University of Freiburg
Dept. of Computer Science
Prof. Dr. F. Kuhn

# Algorithms and Data Structures
# Winter Term 2021/2022
# Exercise Sheet 1

## Exercise 1: Bubblesort

The following pseudocode describes the BUBBLESORT algorithm with input array $A$ of length $n$.

**Algorithm 1** BUBBLESORT($A[0 \ldots n-1]$)

```
for i = 0 to n − 2 do
    for j = 0 to n − 2 do
        if A[j] > A[j+1] then
            SWAP(j, j+1)        ▷ operation SWAP(j, j+1) swaps array entries A[j] and A[j+1]
```

(a) Assume BUBBLESORT runs on input $A = [24, 9, 15, 11, 4, 21]$. Give $A$ after the end of each iteration of the outer for-loop.

(b) Argue why BUBBLESORT is correct (i.e., array $A$ is always sorted after the algorithm is finished).

## Exercise 2: Counting Sort

The following pseudocode describes the COUNTINGSORT algorithm which receives an array $A[0 \ldots n–1]$ as input containing values in $[0..k]$. Additionally there is an Array `counts`$[0 \ldots \text{k}]$ initialized with 0.

**Algorithm 2** COUNTINGSORT($A$, `counts`)  ▷ **integer arrays** $A[0 \ldots n-1]$, `counts`$[0 \ldots k]$

```
for i ← 0 to n − 1 do
    counts[A[i]] ++                ▷ ++ is the increment operation
i ← 0
for j ← 0 to k do
    for ℓ ← 1 to counts[j] do
        A[i] ← j
        i ++
```

(a) Assume COUNTINGSORT runs on input $A = [5, 2, 3, 0, 5, 3, 4, 2, 5, 0, 1, 3, 5, 0, 0]$. Give $A$ and `counts` after the algorithm has terminated.

(b) Argue why COUNTINGSORT is correct (i.e., the algorithm has sorted array $A$ after finishing).

## Exercise 3: Implementation

(a) Implement one of the above two algorithms in a programming language of your choice (in the lecture and exercise class we will see/use Python).[1]

(b) Test your implementation with random inputs as follows. Generate input arrays *of length 10, 30, 100, 200, 300, 500, 700, and 1000* respectively, each *filled with randomly generated integer values ranging from 0 to 200*. Run the algorithm on each input and check the correctness.

(c) Implement some functionality to measure the elapsed time of the algorithm from start to finish (e.g., by using the python-module *time*). Run the algorithm again with the above inputs and note down the elapsed times. What do you think is the dependency of the running time on $n$ (and $k$, in case of the CountingSort algorithm)?

# 1 Solution: Bubblesort

## 1.1 State

$[9, 15, 11, 4, 21, 24]$

$[9, 11, 4, 15, 21, 24]$

$[9, 4, 11, 15, 21, 24]$

$[4, 9, 11, 15, 21, 24]$

$[4, 9, 11, 15, 21, 24]$

## 1.2 Correctness

First of all, elements are only swapped, so clearly the final array contains the same elements as in the original array. Let's now prove that the elements in the final array are sorted.
**Observation:** The inner loop "pulls" the current ($j^{th}$) element further to the back of the array (by repeated swaps) until either the end of the array is reached or until it finds a bigger element. In the latter case it will continue doing the same with the bigger element that has been found.
**Informal proof idea:** After iteration $i$ of the outer loop, the algorithm maintains the condition that the last $i+1$ elements in the array $A$ are the largest in the array in sorted order.[2] After the next iteration $i+1$ the algorithm ensures that the current largest of the first $n-i-1$ elements in $A$ is swapped into its correct position (due to the above observation) so that now the last $i+2$ elements in $A$ are the largest elements in sorted order.
If the above argument is too informal for the reader, we follow this up by a more formalized proof.
**Formal proof:** We argue that *after* the $i^{th}$ iteration of the outer loop, the sub-array $A[n{-}i{-}1 \dots n{-}1]$ (i.e., the last $i{+}1$ entries of $A$) is sorted and contains the $i{+}1$ largest integers. We prove this invariant by induction.
**Induction base:** During the first iteration (i.e., for $i = 0$), the largest element (or one of the largest elements in case there are many), will always be swapped to the end of the array. Thus the condition is obviously true as the single entry $A[n{-}1]$ is sorted and contains the largest element.
**Induction hypothesis:** Presume that after the $i^{th}$ iteration, $A[n{-}i{-}1 \dots n{-}1]$ is sorted and contains the $i{+}1$ largest elements.

[1] As a side-note: In this course we assume that you have some (very) basic programming skills, enabling you to implement short pseudo codes like the ones given above in a programming language of your choice. Since this course is more on the theoretical side, we will not ask much more than that in terms of programming skills. If you never attended some programming-course and/or experience difficulties to implement the above algorithms, please try to catch up using literature, tutorials and/or contact us.

