TECHNICAL UNIVERSITY OF APPLIED SCIENCES ULM

Assignment #3

Student names and Group: *Petar Stoyanov & Eetu Hyvärinen - Group 08*

Course: *Algorithms and Data Structures* – Professor: *Prof. Schied*
Due date: *November 28th, 2019*

Exercise 3.1

An array containing values **5, 9, 4, 7, 3, 6, 8** is given.

(*a*) How does method **partition** used for Quicksort splits this array into two parts? Show the array contents after each iteration of the loop.

(*b*) Show how the array is sorted using *Quicksort*, Indicate the essential intermediate steps corresponding to recursive calls so that the dividend-and-conquer principle can be recognized. Partitioning does not have to be done exactly according to the method **partition** presented in the lecture.

(*c*) Show how the array is sorted using **Mergesort**. Indicate the essential intermediate steps corresponding to recursive calls so that the dividend-and-conquer principle is recognizable.

Answer.

(*a*) A pivot element is selected, where values smaller than it are sent to the left of it and values bigger than it - to the right of it. That creates 2 new arrays which are then sorted recursively using the same method
There are however different versions of **Quicksort**, depending on which element is picked to be the pivot element:

(*1*) Pivot element is always the first element.

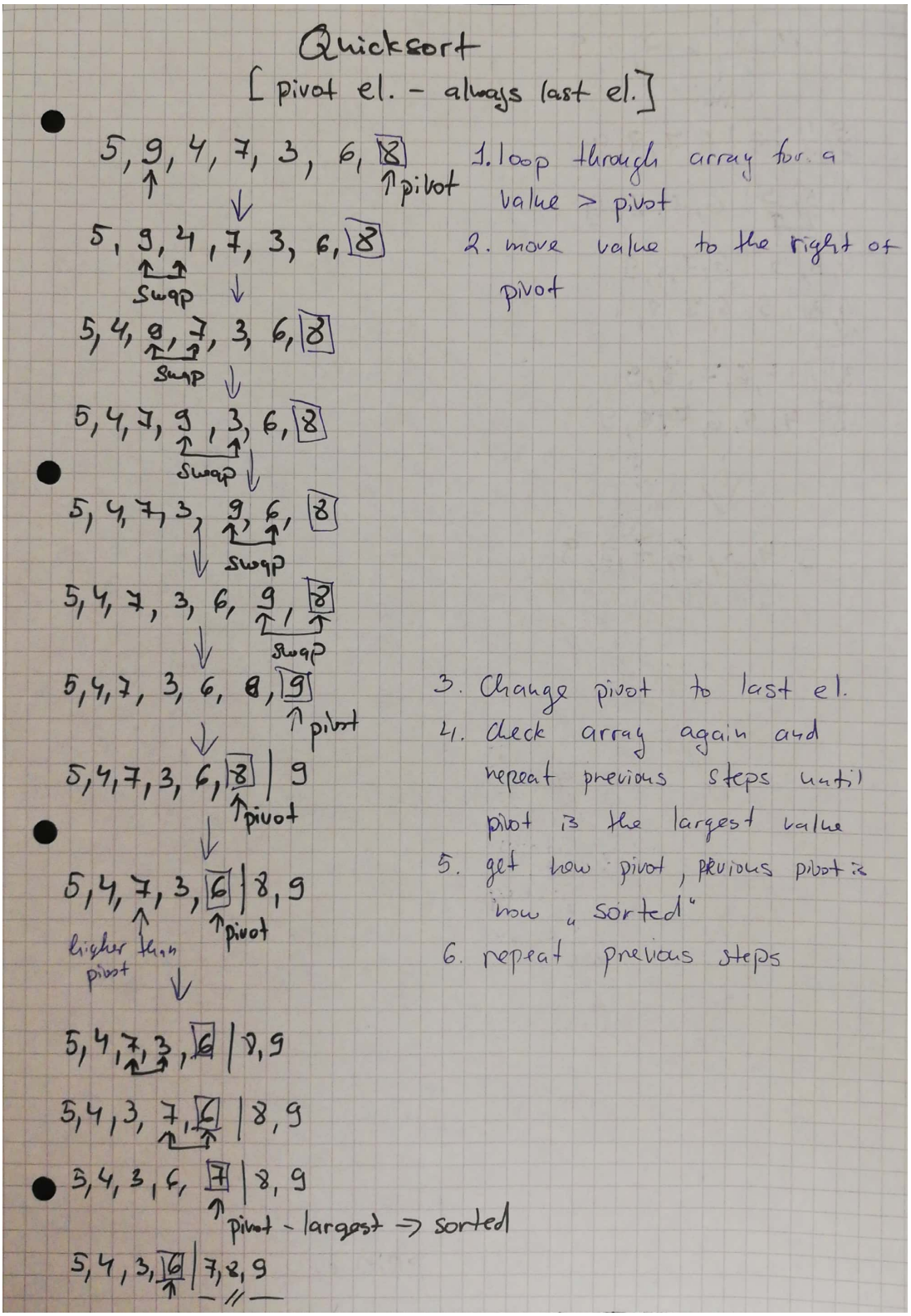
(*2*) Pivot element is always the last element.

