Challenge I: Lexicographic Permutations

Description

This challenge considers an algorithm that takes any sequence A as input, and enumerates all possible permutations of A. Moreover, it enumerates these permutations in *sorted (lexicographic) order*. For example, when given a sequence `A B C` of characters as input, the algorithm enumerates and outputs the following six sequences: `A B C`, `A C B`, `B A C`, `B C A`, `C A B`, `C B A`, in that specific order.

Let us restrict ourselves to integer sequences for the purpose of this challenge. For example, given an integer sequence `3 1 2` as input, the algorithm enumerates and reports the sequences `1 2 3`, `1 3 2`, `2 1 3`, `2 3 1`, `3 1 2`, `3 2 1`, in that order.

The enumeration algorithm, named `permut`, is shown on the next page. At the heart of this algorithm lies a procedure named `next(`A`)` that takes an integer array A as input and modifies A to be the *next permutation* in the enumeration sequence. For example, if A were `2 3 1`, then `next(`A`)` would modify A to be `3 1 2`. The `next` procedure does not wrap, but instead returns a Boolean value that indicates whether a next permutation was found. For example, `next(2 3 1)` yields `true`, whereas `next(3 2 1)` yields `false` as it is the last permutation.

Verification tasks

The verification challenges related to `next` are:

1. Verify that `next` is memory safe.
2. Verify that `next` terminates for every input.
3. Verify that any changes on A performed by `next(`A`)` are permutations.
4. Verify that, if `next(`A`)` returns `false`, then A is left unmodified and is indeed the "last permutation" in the permutation sequence.
5. Verify that, if `next(`A`)` yields `true`, then A is modified to be the proper "next permutation" in the sequence.

The verification challenges related to the `permut` procedure are:

6. Verify that `permut` is memory safe.
7. Verify that `permut` terminates for every input.
8. Verify that any permutation reported by `permut` is unique.

9. Verify that `permut(`A`)` reports all permutations of A.

10. Verify that `permut` outputs all permututations in lexicographic order.

```
1  seq⟨int⟩ permut(int[] A) {
2     seq⟨int⟩ result := seq();
3
4     if (A = null) return result;
5
6     sort(A);
7
8     do { result := result ++ seq(to_seq(A)); }
9     while (next(A));
10
11    return result;
12 }
13
14 bool next(int[] A) {
15    int i := A.length − 1;
16    while (i > 0 ∧ A[i − 1] ≥ A[i]) {
17       i := i − 1;
18    }
19
20    if (i ≤ 0) return false;
21
22    int j := A.length − 1;
23    while (A[j] ≤ A[i − 1]) {
24       j := j − 1;
25    }
26
27    int temp := A[i − 1];
28    A[i − 1] := A[j];
29    A[j] := temp;
30
31    j := A.length − 1;
32    while (i < j) {
33       temp := A[i];
34       A[i] := A[j];
35       A[j] := temp;
36       i := i + 1;
37       j := j − 1;
38    }
39
40    return true;
41 }
```