[2] Such a condition is usually called a *loop invariant*. A loop invariant is a condition that holds in *every* iteration (usually immediately before or after the code within the loop is executed). A loop invariant often depends on the number of iterations.

**Induction step:** We have to show that *after* the $(i+1)^{th}$ iteration of the outer loop the sub-array $A[n-i-2 \ldots n-1]$ is sorted and contains the $i+2$ largest elements. Let $x$ be an element in $A[0 \ldots n-i-2]$ that is $(i+2)^{th}$-largest. This element (or one that has the same value) will be swapped to position $A[n-i-2]$ during that iteration (but not any further as elements in $A[n-i-1 \ldots n-1]$ are at least as large by the hypothesis). Therefore, and due to the induction hypothesis $A[n-i-2]$ is sorted and contains the $i+2$ largest elements.

### 1.3 Implementation

```
def swap(A, x, y):
  A[x], A[y] = A[y], A[x]

def bubblesort(A):
  n = len(A)
  for i in range(0, n-2 + 1):
    for j in range(0, n-2 + 1):
      if A[j] > A[j+1] :
        swap(A, j, j+1)
```

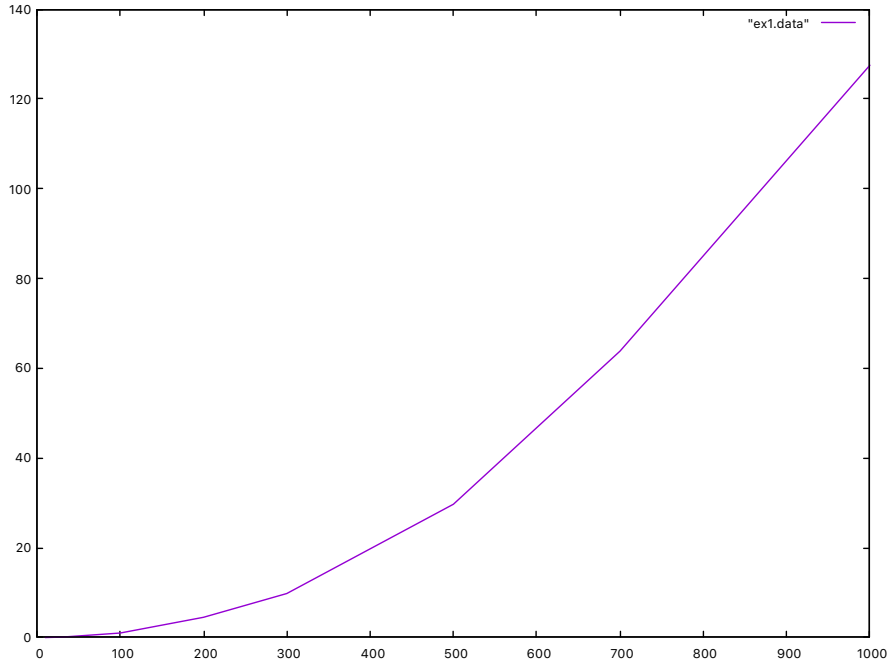
### 1.4 Plot

Python code to measure time:

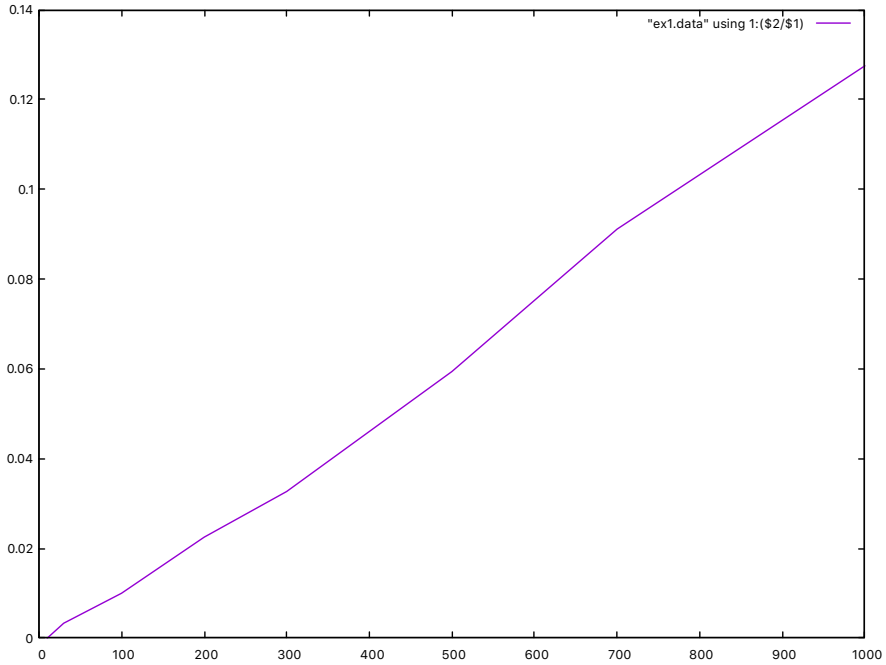
```
import random
import time

def measure():
  for n in [10, 30,100, 200, 300, 500, 700, 1000]:
    array = [random.randint(0,200) for k in range(0,n)]
    start_time = time.time()
    bubblesort(array)
    run_time = (time.time() - start_time) * 1000
    print("%d\t%.1f" % (n,run_time))
```

By plotting the data, we get the following:




If we now plot $run\_time/n$ as a function of $n$, we get the following:




This suggests that $run\_time/n$ is linear in $n$, and thus that $run\_time$ is quadratic in $n$.