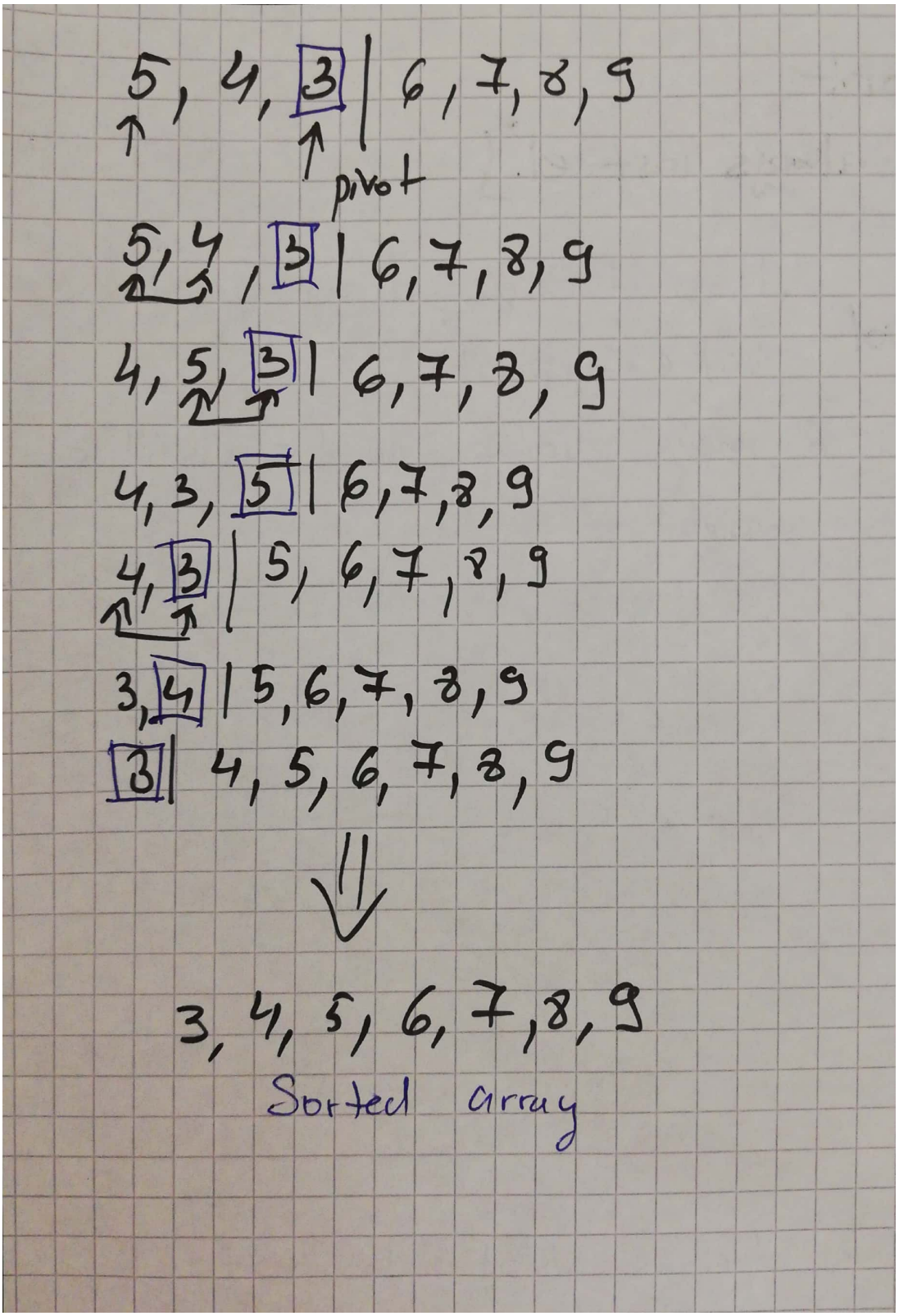
(*3*) Pivot element is a random element from the array.

