Technische Hochschule Rosenheim
Technical University of Applied Sciences

Object-oriented programming
Chapter 7a – Sorting

Prof. Dr Kai Höfig

Sorting

- Sorting is a process in which an order is created by **comparing** elements and **swapping** them where necessary, so that the *smallest* element is in the first position. Today we want to work out a few of these sorting methods.

    1. **Bubble sort (comparing and swapping)**
    2. **Selection sort**
    3. **Insertion sort**
    4. **Sorting by divide-and-conquer algorithm**
    5. **Quicksort**

- The different procedures are visualised on the website https://visualgo.net/de/sorting.

Bubble sort (comparing and swapping)

- In order to compare two primitive data types (`short`, `int`, `float`, etc.), we can use the Java comparison operators <, > and ==. In order to compare two objects, we already know the Java interfaces `Comparable<T>` and `Comparator<T>`, with their methods `compareTo(T other)` and `compare(T t1, T t2)`. For the sake of clarity, we will initially restrict ourselves to `int` and the < operator. At the end of this chapter, we will generalise the algorithms in order to define our own sorting orders.

- It becomes more interesting when swapping; a helper variable must be used here:

```
class Sorting {
  static void swap(int[] a, int i, int j) {
    int helper = a[i];
        a[i] = a[j];
        a[j] = helper;
    }
}
```

- Together with the syntax elements for conditional (`if-else`) and repeated (`for, while`) execution, we are now familiar with the basic tools for sorting.

Selection sort

- One of the easiest and clearest sorting methods is sorting by selection (**selection sort**).
  - To do so, we treat the array to be sorted as two sub-arrays: a sorted part **S** (left) and an unsorted part **U** (right).
  - At the beginning, the sorted part is empty, so it ends before the first element.
  - We then search sequentially for the next smallest element in the unsorted part, and swap it with the first element of the unsorted part. As a result, the sorted part now increases by one element.

```
a = [|3 2 4 1 ]  // min = 1; tausche mit 3
    S|U

    [ 1|2 4 3 ]  // min = 2; korrekt positioniert
      S|U

    [ 1 2|4 3 ]  // min = 3; tausche mit 4
        S|U

    [ 1 2 3|4 ]  // 1-elementiges Array ist sortiert
          S|U

    [ 1 2 3 4|]  // fertig!
            S|U
```

Selection sort implementation

- We can express the algorithm (repeated minimum search and swap) in Java as follows:

```
static void ssort(int[] a) {
  for (int i = 0; i < a.length; i++) {
    // position of the current minimum
    int p = i;

    // search for smaller value, note position
    for (int j = i+1; j < a.length; j++)
      if (a[j] < a[p])
            p = j;

    // swap if necessary
    if  (i != p)
      swap(a, i, p);
    }
}
```

- We can see there is only repeated swapping here, the array is not duplicated at any time. *Selection sort* is a so-called **in-place sorting method**, which does not require any additional memory, except for a few helper variables.

- The effort required for this sorting method is O(n^2), as can easily be seen from the two nested `for` loops. Since this is an in-place method, it is particularly suitable for arrays.

Insertion sort

- Related, but somewhat trickier to implement, is sorting by insertion (**insertion sort**). Here, we also distinguish between the already sorted part `S` and the unsorted part `U`. However, we do not search for the minimum in `U`, but instead take the first element `x` out of `U` and insert it in the correct position in `S`, by shifting all elements $y > x$ in `S` one position to the right:

```
a = [ 3 2 4 1 ]  // 3 einsortieren; bereits am richtigen Ort
    S|U

    [ 3|2 4 1 ]  // 2 einsortieren
      S|U

        2
    [ 3 -|4 1 ]  // Element herausnehmen, S vergrößern
    [ - 3|4 1 ]  // größere nach rechts schieben
    [ 2 3|4 1 ]  // einfügen
        S|U

          4
    [ 2 3 -|1 ]  // Element herausnehmen, S vergrößern
    [ 2 3 4|1 ]  // nichts zu schieben, einfügen
          S|U

            1
    [ 2 3 4 -|]  // Element herausnehmen, S vergrößern
    [ 2 3 - 4|]  // größere nach rechts schieben
    [ 2 - 3 4|]
    [ - 2 3 4|]
    [ 1 2 3 4|]  // einfügen, fertig!
            S|U
```

Insertion sort implementation

- Here too, the effort is O(n^2), since there are once again two nested loops. This is another in-place sorting method, but we can also see that depending on the data situation, many swap operations may be necessary to achieve removal from U and insertion into S.

- Since insertion and removal in chained lists is quick and easy, this process is particularly suitable for chained lists.

```
static void isort(int[] a) {
  for (int i = 1; i < a.length; i++) {
    // remove current value
        // (a[i] can thus be overwritten)
    int x = a[i];

    // shift all elements one position to the right
        // until the insertion position is found
    int j = i-1;
    while (j >= 0 && a[j] > x) {
      swap(a, j, j + 1);
            j--;
        }

    // insert in the "free" position
    a[j+1] = x;
    }
}
```

Sorting by divide-and-conquer algorithm

- Today we will apply recursion to the sorting problem in order to be able to describe the problems more easily.
- If we want to describe sorting recursively, we can determine:
  - An empty or single-element array is already sorted (base case, i.e. terminating case).
  - A multi-element array is divided into two parts, sorted (using recursion), then put back together again.

```
      [38 27 43 3|9 82 10]
                /  \     <------- teilen (Rekursion)
    [38 27|43 3]   [9 82|10]
         / \             / \ <-- teilen
  [38|27]   [43|3] [8|82]   10   Terminalfall!
    / \       / \   / \  <--|--- teilen
   38 27     43  3  8   82   |    Terminalfall!
------------------------------
    \ /       \ /   \ /  <--|--- zusammenführen
  [27 38]   [3 43] [8 82]   |
         \ /             \ / <-- zusammenführen
    [3 27 38 43]   [9 10 82]
                \ /  <---------- zusammenführen
      [3 9 10 27 38 44 82]
```

