# Challenge III: Shearsort

### Description

For this challenge we look at `shearsort`, which is a parallelisable algorithm for sorting an $n \times n$ integer matrix in a snake-like order. With snake-like we mean that, after termination of `shearsort`, the rows of the given input matrix have been sorted in alternating direction.

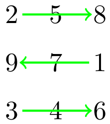
The next page shows a pseudo-code implementation of `shearsort`. It takes an integer matrix $M$ as input, which is assumed to be of size $n \times n$ (with $n$ a positive integer). Then the following two steps are repeated $\lceil \mathtt{log}_2(n) \rceil + 1$ times:

1. Sort all rows of $M$ in an alternating manner.
2. Sort all columns of $M$ in ascending order.

Below an example application of `shearsort` is given, on a $3 \times 3$ matrix:

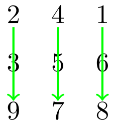
$$\begin{matrix} 5 & 8 & 2 \\ 9 & 1 & 7 \\ 3 & 6 & 4 \end{matrix}$$

Input matrix



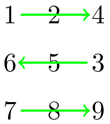
$$\begin{matrix} 2 & \rightarrow 5 \rightarrow & 8 \\ 9 & \leftarrow 7 \leftarrow & 1 \\ 3 & \rightarrow 4 \rightarrow & 6 \end{matrix}$$

Round 1: sorting rows



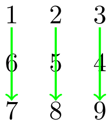
$$\begin{matrix} 2 & 4 & 1 \\ \downarrow & \downarrow & \downarrow \\ 3 & 5 & 6 \\ \downarrow & \downarrow & \downarrow \\ 9 & 7 & 8 \end{matrix}$$

Round 1: sorting columns



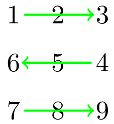
$$\begin{matrix} 1 & \rightarrow 2 \rightarrow & 4 \\ 6 & \leftarrow 5 \leftarrow & 3 \\ 7 & \rightarrow 8 \rightarrow & 9 \end{matrix}$$

Round 2: sorting rows



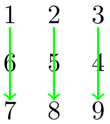
$$\begin{matrix} 1 & 2 & 3 \\ \downarrow & \downarrow & \downarrow \\ 6 & 5 & 4 \\ \downarrow & \downarrow & \downarrow \\ 7 & 8 & 9 \end{matrix}$$

Round 2: sorting columns



$$\begin{matrix} 1 & \rightarrow 2 \rightarrow & 3 \\ 6 & \leftarrow 5 \leftarrow & 4 \\ 7 & \rightarrow 8 \rightarrow & 9 \end{matrix}$$

Round 3: sorting rows



$$\begin{matrix} 1 & 2 & 3 \\ \downarrow & \downarrow & \downarrow \\ 6 & 5 & 4 \\ \downarrow & \downarrow & \downarrow \\ 7 & 8 & 9 \end{matrix}$$

Round 3: sorting columns

$$\begin{matrix} 1 & 2 & 3 \\ 6 & 5 & 4 \\ 7 & 8 & 9 \end{matrix}$$

Output matrix

The row and column sorts in every round can be performed in parallel, as they operate on disjoint memory. The implementations of `sort-row` and `sort-column` are left abstract, but could be chosen to be any sorting function.

Furthermore, an alternative implementation of `shearsort` is given, that uses a matrix `transpose` operation instead of `sort-column`. This version should be equivalent to `shearsort`, although less efficient. Feel free to perform the verification tasks using `alternative-shearsort` instead, if that is more convenient.

Moreover, in case your verifier does not support reasoning about concurrency, feel free to turn all parallel for-loops into sequential ones.

### Verification tasks

The verification tasks for `shearsort` are:

1. Verify that `shearsort` terminates, and is memory safe.
2. Verify that `shearsort` permutes the input matrix.
3. Verify that `shearsort` sorts the matrix in a snake-like manner.
4. Verify that (parallel) `shearsort` satisfies the same specification as sequential `shearsort`, in which all parallel for-loops are replaced by sequential ones.
5. Verify that `shearsort` and `alternative-shearsort` satisfy the same specification.
6. Extra: give implementations to `sort-row`, `sort-column` and `transpose`, and verify these as well.

```
// Sorts the row-th row of $M$ in ascending order if ascending is true,
// or in descending order if ascending is false.
void sort-row(int[][] $M$, int row, bool ascending) {
  ...

// Sorts the column-th column of $M$ in ascending order.
void sort-column(int[][] $M$, int column) {
  ...

// Sorts $M$ in snake-like order, assuming that $M$ is an $n \times n$ matrix.
void shearsort(int $n$, int[][] $M$) {
  repeat $\lceil \log_2(n) \rceil + 1$ times {
    for int tid $= 0 \ldots n$ do in parallel {
      sort-row($M$, tid, tid $\% 2 = 0$);
    }
    for int tid $= 0 \ldots n$ do in parallel {
      sort-column($M$, tid);
    }
  }

// An alternative version of shearsort, that only uses sort-row.
void alternative-shearsort(int $n$, int[][] $M$) {
  repeat $\lceil \log_2(n) \rceil + 1$ times {
    for int tid $= 0 \ldots n$ do in parallel {
      sort-row($M$, tid, tid $\% 2 = 0$);
    }
    transpose($M$);
    for int tid $= 0 \ldots n$ do in parallel {
      sort-row($M$, tid, true);
    }
    transpose($M$);
  }
```