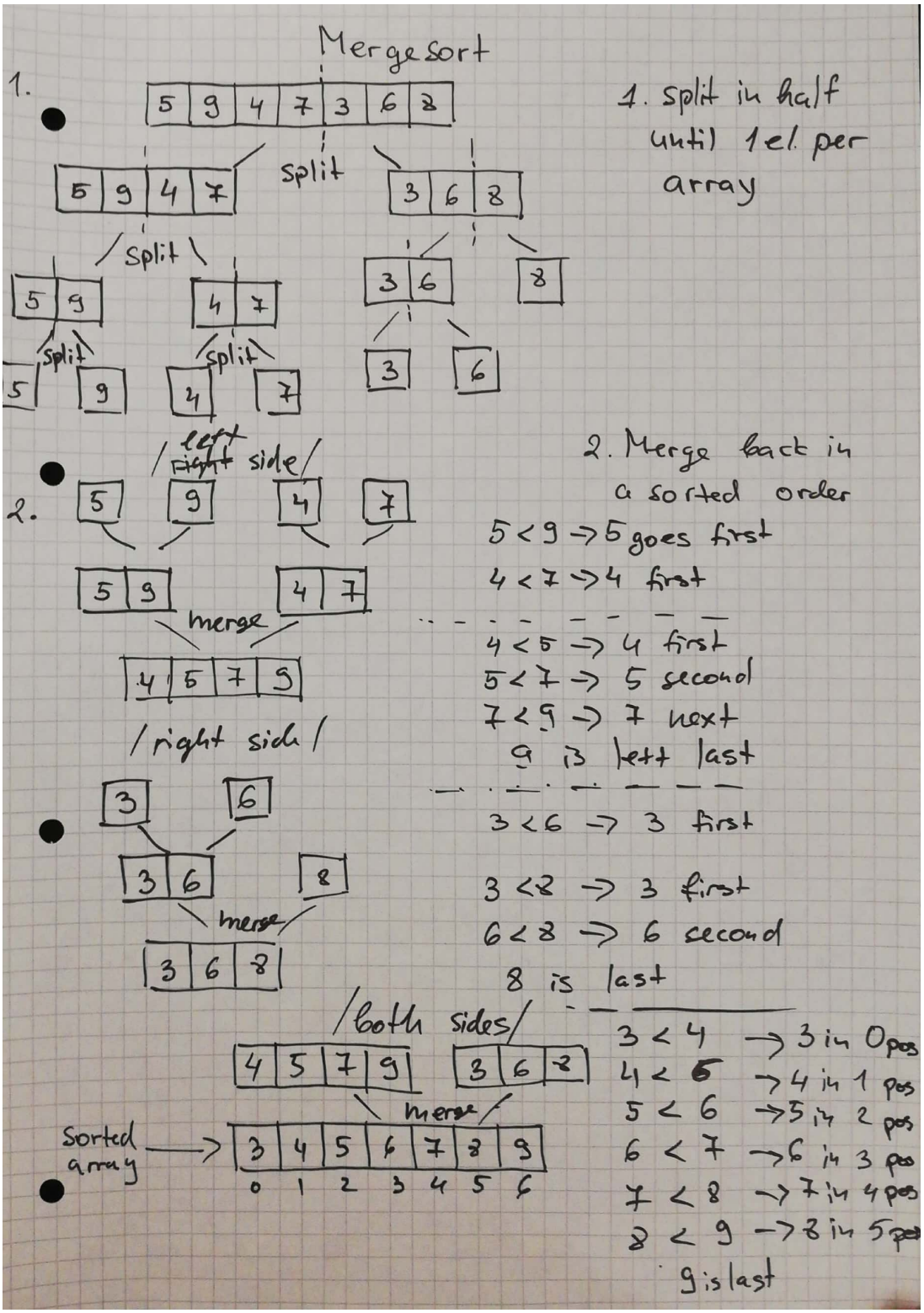
(*4*) A median value is selected as the pivot element.