Merge sort implementation (1)

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- In Java we implement this recursion as follows; putting the parts back together (`merge`) is shown separately for the sake of clarity.

```java
static int[] msort(int[] a) {
  // base case -- already sorted.
  if (a.length < 2)
    return a;

  int p = a.length / 2;

  // sort partial lists
  int[] l = msort(Arrays.copyOfRange(a, 0, p));
  int[] r = msort(Arrays.copyOfRange(a, p, a.length));

  // put sorted partial lists back together
  return merge(l, r);
}
```

- This process is called **merge sort**, and (analogous to insertion into a binary tree) has the complexity O(n log n) -- in terms of the computing effort. However, depending on the implementation, the amount of memory required can be significantly higher if copies of the arrays are created. Here too: since appending to and splitting of lists can be implemented very efficiently, merge sort is particularly suitable for chained lists.

Merge sort implementation (2)

- The question remains as to how two sorted arrays can now be put back together (merged). To do this, we alternately take the smallest element of each array and insert it into a new one:

```java
private static int[] merge(int[] a, int[] b) {
  // new array as big as a and b together
  int[] res = new int [a.length + b.length];

  // three indexes: for res, a and b
  int i = 0, l = 0, r = 0;
  while (l < a.length && r < b.length) {
    if (a[l] < b[r])
            res[i++] = a[l++];
    else
      res[i++] = b[r++];
    }

  // anything remaining on the left or right?
  while (l < a.length)
        res[i++] = a[l++];
  while (r < b.length)
        res[i++] = b[r++];

  return res;
}
```

Quicksort

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Quicksort utilises the basic idea of merge sort, but does not divide the array into two halves systematically; instead, it partitions the array into a left and right half, based on whether the elements are smaller or larger than a pivot element (pivot value). We can show that quicksort is more efficient than merge sort in many cases, so it is used as a standard implementation in many libraries.

```java
static void qsort(int[] a) {
  qsort(a, 0, a.length);
}

private static void qsort(int[] a, int from, int to) {
  // short arrays already sorted
  if (to - from < 2)
    return;

  int p = partition(a, from, to);
  if (from < p - 1)
    qsort(a, from, p);
  if (p < to)
    qsort(a, p, to);
}
```

Comparable and Comparator

- In the above examples, the sorting order has always been established using the `<` operator. If we now want to compare objects, we must either implement `Comparable`, and thus the method `compareTo`, or we can use a `Comparator` that can compare two objects with `compare`. Revision: both methods have `int` as the return value, namely

- < 0 (usually -1) if the first element is smaller than the second;
- 0 if both elements are the same; and
- > 0 (usually +1) if the first element is larger than the second.

- For the primitive data types, the corresponding wrapper type implements the `Comparable` interface and a static `compare` method:

```java
System.out.println(Integer.compare(1, 5));  // "-1"
System.out.println(Integer.compare(3, 3));  // "0"

Integer i = 4;
System.out.println(i.compareTo(2));  // "1"
```

Sorting using Comparator

```java
static void ssort(Integer[] a, Comparator<Integer> c) {
  for (int i = 0; i < a.length; i++) {
    // position of the current minimum
    int p = i;

    // search for smaller value, note position
    for (int j = i+1; j < a.length; j++)
      if (c.compare(a[j], a[p]) < 0)
        p = j;

    // swap if necessary
    if  (i != p)
      swap(a, i, p);
    }
}
```

Previously

```java
if (a[j] < a[p])
```

- The sorting order is thus defined solely by whether the result of `compareTo` or `compare` is negative (i.e. "smaller").

Sorting in reverse

- Similarly, we can also define our own sorting order, e.g. descending, so the largest number first. In simple terms, this now means that the largest takes the place of the smallest for the sorting algorithm, so that the sign before the number in the result of compare is exactly the other way round.

```
Integer[] a = {3, 2, 4, 1};

Sorting.ssort(a, new Comparator<Integer>() {
  public int compare(Integer a, Integer b) {
    return -1 * a.compareTo(b);  // reverse the sign before the number!
  }
});
```

- Alternatively, the above could also be achieved in this way:

- `return b.compareTo(a)`, i.e. by reversed order
- `return Integer.compare(b, a)`, also reversed order
- `return b - a` which is negative if a > b

Sorting orders with multiple criteria

- However, sometimes one sorting criterion is not enough; think of a list of names, for example: Here, the sorting order is initially by surname, but in case of equality, the list is then sorted further by first name. Such a `Comparator` must compare hierarchically accordingly:

```
class PersonComparator implements Comparable<Person> {
  public int compare(Person a, Person b) {
    // Surname the same? Then please sort by first name.
    if (a.getSurname().equals(b.getSurname()))
      return a.getFirstName().compareTo(b.getFirstName());
    else
            return a.getSurname().compareTo(b.getSurname());
    }
}
```

Summary

- Sorting is primarily based on comparing and swapping.
- Sorting algorithms create an ascending order, i.e. the smallest element first.
- With `Comparable` or `Comparator`, we can define these orders as we like: if one object is smaller than another, then a value <0 must be returned.
- If an existing order should be reversed, then it is sufficient to invert the sign before the number.
- Selection sort and insertion sort both have a complexity of O(n^2), whereby the latter is particularly suitable for lists.
- Although merge sort is more efficient with a complexity of O(n log n), it requires double the memory for the simple implementation.
- Many operating systems and libraries use quicksort as the default sorting method.