In 2021, Stanley P.Y. Fung published the sorting algorithm shown to the right ([arxiv.org/pdf/2110.01111.pdf](https://arxiv.org/pdf/2110.01111.pdf)). He found it sort of by accident, It is, perhaps, the simplest and shortest sorting algorithm, taking only three lines, but that is its only advantage. Its worst case and expected case times are in O(b.length2). It's not stable. It unnecessarily compares all pairs of positions twice. It is not obvious that it works. Even its proof of correctness, given below, is not particularly easy. But it does show an alternative to insertion sort!

/\*\* Sort b \*/

public static void sillySort(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];

}

# The invariant of the outer loop

The invariant of the outer loop appears to the right, where prefix operator sorted means that its following operand is in non-decreasing order and prefix operator ↑ stands for the maximum value of its operand.

P1: 0 ≤ h ≤ b.length  
P2: sorted b[0..h-1]  
P3: 1 ≤ h ⇒ b[h-1] = ↑ b[0..]

P1 is true initially and obviously remains true and won't be discussed further. Also, it is clear that the outer loop terminates.

The initialization h= 0; truthifies P1, P2, and P3. Thus, the invariant is initially true.

The loop terminates with h = b.length, and in that case, P2 shows that b[0..] is sorted. Thus, upon termination, the required result holds.

It remains to prove that the repetend of the outer loop (i.e. the inner loop) maintains invariants P1, P2, and P3. To the right, we give precondition pre of the repetend. What is the postcondition? Since the assignment h= h+1; is executed after the repetend, postcondition post is loop invariant P1 ∧ P1 ∧ P3 with each occurrence of h replaced by h+1 (and the result simplified). The rest of our proof concerns proving that {pre} repetend {post} is true. It's not simple!

pre: 0 ≤ h < b.length ∧ P1 ∧ P2 ∧ P3

repetend

post: 0 ≤ h < b.length ∧   
 sorted b[0..h] ∧

0 ≤ h ⇒ b[h] = ↑ b[0..]

# Proof of {pre} repetend {pos }

Case h = 0. The first iteration of the outer loop has h = 0. Thus, think of the repetend, shown to the right, with all occurrences of h replaced by 0. With h = 0, it is easily seen that this loop swaps the largest value into b[0], and this truthifies post: h is 0, b[0..h] is sorted since it contains one value, and b[h] = ↑ b[0..].

**Repetend:**

for (var k= 0; k < b.length; k++ )

if (b[h] < b[k]) Swap b[h] and b[k]

# Case h > 0. Purpose of the inner loop

It will help to understand the inner loop in the case h > 0 with the example shown to the right with h = 5. The example shows array b at the start and after each iteration. We put the maximum value in boldface. Note that b[0..h-1] is sorted and b[h-1] = ↑ b[0..].

0 h

start: 1 3 5 6 **8** 4 7 2   
iteration 0: 1 3 5 6 **8** 4 7 2

iteration 1: 1 3 5 6 **8** 4 7 2

iteration 2: 1 3 4 6 **8** 5 7 2

iteration 3: 1 3 4 5 **8** 6 7 2

iteration 4: 1 3 4 5 6 **8** 7 2

*The purpose of the inner loop* is to insert b[h] into its sorted position in b[0..h]. Unlike insertion sort, which pushes b[h]down step by step, the inner loop puts it in its final position immediately by swapping with the value in that position.

Iteration 0 changes nothing because 4 ≥ 1.  
 Iteration 1 changes nothing because 4 ≥ 3.  
 Iteration 2 swaps 4 and 5 because 4 < 5.  
 Iteration 3 swaps 6 and 5 because 5 < 6.  
 Iteration 4 swaps 8 and 6 because 6 < 8. b[h] is now the maximum array value.  
 The rest of the iterations change nothing because b[h] is the maximum array value.

# Inner loop invariant in the case h > 0

1≤k≤h-1 => b[0..k] ≤ b[h] ≤ b[k+1..h-1]

h≤k => b[h] = ↑ b[0..]