Sorting: Bubble Sort and Selection Sort

Solutions

AddisCoders 2026 — Week 3, Day 12, Lab B

August 11, 2026

1 Bubble Sort

Starting list: 5 1 4 2 8

(a) Iteration 1

Each row compares one neighbouring pair and swaps only when the left value is bigger.

Compare	Swap?	List after this comparison
5 and 1	yes	1 5 4 2 8
5 and 4	**yes**	**1 4 5 2 8**
5 and 2	**yes**	**1 4 2 5 8**
5 and 8	**no**	**1 4 2 5 8**

Notice that after one full walk the largest value, 8, has bubbled all the way to the last position. This is always true: *iteration k places the k-th largest value in its final spot.*

(b) The list at the end of each remaining iteration

- **End of Iteration 2:** **1 2 4 5 8**
  (Only one swap happens here: 4 and 2 are out of order.)
- **End of Iteration 3:** **1 2 4 5 8**
  (No swaps — the list is already sorted.)
- **End of Iteration 4:** **1 2 4 5 8**
  (Again no swaps; the algorithm simply walks the list once more.)

(c) Number of iterations for a list of n elements

Number of iterations: $n-1$

For $n = 5$ this gives $5 - 1 = 4$ iterations, which matches what we did above.

Why $n-1$? Each iteration locks one more value into its final position at the right-hand end. Once $n-1$ values are locked in, the single remaining value must already be in the correct place, so no further walk is needed.

Why did Iteration 4 still run? This version of bubble sort is *not adaptive*: it always performs the same fixed number of walks, no matter what the data looks like. It has no way of noticing that the list became sorted early. A common improvement is to keep a flag that records whether any swap happened during a walk, and stop as soon as a walk finishes with no swaps.

2 Selection Sort

Starting list: 7 3 9 1 5

(a) The table

Shaded boxes are the part of the list that is already sorted and must not be touched again.

Step	Unsorted part	Smallest value in it	List after the swap
1	7 3 9 1 5	1	1 3 9 7 5
2	1 3 9 7 5	**3**	**1** **3** **9** **7** **5**
3	1 3 9 7 5	**5**	**1** **3** **5** **7** **9**
4	1 3 5 7 9	**7**	**1** **3** **5** **7** **9**

After Step 4 only one element is left, so it is automatically in the right place and the list is fully sorted: 1 3 5 7 9

(b) What happens when the smallest value is already in place?

The algorithm swaps the element with itself, which changes nothing — the list stays exactly the same.

(c) Completing the code

```
def selection_sort(lst):
    for i in range(len(lst)):
        smallest = i
        for j in range(i + 1, len(lst)):
            if lst[j] < lst[smallest]:
                smallest = j
        lst[i], lst[smallest] = lst[smallest], lst[i]
    return lst
```

Blank 1: `i + 1` **Blank 2:** `j`

Hint: why does the inner loop not start at 0?
Everything before position `i` has already been sorted and put in its final position by earlier steps, so there is nothing to find there — and looking at it could even cause a wrong swap. The unsorted part runs from `i` to the end of the list, and since `smallest` is already set to `i` before the loop

begins, position `i` has effectively been checked. The inner loop therefore only needs to examine `i + 1` onwards.

Blank 2 note: we store the *index* `j`, not the value `lst[j]`. The final line uses `smallest` as a position for indexing, so it must be an index.

Summary

	Bubble Sort	**Selection Sort**
Main move	swap neighbours	find the minimum, then swap it into place
Passes over data	$n - 1$	n (the last one does nothing)
Swaps	many (one per out-of-order pair)	at most one per step
Sorted part	grows at the right end	grows at the left end