined that *implements* Comparator
- Implementations must override the `compare()` method
  - Method must return an integer
  - Negative, zero, or positive (less than, equal to, or greater)
- Allows sorting of classes containing (multiple) data types
- Unlike Comparable, we can implement *multiple* Comparators to apply on the same classes (data)

CricketTeam.java

```
>> java CricketTeam

  Sorted by Average
--------------------
Joe     Root    50.29
Virat   Kohli   49.29
Temba   Bavuma  35.25
Beth    Mooney  33.55
Sidra   Ameen   27.87
Mandy   Mangru  15.0

  Sorted by Surname
--------------------
Sidra   Ameen   27.87
Temba   Bavuma  35.25
Virat   Kohli   49.29
Mandy   Mangru  15.0
Beth    Mooney  33.55
Joe     Root    50.29
```

- Uses CricketBatter (data) class
- CricketTeam creates instances then sorts `team` ArrayList
- Using `Collections.sort()` and *custom comparators:*
  - CricketCompareAves – by average (`float`)
  - CricketCompareNames – by surname (`String`)

References

Knuth, D. E. (1973). *The Art of Computer Programming: Fundamental Algorithms*, volume 1 (2nd ed.). Addison-Wesley Professional.

Loy, M. (2020). *Learning Java: an introduction to real-world programming with Java* (Fifth). O'Reilly.

Schildt, H. (2022). *Java: a beginner's guide* (9th ed.). McGraw Hill.