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

Algorithms and Data Structures
Winter Term 2019/2020
Sample Solution Exercise Sheet 1

Remark: For this exercise sheet, watch the first two video lectures given on the lecture website. Ignore the part on organization in the video. If you have questions about the exercise sheet (or this course in general) you can ask them in the forum. The link for the forum is on the website.

Exercise 1: Bubblesort

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

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

```
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 $A = [27, 8, 19, 5, 23, 12]$. Give A after the end of each iteration of the outer for-loop.

(b) Give the (worst-case) runtime for BUBBLESORT as a function of n. Explain your answer.

Remark: To simplify things, you may assume that each cycle of a loop (inner or outer) takes one unit of time and neglect the time required for other operations.

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

Sample Solution

(a) $[27, 8, 19, 5, 23, 12]$
$[8, 19, 5, 23, 12, 27]$
$[8, 5, 19, 12, 23, 27]$
$[5, 8, 12, 19, 23, 27]$

(b) The outer loop makes $n-1$ iterations. For each iteration of the outer loop we have $n-1$ iterations of the inner loop. So the total number of iterations of any of the two loops is

$$T(n) = (n-1) + (n-1)(n-1) = n(n-1)$$

Next week we will se that the above runtime is in the asymptotic *runtime class* $\mathcal{O}(n^2)$.

(c) **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.[1] 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.

Induction step: We have to show that *after* the $(i+1)^{th}$ iteration of the outer loop the sub-array $A[n-i-2 \dots n-1]$ is sorted and contains the $i+2$ largest elements. Let x be an element in $A[0 \dots n-i-2]$ 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 $A[n-i-1 \dots n-1]$ 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.

Exercise 2: Counting Sort

The following pseudocode describes the CountingSort algorithm which receives an array $A[0 \dots n-1]$ as input containing values in $[0..k]$. Additionally there is an Array `counts`$[0 \dots k]$ initialized with 0.

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

```
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) Give the (worst-case) runtime of CountingSort as a function of n and k. Explain your answer.

Remark: To simplify this, you may assume that each cycle of a loop (inner or outer) takes one unit of time. You may neglect the time required by other operations.

(c) Argue why CountingSort is correct (i.e., the algorithm has sorted array A after finishing).

[1] 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.

Sample Solution

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

(b) The first loop iterates n times. The second (outer) loop has exactly $k + 1$ iterations. The third (inner loop) is a bit trickier since it depends on the values that are stored in the array `counts`. We count the total number of iterations that it will execute over *all* iterations of the outer loop.

In *total* we will have $\sum_{j=0}^{k} \texttt{counts}[j]$ iterations. Since $\texttt{counts}[j]$ represents the number of times the key j occurs in A, we have that $\sum_{j=0}^{k} \texttt{counts}[j] = n$. Hence the *total* number of iterations of the inner loop is n. Thus we have

$$T(n, k) = n + k + 1 + n = 2n + k + 1$$

This is in the asymptotic run time class $\mathcal{O}(n + k)$ (more on that next week).

(c) The algorithm simply counts the number of occurrences of each key j in $\texttt{counts}[j]$. 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.

Exercise 3: Implementation

(a) Implement the above two algorithms in a programming language of your choice (in the lecture and exercise class we will see/use Python).[2]

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

(c) Implement some functionality to measure the elapsed time of your algorithms from start to finish (e.g., by using the python-module *time*). Run the algorithms again with the above inputs and note down the elapsed times. What do you observe?

Sample Solution

(a) C.f., solution files.

(b) C.f., solution files.

(c) C.f., Figure 1.

1. One observation is that the running time of BUBBLESORT increases significantly which is what we would expect since the runtime increases *quadratically* in n ($\mathcal{O}(n^2)$).
2. The running time of COUNTINGSORT on the other hand does not change much. This is due to the fact that the runtime is *linear* in n *and* k ($\mathcal{O}(n + k)$). Since $k = 1000$ in all cases (and $n \leq 1000$) the increase in n does not affect the runtime much.
3. A third observation is that BUBBLESORT is much faster than COUNTINGSORT for the case $n = 10$. This is due to the fact that BUBBLESORT does not depend on $k = 1000$ and can therefore profit much more from the small input size $n = 10$.

[2] 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.

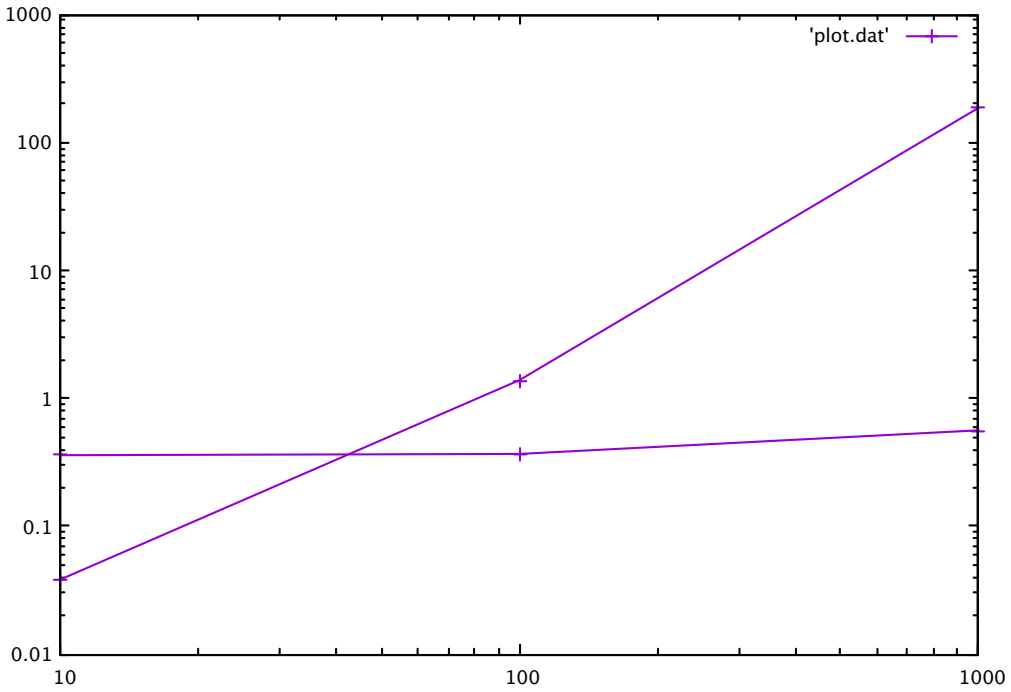

Figure 1: The plot shows the runtime (y-axis) over the array size (x-axis) of the six tests (both sorting algorithms run on the three different array sizes 10,100,1000) on a log scale. The test results belonging to the same sorting algorithm are connected by a line. The horizontal plot belongs to CountingSort.