Exercise 7.1 Implementing a Queue with Stacks.

An ordinary stack supports the following two operations in time $\Theta(1)$:

- **Push($x$)**: puts an object $x$ onto the stack.
- **Pop**: outputs and removes the last object that was added to the stack.

Describe how a queue can be implemented using two stacks such that the following operations have amortized running time $\Theta(1)$:

- **Enqueue($x$)**: adds an object $x$ at the end of the queue.
- **Dequeue**: outputs and removes the object *at the front* of the queue.

Exercise 7.2 Numerical Puzzle.

You are given a sequence of $n$ digits from the set $\{0, \ldots, 9\}$ and a positive integer $\sigma$. If plus signs are inserted between some digits, and if the digits between two plus signs are interpreted as a decimal number, then the sum of the corresponding numbers can be computed. In general, different insertions of plus signs yield different sums.

Example: For the sequence $[6 \ 9 \ 2 \ 5 \ 0 \ 2 \ 1 \ 3]$ we can obtain the sums, for example, $69 + 2 + 5 + 0 + 21 + 3 = 100$ and $6 + 9 + 250 + 21 + 3 = 289$.

The exercise is to decide whether plus signs can be inserted into a given sequence such that the sum equals exactly $\sigma$.

a) Design an efficient algorithm that uses dynamic programming to solve this problem. You may assume that $\sigma$ is relatively small in relation to $n$. Provide also the running time of your solution. Is it polynomial in the size of the input?

*Hint:* Notice that you only need to decide whether $\sigma$ can be achieved or not.

b) How can we efficiently find all arrangements of plus signs that yield the desired sum?

Exercise 7.3 Programming Exercise: Longest Common Subsequence.

In this exercise we are going to implement the dynamic programming algorithm for solving the *longest common subsequence problem* that was presented in the lecture. We are given two strings of text $A = a_1 \cdots a_n$ and $B = b_1 \cdots b_m$, where $a_1, \ldots, a_n, b_1, \ldots, b_m$ are characters from an alphabet $\Sigma$, and we look for a longest string that is a (not necessarily contiguous) subsequence of both $A$ and $B$. For example, if $A = \text{“AGCAT”}$ and $B = \text{“GAC”}$, the longest common subsequences would be one of the following: “AC”, “GC”, “GA”.

*Please turn over.*
The algorithm uses a table $A[\cdot, \cdot]$ with $n + 1$ rows and $m + 1$ columns. For $0 \leq i \leq n$ and $0 \leq j \leq m$, the entry $A[i, j]$ represents the length of the longest common subsequence for the substrings of the original strings $a_1 \cdots a_i$ and $b_1 \cdots b_j$. For $i, j \geq 1$, it is computed as follows:

$$A[i, j] = \begin{cases} A[i - 1, j - 1] + 1 & \text{if } a_i = b_j \\ \max\{A[i - 1, j], A[i, j - 1]\} & \text{otherwise.} \end{cases}$$

(1)

Since $i = 0$ and $j = 0$ represent the empty substrings, $A[i, 0]$ and $A[0, j]$ are set to 0 for every $i$, $0 \leq i \leq n$ and for every $j$, $0 \leq j \leq m$.

After the table has been filled, the entry $A[n, m]$ contains the length $k$ of the longest common subsequence. The longest common subsequence itself is reconstructed from there using backtracing. If $a_n = b_m$, we set $a_n$ as the $k$-th character of the longest common subsequence, set $k \leftarrow k - 1$ and continue in the same fashion with the entry $A[n - 1, m - 1]$. If $a_n \neq b_m$, then we check whether $A[n, m] = A[n - 1, m]$ holds. If yes, then we continue with $A[n - 1, m]$, and with $A[n, m - 1]$ otherwise. We stop once we have read all the $k$ characters of the longest common subsequence. On the right, you can see the table $A[i, j]$ for the example strings $A = \text{“ZEBRA”}$ and $B = \text{“ZIEGE”}$.

**Input** The first line contains only the number $t$ of test instances. After that, we have exactly two lines per test instance. The first line contains the sequence $A$, and the second line contains the sequence $B$. The alphabet used is $\Sigma = \{A, B, \ldots, Z\}$.

**Output** For every test instance we output only one line. This line contains the length of the longest common subsequence followed by the longest common subsequence computed by the above algorithm.

**Example**

```
Input:

2
AGCAT
GAC
ROCK
ROLL

Output:

2 AC
2 RO
```