Poor Insertion Sort

```
/** Sort b */
static void poorInsertionSort(int[] b) {
  // inv: 1 ≤ h ≤ b.length && sorted b[0..h-1]
  for (var h= 1; h < b.length; h++ )
     Put b[h] into its sorted position in b[0..h]
}
```

Method `poorInsertionSort`[1] to the right looks like insertion sort. That well-known method implements the repetend by iteratively comparing the value initially in `b[h]` with its predecessor and swapping if necessary. For example, if `b[0..h]` were initially (2 3 5 7 8 **4**), it would be changed to (2 3 5 7 **4** 8), then (2 3 5 **4** 7 8), and finally to (2 3 **4** 5 7 8). We have bolded the value being pushed down.

```
// pre: 1 ≤ h ≤ b.length && sorted b[0..h-1]
// Put b[h] into its sorted position in b[0..h]
for (var k= 0; k < h; k++)
      if (b[h] < b[k]) Swap b[h] and b[k]
// post: sorted b[0..h]
```

Instead `poorInsertionSort` implements the repetend as shown to the right, inserting `b[h]` immediately into its sorted position by swapping it into that position. Below, we give an example of array segment `b[0..h]` initially and after each iteration:

initial:	2 3 5 7 8 **4**
after iteration 0:	2 3 5 7 8 **4**
after iteration 1:	2 3 5 7 8 **4**
after iteration 2:	2 3 **4** 7 8 5
after iteration 3:	2 3 **4** 5 8 7
after iteration 4:	2 3 **4** 5 7 8

For some, this example may be enough to give them confidence in the algorithm. We prefer to give the invariant of the loop and discuss correctness in terms of it.

```
inner loop invariant:
P1: 0 ≤ k ≤ h
P2: sorted b[0..h-1]
P3: b[0..k-1] ≤ b[h]
```

The invariant for the inner loop is shown to the right. In discussing it, remember that the invariant must hold after executing both the repetend and the following increment `k++`. The assignment `k= 0;` truthifies P1 and P3 (since `b[0..k-1]` is empty), and P2 is in the precondition.

It is obvious that the inner loop, a for-loop, terminates. Upon termination, `k = h`. This fact together with P2 and P3 imply postcondition post: sorted `b[0..h]`.

It remains to show that each iteration maintains the invariant. P1 remains true since the iteration occurs only if `k < h`. We handle the two cases: `b[h] < b[k]` and `b[h] ≥ b[k]`.

Suppose `0 ≤ k < h` and `b[h] ≥ b[k]`.
(1) P2 remains true since `b` and `h` are not changed.
(2) We show that P3 remains true. P2 and `k < h` implies that `b[0..k]` is sorted. Add the fact that `b[h] ≥ b[k]` and we have `b[0..k] ≤ b[h]`. After the repetend, `k` is incremented, giving us `b[0..k-1] ≤ b[h]`, which is P3.

Suppose `0 ≤ k < h` and `b[h] < b[k]`: Element `b[k]` is replaced by `b[h]`.
(1) We show that P2 is maintained. By P3, after the replacement, `b[0..k]` is sorted. By P2, `b[k..h-1]` is sorted. Since `b[k]` is replaced by a smaller value, `b[k..h-1]` remains sorted. Since `b[0..k]` is sorted and `b[k..h-1]` is sorted, `b[0..h-1]` is sorted, which indicates that P2 is true after the replacement.
(2) We show that P3 is maintained. Since P2 is maintained and `k < h` after the swap (but before incrementing `k`), we have `b[0..k] ≤ b[k]` after the swap. This, together with `b[h] ≥ b[k]`, yields `b[0..k] ≤ b[h]` after the swap. Thus, after `k` is incremented, `b[0..k-1] ≤ b[h]`, which is P3.

Don't use poorInsertionSort

Insertion sort takes linear time if `b` is already in sorted (non-descending) order. PoorInsertionSort always takes time $O(\texttt{b.length}^2)$, because the repetend always compares `b[h]` against all elements in `b[0..h-1]`. PoorInsertionSort is also worse than Selection sort, which takes $O(\texttt{b.length}^2)$ time but makes at most `b.length-1` swaps. PoorInsertionSort may make many swaps on each iteration of the main loop. You might ask where this algorithm came from? How did someone come up with it? We answer that question next.