(*b*) For my representation I alawys used the last unsorted element as my Pivot element. *Pictures below.*

(*c*) In my pictures I forgot to mention that values from the left side are compared to the right side, not each to every other.
That can clearly be seen in the last merge between both sides where 4 is compared to 6.
Picture below


Quicksort
[pivot el. - always last el.]
5, 9, 4, 7, 3, 6, 8
pivot
1. loop through array for a value > pivot
2. move value to the right of pivot
swap
3. Change pivot to last el.
4. check array again and repeat previous steps until pivot is the largest value
5. get how pivot, previous pivot is now „sorted"
6. repeat previous steps
higher than pivot
pivot - largest => sorted
5, 4, 3, 6 | 7, 8, 9
5, 4, 3 | 6, 7, 8, 9


5, 4, 3 | 6, 7, 8, 9
pivot
5, 4, 3 | 6, 7, 8, 9
4, 5, 3 | 6, 7, 8, 9
4, 3, 5 | 6, 7, 8, 9
4, 3 | 5, 6, 7, 8, 9
3, 4 | 5, 6, 7, 8, 9
3 | 4, 5, 6, 7, 8, 9
3, 4, 5, 6, 7, 8, 9
Sorted array


Mergesort
1.
5 9 4 7 3 6 8
split
5 9 4 7
3 6 8
split
5 9
4 7
3 6
8
split
split
5
9
4
7
3
6
1. split in half until 1 el. per array
2.
left side
right side
5
9
4
7
5 9
4 7
merge
4 5 7 9
right side
3
6
3 6
8
merge
3 6 8
2. Merge back in a sorted order
5 < 9 -> 5 goes first
4 < 7 -> 4 first
4 < 5 -> 4 first
5 < 7 -> 5 second
7 < 9 -> 7 next
9 is left last
3 < 6 -> 3 first
3 < 8 -> 3 first
6 < 8 -> 6 second
8 is last
both sides
4 5 7 9
3 6 8
merge
Sorted array
3 4 5 6 7 8 9
0 1 2 3 4 5 6
3 < 4 -> 3 in 0 pos
4 < 5 -> 4 in 1 pos
5 < 6 -> 5 in 2 pos
6 < 7 -> 6 in 3 pos
7 < 8 -> 7 in 4 pos
8 < 9 -> 8 in 5 pos
9 is last

Exercise 3.3

Implement an efficient method which calculates the maximum frequency of values that occur in array data.

For example, for the array {4, 2, 1, 5, 1, 6, 8, 1} the result 3 should be returned, since the value 1 is the most frequently occurring value with three occurrences.

Measure the running time for n = 100, 1000,..., 10 000 000 000 elements. Which runtime complexity can be recognized?

Code:

```
public static int maxFrequency(long[] data) {
        int maxOccurences = 0;

        for (int i = 0; i < data.length; i++) {
            int occurrences = 0;
            long holder = data[i];

            // count how many times a certain number/value is
               in the array
            for (long number : data) {
                if (number == holder) {
                    occurrences++;
                }
            }
            // if current number of occurances is more than
               prev max - new max
            if (occurrences > maxOccurences)
                maxOccurences = occurrences;
        }
        // return max
        return maxOccurences;
    }
```

n = 10 | maxFrequency: 5, runtime: 0.00 ms
n = 100 | maxFrequency: 14, runtime: 0.21 ms
n = 1000 | maxFrequency: 43, runtime: 7.63 ms
n = 10000 | maxFrequency: 121, runtime: 63.79 ms
n = 100000 | maxFrequency: 372, runtime: 6549.78 ms
n = 1000000 | maxFrequency: 1099, runtime: 622500.59 ms = ∼10min
Time complexity of the code seems to be $O(n^2)$
Expected runtime for:
n = 10000000 = 622500.59ms × 100 = ∼1040min = ∼17h
n = 100000000 = 170h = ∼ 7 days
n = 1000000000 = 700 days = ∼ 2 years
n = 10000000000 = 70 000 days = ∼ 192 years

Exercise 3.4

The runtime of different sorting methods is to be measured and compared. In Moodle, you will find a class **Sort** that contains implementations of *Bubblesort*, *Selectionsort*, *Insertionsort*, *Heapsort*, *Quicksort*, and *Mergesort*, as well as an auxiliary method to create a randomly filled array of the specified size.

