1 Solutions to the midterm exam

Solution 1.1

First solution It was described in the class how to find largest element in \( n - 1 \) steps. You start with \( \frac{n}{2} \) disjoint pairs and compare them, after you assume only winners there are about \( \frac{n}{4} \) of them and again make \( \frac{n}{8} \) disjoint pairs, in the end you get largest element and it will take you \( n - 1 \) steps.

How to find 3 smallest elements:

1. round
In first round we find the largest element of all \( n \) elements as we described above. It takes \( n - 1 \) comparisons.

2. round
The second largest element has to be compared in some point of 1. round with the second, but each element from first round was in at most \( \log(n) \) comparisons. It means we can use algorithm for finding largest element again on this \( \log(n) \) elements, that takes \( \log(n) - 1 \) comparisons.

3. round - slow
We can notice that the third largest element had to lost in 1. round with the largest or second largest element, in the first round there is \( 2 \log(n) \) such elements and we use the algorithm to find largest element which would cost us \( 2 \log(n) - 1 \) comparisons.

3. round - faster
Instead of slow 3. round we can do it faster. We still have to notice that there is \( \log(n) \) elements which lost to second largest element in 1. round and which can be third largest, but we can notice that in 2.round assume all elements which lost to 1. element in 1. round, so we can now assume only elements which were compared to second largest in the 2. round and there is \( \log \log(n) \) of them, so all together 3.round takes \( \log(n) + \log \log(n) - 1 \) comparisons.

Naive way which uses 3.round - slow algorithm takes \( n + 3 \log(n) \) comparisons, but if we use the 3.round - faster we can get to \( n + 2 \log(n) + \log \log(n) - 3 \) comparisons.

Solution 1.2

Let \( A \) is sorted array with input numbers. We split it into \( \frac{n}{k} \) subarrays of size \( k \), see Fig. 1. If we permute every subarray (each has \( k! \)) and merge them together we get \( k \)-sorted array. So for every permutation in every sub-array we get different permutation of original array and they all are sorted. So what is number of such permutations? One way how to think about it is as \( \frac{n}{k} \) digit number with \( k! \) base? How many such number are there? It is \( (k!)^\frac{n}{k} \). Now we just use the same argument as which is used in the proof that every comparison based sorting algorithm needs \( \Omega(n \log(n)) \) comparisons. Which in this case get
us \frac{\log(k!)}{k} \geq \frac{n}{k} \log(k^{k/2}) = \frac{n}{k} \log(k) = \Omega(n \log(k)).

\begin{verbatim}
Figure 1: An illustration of splitting the array.
\end{verbatim}

\textbf{Solution 1.4}

a) Assume that each digit has credit. In the beginning it is 0 dollars, we would like to satisfy the following condition. In each step (of the first \( n \) increment operations) we want to have 1 dollar on each bit which contains 1. We now prove that if you give me for each operation 2 dollars and each flip of bit costs 1 dollar, I can manage to do \( n \) INCREMENT operations, see Fig. 2.

Our induction hypothesis is that we can INCREMENT up to number \( m \), now we want to show we can do it also for \( m + 1 \). When doing INCREMENT we just flip all the initial 1s and flip next digit to 1. Flipping of 1s is paid from their credit and flipping next digit is paid by current operation and because every operation has 2 dollars we can put one dollar into credit of the highest affected bit. That already conclude that the total cost of the first \( n \) INCREMENT operations costs \( O(n) \). The second part is completely analogical, we can notice that DECREMENT is same as INCREMENT only it treats 1s as 0s and opposite.

We can see that this takes \( O(n) \) from symmetry, we flip the same amount of bits when doing \( n \) INCREMENT as we do when doing \( n \) DECREMENT. That conclude the proof.

\begin{verbatim}
before INCREMENT after INCREMENT
\end{verbatim}

\begin{verbatim}
Figure 2: An visual explanation of dollar credit.
\end{verbatim}

b) The first \( n \) operations we set as INCREMENT, after these \( n \) operations we get the number \( 2^l \), which has binary representation 100...0, where there is one 1 and \( l \) times 0. The rest of the \( n \) is just alternating DECREMENT and INCREMENT, which means alternate two numbers in the array \( 2^l \) and \( 2^l - 1 \). The total cost of first \( n \) operations is \( n \), every INCREMENT flips at least once. In the second half you can notice that every operation flips \( l + 1 \) bits, therefore the second half flips \( ln \) bits which least to lower bound \( n + ln \) which is \( \Omega(ln) \).