- This problemset has *three* questions.
- To get the credit for questions marked as SPOJ, you must get them accepted on `http://www.spoj.com/AOS`, but you **don't** have to send any explanation!
- For other questions, either send the solutions to `gawry1+aos@gmail.com`, or leave them in the envelope attached to the doors of my office (room 321).

1. Prove that the lexicographical order is a total order:

   Solution: Recall that $s \leq t$ if either s is a prefix or t, or s and t agree on first $i-1$ letters, and then $s[i] < t[i]$. To make life simpler, assume that all strings are zero-terminated. Then $s \leq t$ if either $s = t$ or $s[1] = t[1]$, ..., $s[i-1] = t[i-1]$ and then $s[i] < t[i]$.

   (a) for any string a: $a \leq a$,

   Solution: $a = a$ so $a \leq a$.

   (b) for any strings a and b: if $a \leq b$ and $b \leq a$ then $a = b$,

   Solution: If $a = b$, then $a = b$. Similarly for $b = a$. The remaining case is that $a[1] = b[1]$, $a[2] = b[2]$, ..., $a[i] < b[i]$ and at the same time $a[1] = b[1]$, $a[2] = b[2]$, ..., $a[i'] > b[i']$. But then $i = i'$, and we get that $a[i] < b[i]$ and $a[i] > b[i]$ at the same time, which is absurd.

   (c) for any strings a, b, and c: if $a \leq b$ and $b \leq c$ then $a \leq c$,

   Solution: If $a = b$ or $b = c$ the claim is obvious. So, assume that $a[1] = b[1]$, $a[2] = b[2]$, ..., $a[i] < b[i]$ and at the same time $b[1] = c[1]$, $b[2] = c[2]$, ..., $b[i'] < c[i']$. Then we have three cases, $i < i'$, $i = i'$, and $i > i'$. In all of them we get that $a \leq c$.

   (d) for any strings a and b: either $a \leq b$ or $b \leq a$.

   Solution: If $a = b$, the claim is obvious. Otherwise $a[1] = b[1]$, $a[2] = b[2]$, ..., $a[i] \neq b[i]$. Depending on the relation between $a[i]$ and $b[i]$ we get that $a \leq b$ or $b \leq a$.

2. The goal of this question is to make sure that you are familiar with binary search. The first part is straightforward, while the second requires some additional insight. Try to solve both!

   (a) You are given a sorted sequence of numbers $a_1 \leq a_2 \leq \ldots \leq a_n$. You can access any of them in constant time. Show how to check if the sequence contains a number x in just $\mathcal{O}(\log n)$ steps. Provide either some pseudocode or a clear description of your method.

Solution: Check if $a_1 = x$ or $a_n = x$. Otherwise, assume that $a_1 < x$ and $x < a_n$, if not terminate. Start with $\ell = 1$ and $r = n$. While $\ell + 1 < r$, set $m = \lfloor \frac{\ell+r}{2} \rfloor$, compare a_m with x. If $a_m = x$, we have our x. If $a_m < x$, set $\ell = m$, if $x < a_m$ set $r = m$.

(b) You are given an $n \times n$ matrix containing numbers $a_{i,j}$. Each row and each column of the matrix is sorted, i.e., $a_{i,j} \leq a_{i+1,j}$ and $a_{i,j} \leq a_{i,j+1}$. You can access any $a_{i,j}$ in constant time. Show how to check if the matrix contains a number x in just $\mathcal{O}(n)$ steps. For extra credit: show an asymptotically better solution or prove that one cannot beat linear complexity here.

Solution: Start with $i = 1$ and $j = 1$. While $i \leq n$ repeat the following: increase j by one as long as $a_{i,j} \leq x$, then if $a_{i,j} = x$ we have our x, otherwise increase i by one. The correctness follows from the observation that for each row we find the smallest j such that $a_{i,j} > x$, and these values of j cannot decrease as we increase i. The number of operations is $\mathcal{O}(n)$ because every step increases $i + j$, and the sum can be at most $2n$.

For extra credit, we prove the lower bound. More precisely, one can show that for some arrays any algorithm has to access at least n cells, otherwise we could fool it into answering NO while the answer is actually YES. The array are very simple: zeroes above the main diagonal, ones below the main diagonal, and undetermined values on the diagonal. For $n = 4$ the situation looks like this:

$$\begin{array}{cccc} 0 & 0 & 0 & ? \\ 0 & 0 & ? & 1 \\ 0 & ? & 1 & 1 \\ ? & 1 & 1 & 1 \end{array}$$

As long as each ? is replaced by something from $[0, 1]$, we get a proper row- and column-sorted array. Now we run the algorithm with $x = \frac{1}{2}$. Each time it accesses a diagonal entry, we put 0 there. At some the algorithm terminates. If all diagonal cells contain 0, the algorithm asked n questions, so it was slow. If some diagonal cell contain ? and the algorithm answered NO, we put $\frac{1}{2}$ in the corresponding cell, so the answer becomes incorrect. If the algorithm answered YES, we put 0 in the corresponding cell, so the answer becomes incorrect, too. Hence there can be no ? which we could replace, so the algorithm must have asked at least n questions.

(SPOJ) 3. Extra credit: Given two strings x and y, find the minimum number of characters to be removed from x in order to obtain a string x' that does not contain y as a substring.

Solution: The limits suggest that your complexity should $\mathcal{O}(|x||y|)$, so it's probably some kind of dynamic programming. First let's think how we could check if x' contains y as a substring. We have covered the Knuth-Morris-Pratt algorithm, which can be seen as a

deterministic state automaton. The states of the automaton are prefixes of the string we are looking for, so they correspond to all $y[1..i]$, where $i = 0, 1, \ldots, |y|$. Then we have a function $next(i, c)$ telling us what should be state after reading letter c if the current state is i. More precisely, $next(i, c)$ is the longest suffix of $y[1..i]c$ which is a prefix of y. All $next(i, c)$ can be precomputed using the method from the lecture. Now in the dynamic programming we compute for each prefix $x[1..j]$ of x and each i denoting the state of the automaton the smallest number of characters to be removed from the remaining part $x[j + 1..|x|]$ of x so that the automaton doesn't find any occurrence if it starts in state i and reads the non-removed characters of $x[j+1..|x|]$. If we denote by $T[i, j]$ the corresponding smallest number of characters, we get the following relation $T[i, j] = \min(1 + T[i+1, j], T[i+1, next(j, x[i])])$.