UNIVERSITY OF LIVERPOOL

Department of Computer Science
Sebastian Wild

Date: 2021-02-24
Version: 2021-04-06 14:42

Tutorial 3 for COMP 526 – Applied Algorithmics, Spring 2021

Problem 1 (Finding the first k elements)

Design an algorithm for the following problem:

> Given array $A[0..n-1]$ of n (pairwise distinct) elements and a number $k \in \{0, \dots, n-1\}$, rearrange the elements so that the first k positions contain the k smallest elements in sorted order.

Formally, after the execution we require

$$A[0] \le A[1] \le \cdots \le A[k-2] \le A[k-1] \qquad \text{and} \qquad \forall i \in \{k, \dots, n-1\} : A[k-1] \le A[i].$$

to hold. The elements can be any objects; only assume a total order of the elements (given via a suitably overloaded operator $<$).

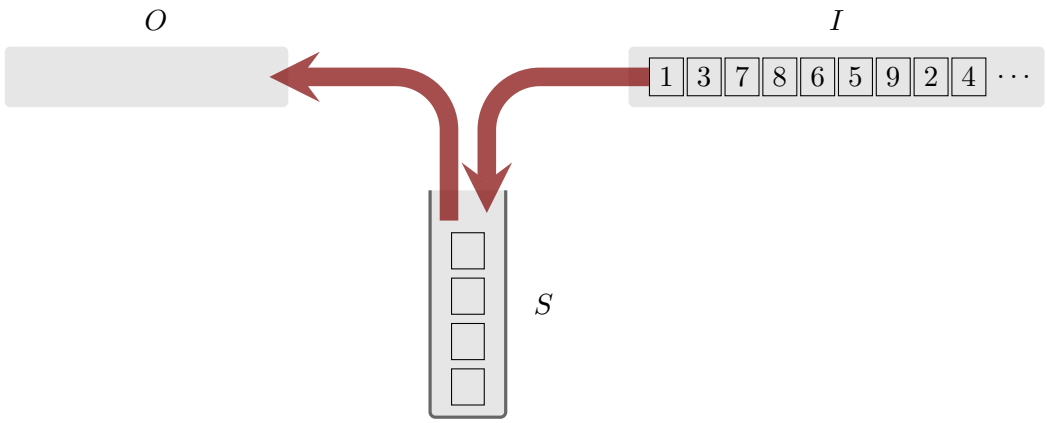
A full solution must have running time in $O(n + k \log n)$ and use $O(1)$ extra space. Correct algorithms violating one or both requirements are a valuable intermediate step.

Bonus: Can you give an algorithm with running time in $O(n + k \log k)$?

Problem 2 (Sorting with Stacks)

In this exercise, we consider sorting in a specific streaming model: You are given n (pairwise distinct) elements as an input stream I, from which you can obtain the elements one at a time, and you are supposed to put your output into an output stream O, again one at a time. You cannot otherwise gain access or modify I and O. You can imagine the streams as two queues, where I allows only the dequeue operation and O allows only enqueue. The total number n elements in the input is known to you up front.

Apart from the source and sink queues I and O (and potentially a constant amount of local variables), your only means of storing elements is *one stack* S. Note that this means that at any point in time, you can only do the following operations: Take an element from I or from the top of S; put the element into O or onto S.



Remark: Comparisons are only possible between the element currently at the top of S and the element currently at the front of I. Hence between any two (non-redundant) comparisons we must have a move "$I \to S$" or "$S \to O$".

a) Prove that in the above model, it is *not* always possible to produce a sorted output stream, i.e., for some permutations of the n elements in I, we cannot insert the elements into O in ascending order.

   You may assume a "large enough n" for the purpose of this proof.

b) Now assume O is used as input for a second round, i.e., O and I are connected and form one large queue.

   We assume for simplicity that we always finish one round of moving the n items through the stack before starting the next round, i.e., before we are allowed to take an element the second time out of I, we must have put all other elements into O first. That means, elements cannot lap each other and any execution has a well-defined number of rounds k.

   Design a sorting algorithm for this model, i.e., a program that outputs a sequence of moves "$I \to S$" or "$S \to O$". These operations are the only means of rearranging the data, but your algorithm may take any amount of time and space for computing the next move and can compare the elements $S.top()$ and $I.front()$ for free.

   Analyze how many rounds your algorithm needs in the worst case for sorting an input of n (distinct) elements. For full credit, your algorithm must achieve $k \in O(\log n)$.

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

   Bonus (hard): Find an algorithm with $k \leq \lceil \log_2 n \rceil$.

c) Prove a nontrivial lower bound on k valid for *any* sorting method in the model.