Marburg University

Department of Computer Science
Prof. Dr. Sebastian Wild

Date: 2025-10-07
Version: 2025-11-11 00:45

Exercise Sheet 4 for Effiziente Algorithmen (Winter 2025/26)

Hand In: *Until 2025-11-14 18:00, on ILIAS.*

Disclaimer: English translations of our exercise sheets are provided as as best-effort service; in case of doubt, the German versions take precedence.

Problem 1

$20 + 10 + 10$ points

Assume that the Quicksort algorithm presented in the lecture always selects the *last* element as pivot. How does the Θ-class of the expected running time of the algorithm behave on the following inputs?

Give the extreme cases for each case, and analyse them by describing how Quicksort proceeds (swaps, algorithm state at each recursion, etc.). Discuss also any negative effects that occur for less extreme cases. If necessary, propose improvements to the quicksort implementation that counteract these effects.

a) Keys appear multiple times, i.e., duplicates may exist.

b) The input is partially sorted.

c) The input is partially sorted, but in reverse order.

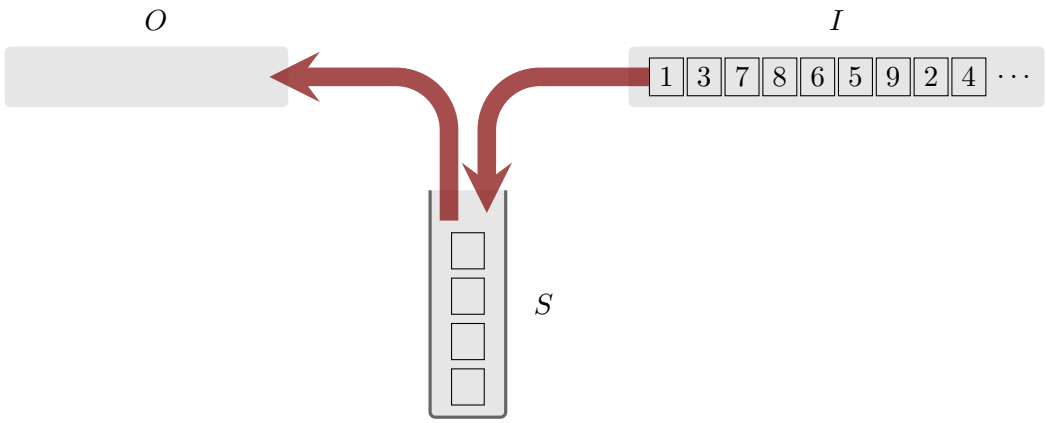
Problem 2

$10 + 30 + 20$ points

Consider the following streaming model:

Given n (pairwise distinct) elements in an input stream I, you may repeatedly either consume the next element in I, or output an element to the output stream O. There is no other way to either modify or access I or O. You may think of the two streams as two queues, where I supports only *dequeue* operations, and O only *enqueue* operations. The number n of elements is known in advance.

In addition to I and O (and, perhaps, a constant number of local variables), the elements may also be stored in a *stack* S. Thus at any point you can do one of the following: remove an element from I or S; or insert an element into O or S.



Note: Comparisons are only allowed between the element S on top of the stack and the next element coming from the stream I. Thus, between any two non-redundant operations, the elements must be moved by an operation $I \to S$ or $S \to O$.

a) Show that in this model, it is *not* always possible to generate a sorted output stream. In other words, for some permutation n of the elements in I, it is impossible to insert the elements into O in increasing order.

   You may assume a large enough n in this proof.

b) Assume that O can be fed back into I as input for another pass, i.e. I and O are connected and form one large queue.

   We assume for simplicity that a pass must finish before the next one starts, i.e. before an element can be removed from I a second time, all n elements must have been emitted to O. Hence elements cannot "overtake" each other, and each execution has a well-defined number k of rounds.

   Design a sorting algorithm for this model, i.e. a sequence of $I \to S$ or $S \to O$ operations that correctly sorts every possible input. No other operations for rearranging data are available, but your algorithm can use any amount of time or memory to compute the next operation. Comparisons of Comparisons of $S.top()$ and $I.front()$ are free.

   Analyse the worst case for the number of rounds for your algorithm to sort n elements. To receive full credit, the algorithm must achieve a runtime of $k \in O(\log n)$.

   Hint: You may draw inspiration from sorting on tape drives*:*
   `https://en.wikipedia.org/wiki/Merge_sort#Use_with_tape_drives`.

c) Give a nontrivial lower bound for k that *any* sorting algorithm must use in this model.

Problem 3

30 + 10 + 20 points

An array $A[0..n)$ is *d-deletion sortable* if there exist positions $0 \leq i_1 < i_2 < \cdots < i_d < n$ such that, after deleting the elements at positions $i_1, \ldots, i_d$ from A, the resulting sequence is sorted. For example,

$$2, 4, 1, 6, 7, 5, 8, 12, 0$$

is 3-deletion sortable (by removing the elements $1, 5, 0$), but is *not* 2-deletion sortable.

Below we assume that the array $A[0..n)$ is d-deletion sortable. You may also assume that the elements of A are pairwise distinct.

a) Design an adaptive sorting algorithm for A, which receives as input A and a sorted array $D[0..d)$ containing the positions $i_1, \ldots, i_d$ which make A d-deletion sortable.

   Under the assumption that $d \ll n$ (i.e. d is "much smaller" than n), your algorithm should run in $o(n \log n)$ time; a full solution would sort a $\sqrt{n}$-deletion sortable sequence $A[0..n)$ in $O(n)$ time.

   Describe your algorithm (either in text or pseudocode), and analyse its Θ-asymptotic running time.

b) For which size of d does your solution use $\omega(n)$ time?
   For which size of d does your solution use $\Omega(n \log n)$ time?

c) Design an algorithm like in part a) without receiving d or the set of positions as input.

   Hint: Can you find a set of positions I such that, after removing the elements at those positions, A becomes sorted, while keeping the size of I small?