# 2 Solution: Counting Sort

## 2.1 State

$A = [0, 0, 0, 0, 1, 2, 2, 3, 3, 3, 4, 5, 5, 5, 5]$, `counts` $= [4, 1, 2, 3, 1, 4]$.

## 2.2 Correctness

The algorithm simply counts the number of occurrences of each key $j$ in `counts`$[j]$. Subsequently it writes the keys into the array in ascending order, thus it must obviously be sorted. The (multi-)set of keys in array $A$ after sorting is also the same as before, since for each key, we write exactly the same number of keys into the array that we counted before.

## 2.3 Implementation

```
def countingsort(A, counts):
  n = len(A)
  k = len(counts) - 1
  for i in range(0, n-1 + 1):
    counts[A[i]] += 1
  i = 0
  for j in range(0, k+1):
    for l in range(1, counts[j]+1):
      A[i] = j
      i += 1
```

## 2.4 Plot

To better understand the dependency on $n$ and $k$, we use the following code (abd run it for different values of $k$):

```
import random
import time
import sys
```

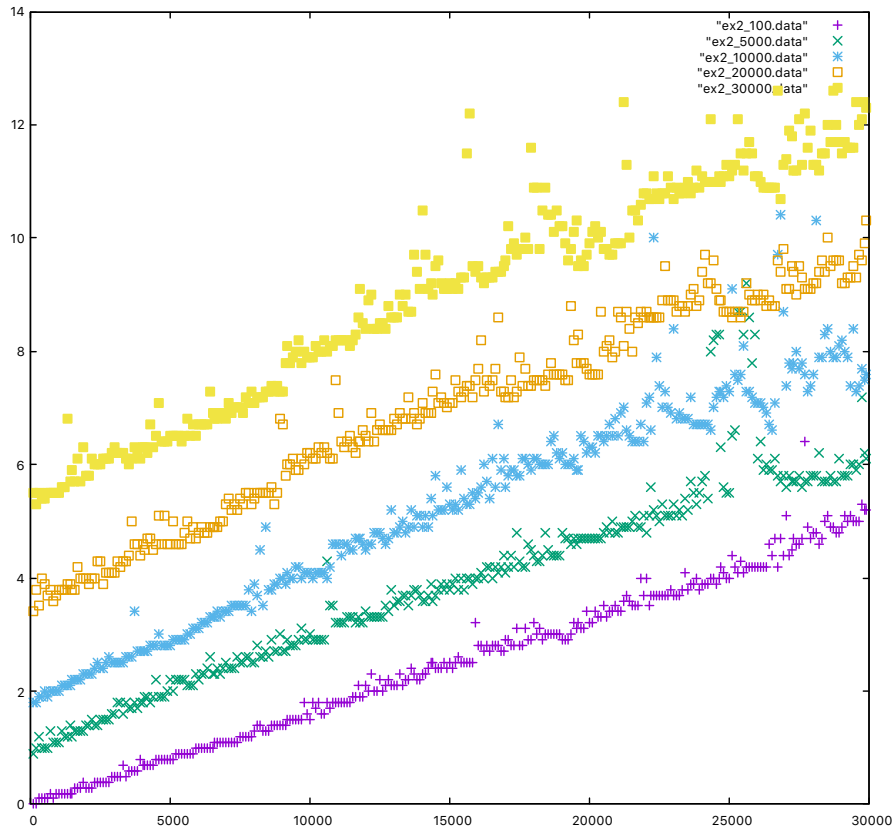
```
def measure(k):
  for n in range(100,30000, 100):
    array = [random.randint(0,k) for i in range(0,n)]
    counts = [0]*(k+1)
    start_time = time.time()
    countingsort(array,counts)
    run_time = (time.time() - start_time) * 1000
    print("%d\t%.1f" % (n,run_time))

measure(int(sys.argv[1]))
```

By running the following bash script we generate the required data:

```
for i in 100 5000 10000 20000 30000; do
    python3 ex2.py $i > ex2_$i.data
done
```

By plotting the data, we get the following:




We can notice that, for all values of $k$, the running time appears to be linear in $n$, but it also seems that the whole data has a vertical offset that linearly depends on $k$, and this suggests that the running time is linear in $n + k$.