Challenge 1: Pair Insertion Sort

Although it is an algorithm with $O(n^2)$ complexity, this sorting algorithm is used in modern library implementations. When dealing with smaller numbers of elements, insertion sorts performs better than, e.g., quicksort due to a lower overhead. It can be implemented more efficiently if the array traversal (and rearrangement) is not repeated for every element individually.

A Pair Insertion Sort in which two elements are handled at a time is used by Oracle's implementation of the Java Development Kit (JDK) for sorting primitive values. In the following code snippet a is the array to be sorted, and the integer variables `left` and `right` are valid indices into a that set the range to be sorted.

```
for (int k = left; ++left <= right; k = ++left) {
    int a1 = a[k], a2 = a[left];

    if (a1 < a2) {
        a2 = a1; a1 = a[left];
    }
    while (a1 < a[--k]) {
        a[k + 2] = a[k];
    }
    a[++k + 1] = a1;

    while (a2 < a[--k]) {
        a[k + 1] = a[k];
    }
    a[k + 1] = a2;
}
int last = a[right];

while (last < a[--right]) {
    a[right + 1] = a[right];
}
a[right + 1] = last;
```

(in `DualPivotQuicksort.java` line 245ff, used for `java.util.Arrays.sort(int[])`)

(This is an optimised version which uses the borders a[left] and a[right] as sentinels.)
While the problem is proposed here as a Java implementation, the challenge does not use specific language features and can be formulated in other languages easily.

A simplified variant of the algorithm in pseudo code for sorting an array A whose indices range from 0 to `length(A)-1` is the following:

```
i = 0                               i is running index (inc by 2 every iteration)
while i < length(A)-1
    x = A[i]                        let x and y hold the next to elements in A
    y = A[i+1]

    if x < y then                   ensure that x is not smaller than y
        swap x and y

    j = i - 1                       j is the index used to find the insertion point
    while j >= 0 and A[j] > x       find the insertion point for x
       A[j+2] = A[j]                shift existing content by 2
       j = j - 1
    end while
    A[j+2] = x                      store x at its insertion place
                                    A[j+1] is an available space now

    while j >= 0 and A[j] > y       find the insertion point for  y
        A[j+1] = A[j]               shift existing content by 1
        j = j - 1
    end while
    A[j+1] = y                      store y at its insertion place

    i = i+2
end while

if i = length(A)-1                  if  length(A) is odd, an extra
    y = A[i]                        single insertion is needed for
    j = i - 1                       the last element
    while j >= 0 and A[j] > y
        A[j+1] = A[j]
        j = j - 1
    end while
    A[j+1] = y
end if
```

VerifyThis – Competition, Uppsala 2017

Verification Tasks:

1. Specify and verify that the result of the pair insertion sort algorithm is a sorted array.
2. Specify and verify that the result of the pair insertion sort algorithm is a permutation of the input array.

Getting Started. To make the exercise more accessible, feel free to start with stripped down versions of the problem. A few possibilities for simplifications are:

1. Absence of unexpected runtime exceptions
2. Verify a single-step insertion sort algorithm in which every element is handled individually.
3. For permutations proofs, it may be simpler to not remember the values in temporary variables (x and y in the pseudocode), but to swap repeatedly.

Challenge. Try to get as close as possible to Oracle's implementation (outlined above) from the beginning.

Verification Bounds. In reality, pair insertion sort is used only for small problem instances: in JDK's case, if the array has less than 47 elements. If it helps your efforts, you may assume a suitable length restriction for the array.