UNIVERSITY OF LIVERPOOL

Department of Computer Science
Sebastian Wild

Date: 2020-02-26
Version: 2020-02-27 13:35

Tutorial 3 for COMP 526 – Applied Algorithmics, Winter 2020

—including solutions—

It is highly recommended that you first try to solve the problems on your own before consulting the sample solutions provided below.

Disclaimer: This document gives much more than the straight-forward answers to the problems; instead it discusses several alternative solutions, and contains remarks going beyond what was asked (these are typeset in blue).

This is intended to provoke thought among those interested to learn more about these problems, but as a result, this document has reached a scary length that is not at all indicative of the expected length of satisfactory answers.

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 $O(n + k \log k)$?

Solutions for Problem 1 (Finding the first k elements)

Variant 1: Truncated Heapsort

- Idea:
  We use a min-oriented binary heap (as in heapsort) but we stop the sort-down phase after k steps.
- Code:

```
1  procedure partialSort(A)
2      // build min−heap
3      Q := new minHeapPQ()
4      for i = 0, ..., n − 1
5          Q.insert(A[i])
6      // extract k smallest elements and put them at end of A
7      for i = 0, ..., k − 1
8          A[i] := Q.deleteMin()
9      for i = k, ..., n − 1
10         A[i] := Q[i − k + 1] //heaps stored in 1−based array
```

- Correctness:
  The elements extracted from the min heap using deleteMin are the smallest of the stored elements and come in sorted order. The last for-loop simply copies the heap contents to the second part of A.
- Analysis:
  This algorithm runs in $O(n + k \log n)$ time since it does k deleteMins on a heap of size $\leq n$; the space usage is linear.

Bonus:

We will show that actually any solution runs in $O(n + k \log k)$ time, as well:

- For $k \leq n/\log n$, we have $k \log n \leq n$, so $n \leq n + k \log n \leq 2n$ and similarly $n \leq n + k \log k \leq n + k \log n \leq 2n$, so the achieved and the required bound are both within a constant factor of n, and thus within a constant factor of each other: $\Theta(n + k \log n) = \Theta(n + k \log k)$
- For $k > n/\log n$, we have

$$\begin{aligned} k \log n &\geq k \log k \\ &> k \log\left(\frac{n}{\log n}\right) \\ &= k \log n - k \log \log n \\ &= k \log n \left(1 - \frac{k \log \log n}{k \log n}\right). \end{aligned}$$

Since

$$\frac{k \log \log n}{k \log n} = \frac{\log \log n}{\log n} \to 0, \qquad (n \to \infty),$$

there is a n_0, so that for all $n \geq n_0$ holds $\frac{\log \log n}{\log n} \leq \frac{1}{2}$. (In fact, $n_0 = 20$ suffices: `http://www.wolframalpha.com/input/?i=plot+log2(log2(n))%2Flog2(n)+for+n+%3D2+to+100`)

We thus have for $n \geq n_0$:

$$k \log n \geq k \log k \geq \frac{1}{2} k \log n.$$

So again both terms are within a constant factor of each other.

Together, we find that indeed $\Theta(n + k \log n) = \Theta(n + k \log k)$ for all k.

Variant 2: Select, partition and sort prefix We determine the kth smallest element x using Quickselect (in $O(n)$ time on average). This already leaves the array partitioned into the $k-1$ elements smaller than x and the elements larger than x. Then we sort the prefix of length k in $O(k \log k)$ time by Heapsort.

It uses $O(1)$ extra space, if we implement Quickselect with tail-recursion elimination.

This method overall runs in $O(n + k \log k)$ *expected* time unless we use a worst-case linear-time time selection algorithm. The running time is achieved also in the worst case, but it is not possible with the methods from class to retain the constant space.

Variant 3: partial Quicksort A nice alternative of Variant 2 is to use Quicksort and pursue all recursive calls that contain some of the first indices (this always includes the left call, and sometimes additionally the right call).