(*a*) Determine the running time for sorting arrays with n = 10, 100,1000,10 000 and 100 000 elements (use the system clock *System.nanoTime()*).
Make sure that each time you call a sort method, unsorted data is used and that the calculation of the test data itself is not included in the time measurement.
Your measurement program and a table with the measurement results should be submitted.

(*b*) Estimate the running times of **Selectionsort** and **Mergesort** for sorting n = 1 000 000 and n = 10 000 000 entries, based on the values you measured (with brief explanation).

Answer. *a)* Table of results:

Insertion sort

n = 10, runtime: 0.005ms
n = 100, runtime: 0.0827ms
n = 1000, runtime: 3.6436ms
n = 10000, runtime: 62.1189ms
n = 100000, runtime: 1734.6013ms

Selection sort

n = 10, runtime: 0.0058ms
n = 100, runtime: 0.307ms
n = 1000, runtime: 3.2896ms
n = 10000, runtime: 41.4158ms
n = 100000, runtime: 3798.1912ms

Quick sort

n = 10, runtime: 0.0188ms
n = 100, runtime: 0.0547ms
n = 1000, runtime: 0.4114ms
n = 10000, runtime: 1.4382ms
n = 100000, runtime: 12.5554ms

Bubble sort

n = 10, runtime: 0.0053ms
n = 100, runtime: 0.271ms
n = 1000, runtime: 8.1275ms
n = 10000, runtime: 219.4975ms
n = 100000, runtime: 32154.2296ms

Heap sort

n = 10, runtime: 0.01ms
n = 100, runtime: 0.0954ms
n = 1000, runtime: 0.5776ms
n = 10000, runtime: 2.3724ms
n = 100000, runtime: 17.3788ms

Merge sort

n = 10, runtime: 0.025ms
n = 100, runtime: 0.0568ms
n = 1000, runtime: 0.6366ms
n = 10000, runtime: 2.4914ms
n = 100000, runtime: 25.0809ms

b)

1. Merge sort is a stable sort with time complexity of O(n log(n))

$$\text{We take the 2 values we have: } \sim 25 \text{ and } \sim 2.5$$

$$25 - 2.5 = 22.5$$

$$(25.0809 + 22.5) \times 10 = 47.5809 \times 19 = 475.809\text{ms for n = 1 000 000}$$

$$\text{and for n = 10 000 000}$$

$$\sim 476 \text{ and } \sim 25 \text{ , from them we take the first two numbers :}$$

$$47 - 25 = 22$$

$$(475.809 + 22) \times 10 = 497.809 \times 10 = 4978.090\text{ms}$$

2. Selection sort has a time complexity of O(n^2):

Runtime expectations for n= 1 000 000 is runtime of n=100 000 $\times$ 100
And for n = 10 000 000 its the runtime of n = 1 000 000 $\times$ 100

so: n = 1 000 000, runtime: 379 819.12ms = 380s
and n = 10 000 000, runtime: 3 7981 912ms = 38000s = 10.5h

Code:

```
public final static int MAX_N = 100_000;
public static double[] generateTestData(int length) {
    double[] values = new double[length];

    for (int i = 0; i < values.length; i++) {
        values[i] = Math.random();
    }
    return values;
}

public static void main(String[] args){
    for (int n = 10; n <= MAX_N; n *= 10) {
        double[] testData = generateTestData(n);

        // replace bubblesort with any other sort to use them
        long start = System.nanoTime();
        bubblesort(testData);
        long end = System.nanoTime();

        System.out.printf("n = " + n + ", runtime: "+ (end -
           start) / 1e6 + "ms \n");
    }
    System.out.println("- done -");
}
```

Exercise 3.6

Bucketsort should be used to sort articles by article numbers. It is assumed that the article numbers are in the range of 0 to 999. The constant c = 3 is also given.

(*a*) Explain the sorting algorithm using the following sequence of n=15 key values as an example:
631, 284, 597, 92, 198, 970, 648, 17, 558, 281, 134, 466, 535, 99, 654

(*b*) How would the way of processing change if 150 elements were to be sorted instead of 15?