[1] If array `b` has length 0, the main loop terminates immediately, with the proper result: an empty array is sorted. Therefore, throughout, we assume that `0 < b.length`. Also, an unmentioned invariant is that `b` is a permutation of its initial value, since it is changed only by swapping two of its elements.

Poor Insertion Sort

SimpleSort, by Stanley Fung

In 2021, Stanley P.Y. Fung published the sorting algorithm shown to the right (arxiv.org/pdf/2110.01111.pdf). He found it sort of by accident. It could be the simplest and shortest sorting algorithm, taking only three lines, with both loops sequencing through all array elements, but that is its only advantage. Its worst case and expected case times are in O(b.length^2). It's not stable. It unnecessarily compares all pairs of positions twice. It is not obvious that it works! Let's work on proving it.

```
static void simpleSort(int[] b) {
  for (var h= 0; h < b.length; h++ )
    for (var k= 0; k < b.length; k++ )
      if (b[h] < b[k]) Swap b[h] and b[k]
}
```

Hey, when `h = 0`, during the first iteration of the outer loop, the inner loop simply looks at each value in `b` and, if it is larger than `b[0]`, swaps that value with `b[0]`. Thus, the first iteration truthifies `b[0] = ↑b[0..]` (we use ↑ for the maximum operator applied to its following operand). We won't give a formal proof of this, for this is something every respectable computer scientist can do. Thus, after the first iteration (including `h++`), `b[h-1] = ↑b[0..]`.

In fact, we can now use Q0, Q1, and Q2, to the right, as the invariant of the outer loop. We know already that it is true initially, when `h = 0`, and after the first iteration. And upon termination, we have `h = b.length`; that together with Q1 implies that `b[0..]` is sorted.

simpleSort outer loop invariant:
Q0: $0 \le h \le$ b.length
Q1: sorted b[0..h-1]
Q2: $0 < h \Rightarrow$ b[h-1] = ↑b[0..]

Now compare the inner loop of `simpleSort`, above, with the inner loop of `poorInsertionSort`, to the right. One has the stopping condition `k = b.length`, the other `k = h`. This tells us that iterations of the inner loop of simpleSort with `k` in the range `1 ≤ k < h` satisfy the inner-loop invariants P1, P2, and P3 of `poorInsertionSort`, and we use for the invariant of `simpleSort` P1, P2, P3, Q2, and Q3 shown to the right below.

```
static void poorInsertionSort(int[] b) {
  // inv: 1 ≤ h ≤ bb.length && sorted b[0..h-1]
  for (var h= 1; h < b.length; h++ )
    // Put b[k] into its sorted position in b[0..h]
    for (var k= 0; k < h; k++ )
      if (b[h] < b[k]) Swap b[h] and b[k];
}
```

When `k = h-1`, either (1) `b[h] = b[h-1]` and no swap is done or (2) `b[h] < b[h-1]` and the maximum value `b[h-1]` is swapped into `b[h]`. Therefore, after the iteration, `b[h] = ↑b[0..]`.

Finally, consider the iterations with `h ≤ k < b.length`. Since `b[h] = ↑b[0..]`, no swaps are performed, and these iterations have no effect on the truth of the invariant.

simpleSort inner loop invariant:
P1: $0 \le k \le$ b.length
P2: sorted b[0..h-1]
P3: b[0..k-1] ≤ b[h]
Q2: $0 < k < h \Rightarrow$ b[h-1] = ↑b[0..]
Q3: $0 < h \le k \Rightarrow$ b[h] = ↑b[0..]

Consequently, when `k = b.length` and the inner loop terminates, P2 together with Q3 imply the postcondition, `sorted b[0..h]`. This ends the proof.

Discussion

In his paper, *Is this the simplest (and most surprising) sorting algorithm ever?* (arxiv.org/pdf/2110.01111.pdf), Stanley Fung presents this material in a different order. He starts off with simpleSort, as shown to the right, calling it *ICan'tBelieveItCanSort*. He had stumbled onto it in some fashion, and proved that it actually sorted array `b`. He then began dissecting it and noticed that it had a version of insertion sort in it.

```
static void simpleSort(int[] b) {
  for (var h= 0; h < b.length; h++ )
    for (var k= 0; k < b.length; k++ )
      if (b[h] < b[k]) Swap b[h] and b[k]
}
```

We have changed the narrative around, first showing the new version of insertion sort and then using its proof in the development of the proof of simpleSort. Along the way, we made more explicit the invariants for the inner loop.