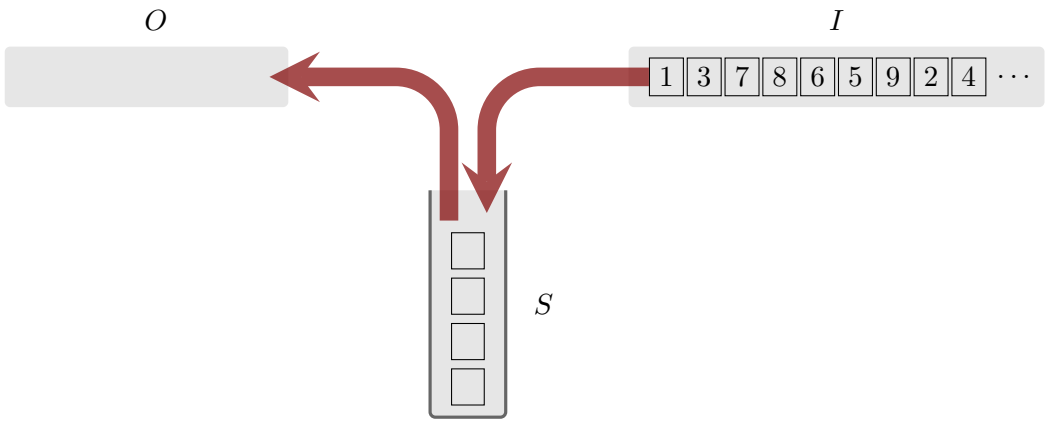
One can prove that the expected overall running time is essentially the same as Variant 2.[1]

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.

[1] Martínez, Conrado. 'Partial quicksort.' Proceedings of the 1st ACM-SIAM Workshop on Analytic Algorithmics and Combinatorics. 2004.



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.

Solutions for Problem 2 (Sorting with Stacks)

This is known as stack-sortable permutations and rather well-studied under this name in combinatorics. (Warning: There are different notions of "k-stack sortability" in the literature.)

For working in the model, we program *drivers* that only use the public interface of S, I and O, i. e, the stack and queue ADT operations.

a) There are several possible arguments.

- A simple counting argument.

  Ultimately, the stack-machine applies a certain permutation to the input. For this result, we can restrict our attention to the two operations $I \to S$ and $S \to O$. So in each step, we either push an element form I onto S, or we pop an element from S and put it into O. In every such step, either the size of I shrinks by one or the size of O grows by one, so we need exactly $2n$ steps to pass all n elements through the stack.

  Overall, this gives at most $2^{2n} = 4^n$ different possible executions (permutations that are applied to the input), but there are $n!$ different input permutations. For $n \geq 9$ when $4^n < n!$, some permutations thus cannot be sorted because they are different, but receive the same treatment.

- Counterexample.

  The shortest permutation that cannot be sorted using a stack is $2, 3, 1$; no matter how we use the stack, we cannot output 1 first without getting 2 and 3 out of order.

- Sorting lower bound.

  Similar to the first argument, we observe that we can do at most $2n$ non-redundant comparisons during one execution since we have to move some elements after each comparison before we can learn more about the input. Since for $n \geq 9$ we have $2n < \log_2(n!)$, the lower bound for comparison-based sorting, we cannot sort all inputs correctly.

b) There are several solutions.

Quicksort If we allow storing and processing the elements outside of the stack-machine, we can also simulate Quicksort:

We select the first element as "pivot". Then, we put all elements greater than the pivot into S, all others into O. Once all elements have been taken out of I, we empty S into O. This explains how we can partition. But we cannot easily use recursion in this model, so we have to work through the Quicksort recursion tree "sideways", i.e., level by level. We can do that by keep track of the sizes of all segments we created up to now. Note that the stack model only restricts how we

are allowed to rearrange the elements to sort; in the given formulation, there is nothing that stops us from using external memory for other purposes.

In each of the following rounds, we repeat the above process for the segments so far (unless they are single elements). In expectation (for random inputs), $\mathcal{O}(\log n)$ rounds will sort the elements.