Answer.

(*a*) Create 10 "buckets" for the values and divide 1000 (0-999) into 10 parts

(*1*) Bucket contains values: 0-99
(*2*) Bucket contains values: 100-199
(*3*) Bucket contains values: 200-299
(*4*) Bucket contains values: 300-399
(*5*) Bucket contains values: 400-499
(*6*) Bucket contains values: 500-599
(*7*) Bucket contains values: 600-699
(*8*) Bucket contains values: 700-799
(*9*) Bucket contains values: 800-899
(*10*) Bucket contains values: 900-999

Then we take values from the array and place them into corresponding buckets. once all values have been placed into buckets, the buckets will look like this:

(*1*) Bucket contains values: 92, 17, 99
(*2*) Bucket contains values: 198, 134
(*3*) Bucket contains values: 284, 281
(*4*) Bucket contains values:
(*5*) Bucket contains values: 466
(*6*) Bucket contains values: 597, 558, 535
(*7*) Bucket contains values: 631, 648, 654
(*8*) Bucket contains values:
(*9*) Bucket contains values:
(*10*) Bucket contains values: 970

After placing the values into the buckets, the values inside each bucket are sorted (usually with insertion sort) to go from low to high.
After this the bucket will look like this:

(*1*) Bucket contains values: 17, 92, 99

(*2*) Bucket contains values: 134, 198

(*3*) Bucket contains values: 281, 284

(*4*) Bucket contains values:

(*5*) Bucket contains values: 466

(*6*) Bucket contains values: 535, 558, 597

(*7*) Bucket contains values: 631, 648, 654

(*8*) Bucket contains values:

(*9*) Bucket contains values:

(*10*) Bucket contains values: 970

From here the values are taken out of the buckets and placed into an array.
The array will look like this:
17, 92, 99, 134, 198, 281, 284, 466, 535, 558, 597, 631, 648, 654, 970

Exercise 3.7

The following kinds of data should be sorted. Would the non-comparison based sorting algorithms **Bucketsort** and/or **Radixsort** be suitable for this? Give a short explanation for each case.

(*a*) Books on Amazon by price

(*b*) Books by 10-digit ISBN number

(*c*) German municipalities by number of inhabitants

(*d*) German municipalities by postcode

(*e*) Historical celebrities by date of birth (year/month/day)

(*f*) The mass of physical objects (e. g. electrons, cars, galaxies,...).

Answer.

(*a*) Both sorting algorithms would be suitable because we can split the price into price ranges (e.g. 0 - 50, 50-100, etc. for radixsort and 0-9, 10-19, 20-29 etc for bucketsort)

(*b*) Neither of the sorting algorithms would be suitable for sorting 10-digit ISBN numbers, because using buckets would be difficult and you can't sort them from least to most significant number

(*c*) Bucketsort would work by splitting numbers of inhabitants into different ranges. There might be many empty buckets though, considering that you might have a large difference between rural areas and cities Radixsort would work well.

(*d*) Both sorting algorithms would be suitable because all post codes are 5 digits long and digits represent areas from most significant to least significant digit.

(*e*) Bucketsort is suitable for this case since dividing it into smaller parts (buckets) is good. Radixsort would also work but it might be very slow in comparison.

(*f*) Splitting the masses into buckets might require too many buckets, depending on the size of an individual bucket. Radixsort works better for this one.

Exercise 3.8

Every German citizen is given a unique, lifelong tax identification number (tax ID) from birth. A tax ID is a sequence of 11 digits.

(*a*) Class *Person* is given with attributes for the name of the person and their tax ID. Implement a method in the *Person* class

public static void *taxIdSort(Person[] plist)*,

which sorts an array of *Person* objects ascending by tax ID using Radixsort. In Moodle you will also find a jUnit test class.

(*b*) Measure the runtimes for sorting n = 100, 1 000, 10 000, 100 000, and 1 million persons. An auxiliary method *generatePersonList(int n)* for generating random test data can be found in the *Person* class.
Does the measured runtime behaviour correspond to the theoretically expected complexity?

(*c*) Compare the runtime of sorting with *java.util.Arrays.sort()* to the runtime of your Radixsort implementation. Are you able to get faster with your Radixsort than *Arrays.sort()*?

No answer to this question.