Mergesort We can also simulate a variant of mergesort. By moving one run into S and keeping the other one in I, we can simulate the merging procedure. Catch one: S reverses the run, so before we get started with merging, we will have a preparatory round the reverses every other run.

Catch two: The mergesort procedure presented in class is not suitable, again because of the recursion. But we can – again – replace the recursion by a level-wise order: initially, all elements are solitary. In the first round, we merge them into pairs. In the next round, we merge pairs into runs of length 4, and so on. This is called bottom-up mergesort.

- Idea:
  The stack can be used to merge a run on the stack with one still in I. We can thus simulate (bottom-up) Mergesort.

  How to merge:
  If we have two runs in I, the first in descending order, the second in ascending order, we can push the first onto S, thereby reversing its order. Then we can alternatingly pop resp. take from I and merge the two runs into one large ascending run.

  Since the run gets reversed while being put into S, we need every other in descending order. To obtain runs alternatingly in ascending and descending order, we spend a second round after merging, in which we reverse every other run.

- Code:
  We assume that O and I are given as one connected queue I. The following code uses the typical ADT interface for queue and stack to sort using I and S.

```
1  mergesortDriver(I, S, n)
2      for (len = 1; len < n; len *= 2) // 2 rounds each
3          // reverse every other run
4          for (offset = 0; offset < n; offset += 2*len)
5              len1 = min { len, n - offset }
6              len2 = max { 0, min { len, n - offset - len } }
7              // reverse first run
8              for (i = 0; i < len1; ++i) S.push(I.dequeue) // I → S
9              for (i = 0; i < len1; ++i) I.enqueue(S.pop()) // S → O
10             // copy second run
11             for (i = 0; i < len2; ++i)
12                 O.enqueue(I.dequeue()) // I → S, S → O
```

```
13          // first round finished, now merge runs
14          for (offset = 0; offset < n; offset += 2*len)
15              len1 = min { len, n - offset }
16              len2 = max { 0, min { len, n - offset - len } }
17              for (i = 0; i < len1; ++i)
18                  S.push(in.dequeue()); // S → I
19              int i1 = 0, i2 = 0
20              while (i1 < len1 || i2 < len2)
21                  if (i1 == len1)
22                      ++i2; I.enqueue(I.dequeue()) // I → S, S → O
23                  else if (i2 == len2)
24                      ++i1; I.enqueue(S.pop())  // S → O
25                  else if (S.top() <= I.front()) // compare(S, I)
26                      ++i1; I.enqueue(S.pop()) // S → O
27                  else
28                      ++i2; I.enqueue(I.dequeue()) // I → S, S → O
29              end while // one merge finished
30          end for
31          // second round finished
32      end for
```

Note that we never retrieve any actual value from the stack-machine, but only uses direct comparisons between $S.top()$ and $I.front()$.

The given code uses n to control the code. This avoids explicit checks for corner cases and thus makes the code more readable; it is not required though and could be replaced by empty-checks on S and I. This algorithm hence effectively allows to operate blindly.

- Correctness:
  The overall structure is bottom-up mergesort. In the first round, we make sure that the first run in every pair of runs is reversed. Hence in the second round, merges can be done by pushing the first run into S thereby reversing it again. So the correctness follow from the correctness of bottom-up Mergesort and merge.
- Analysis:
  The outer loop (line 2) runs $\lceil \log_2 n \rceil$ times, and each iteration executes 2 rounds of running all n elements from I through S back into I (resp. O). Hence we need $k = 2\lceil \log_2 n \rceil = O(\log n)$ rounds.

  Note: If n is not known up front, we have to spend another round to count the elements.

  Unlike the above Quicksort version, this code uses constant extra space and constant time between any two operations send to the stack-machine.

Apart from the reversal needed for the stack, the above code borrows tricks from the setting of sorting data stored on tape drives which essentially behave like large queues. Unlike in our model, having additional tapes is allowed and can be used

to speed up sorting. Tape sorting methods typically work without knowing n up front (and without using an expensive extra round to determine it).

Bonus:

- Idea:
  We can indeed simulate a version of bottom-up mergesort using $\lceil \log n \rceil$ rounds by figuring out the correct order for runs up front.
- Code:
  Below is Java code illustrating the idea. It take $I = O$ as input and sorts the elements therein using a stack. Internally, we move elements in each round from `in` (I) into another queue `out` (O) and then copy them back to I for the next round. we could alternatively put elements directly back into I and use counter to determine the end of the round.

  The code allocates an array to store the direction of runs. At the expense of more complicated code, we could get rid of that and use that the sequence of run directions forms the *Thue-Morse sequence*, whose elements can be computed explicitly (`https://en.wikipedia.org/wiki/Thue%E2%80%93Morse_sequence`).

```
1  public static <Elem> void stackMergesort(Queue<Elem> in, ↩
       Comparator<Elem> c) {
2      int n = in.size();
3      Deque<Elem> stack = new ArrayDeque<>(n);
4      Queue<Elem> out = new ArrayDeque<>(n);
5      boolean ascStart = (ceilLog2(n) & 1) != 0;
6      for (int len = 1; len < n; len *= 2) {
7          boolean[] asc = runDirection(n / (2 * len) + 1, ascStart);
8          for (int merge = 0; merge <= n / (2*len); ++merge) {
9              for (int i = 0; i < len && !in.isEmpty(); ++i)
10                 stack.push(in.poll());
11             int inLen = min(in.size(), len);
12             for (int i = 0; i < 2*len; ++i) {
13                 if (stack.isEmpty() && in.isEmpty()) break;
14                 if (stack.isEmpty()) { out.add(in.poll()); --inLen; }
15                 else if (inLen == 0) out.add(stack.pop());
16                 else {
17                     Elem x = stack.peek(), y = in.peek();
18                     if (asc[merge] && less(x,y,c)
19                             || !asc[merge] && less(y,x,c))
20                         out.add(stack.pop());
21                     else
22                         { out.add(in.poll()); --inLen; }
23                 }
24             }
25         }
26         assert in.isEmpty();
27         for (int i = 0; i < n; ++i) in.add(out.poll());
28         ascStart = !ascStart;
29     }
```

```
30 }

32 public static boolean[] runDirection(int n, boolean first) {
33     boolean[] res = new boolean[n];
34     res[0] = first;
35     for (int l = 1; l < n; l *= 2)
36         for (int i = l; i < 2 * l && i < n; ++i)
37             res[i] = !res[i - l];
38     return res;
39 }

41 public static int ceilLog2(int n) {
42     if (n <= 0) throw new IllegalArgumentException();
43     int i = 0, twoToI = 1;
44     while (twoToI < n) { ++i; twoToI <<= 1; }
45     return i;
46 }
```

- Analysis:
  We use one round for each level / run-length, so we use $\lceil \log_2 n \rceil$ rounds.

c) Again, there are several options.

- Counting argument:
  The k rounds perform independent movements of the elements, so we can upper bound the number of different runs by $(4^n)^k$ (recall from a): 4^n is the number of different runs for one round. We thus need $4^{nk} \geq n!$ to be able to sort, which implies $k \geq \log_4(n!)/n \leq \frac{1}{2}\frac{n \log_2(n)}{n}$.

  This bound might not be tight, but the actual number of rounds needed seems to be an open problem.

  Note that this bound even holds if we allow the algorithm the superpower to "know" the permutation up front, so that it only has to figure out the shortest sequence of movements to transform it into the sorted permutation.

- Sorting Lower Bound:
  This applies only to the strict version of the model.

  Since we have to do (at least) one move between any pair of non-redundant comparisons, we get at most $2n$ comparisons per round (see also a)) and $\leq k \cdot (2n)$ comparisons from k rounds. As the lower bound for comparison based sorting dictates $k(2n) \geq \log_2(n!)$, we find $k \geq \frac{1}{2}\frac{\log_2(n!)}{n} \leq \frac{1}{2}\log_2 n$.