Consider the following algorithm:

```
def order(L):
    """ (list of numbers) -> None
        Order L from smallest to largest. L is changed in-place. """
    i = 1
    while i < len(L):
        j = i
        while j > 0 and L[j] < L[j-1]:
            L[j], L[j-1] = L[j-1], L[j]  # swap L[j] and L[j-1]
            j = j - 1
        i = i + 1
```

1. Compute the *exact* number of "swaps" (executing the line that says `swap`) performed by the algorithm in the worst-case, on any list L of length n.

   The `def` line can be ignored: it is part of the syntax to define a function, but not something that actually gets executed every time we call the function. So we count only the steps in the body of the function.

   The outer loop iterates over `i = 1,2,3,...,n-1`.

   For each value of `i`, the inner loop iterates over `j = i,i-1,...,2,1`, *as long as* `L[j] < L[j-1]`. In the worst-case (when `L` is initially sorted in reverse order), this happens for *every* value of `j`.

   For each value of `j`, the algorithm swaps once: 1 time when `i = 1` (for `j = 1`), 2 times when `i = 2` (for `j = 2` and `j = 1`), ..., $n-1$ times when `i = n-1` (for `j = n-1` and ... and `j = 1`).

   So in total, the algorithm performs exactly $1 + 2 + \cdots + n - 1 = n(n-1)/2 = n^2/2 - n/2$ swaps, in the worst-case.

2. Compute the *exact* number of "steps" (basic operations) performed by the algorithm in the worst-case, on any list L of length n.

   First, let's indicate the number of "steps" taken by every line in the algorithm. Remember that it doesn't actually matter exactly how we count our steps, as long as every portion of the code that takes constant time (meaning, time independent of the input size) is counted as a constant number of steps. Here, we count one step for every *operator* (=, -, +, <, >, []) and every *keyword* (`while`, `and`, `len`).

```
def order(L):
    """ (list of numbers) -> None
        Order L from smallest to largest. L is changed in-place. """
    i = 1                                  # 1 step
    while i < len(L):                      # 3 steps
        j = i                              # 1 step
        while j > 0 and L[j] < L[j-1]:     # 7 steps
            L[j], L[j-1] = L[j-1], L[j]    # 7 steps
            j = j - 1                      # 2 steps
        i = i + 1                          # 2 steps
```

   As before, we count only the lines in the body. The outer loop iterates over `i = 1,2,3,...,n-1`.

   For each value of `i`, the inner loop iterates over `j = i,i-1,...,2,1`, in the worst-case (as argued in the first question).

   For each value of `j`, the algorithm performs exactly 16 steps. So over all values of `j`, a total of $16i$ steps.

   In addition, for each value of `i`, there are steps performed outside of the inner loop: 6 steps for the lines outside the inner loop, and an additional 7 steps to evaluate the last inner loop condition — when the condition becomes False. So each iteration of the outer loop performs $16i + 13$ steps.

   Together with the first line, and the extra 3 steps for the last outer loop condition, the number of steps performed by the algorithm is exactly:

$$\begin{aligned}\left(\sum_{i=1}^{n-1}(16i+13)\right) + 4 &= 16\left(\sum_{i=1}^{n-1} i\right) + 13\left(\sum_{i=1}^{n-1} 1\right) + 4 \\ &= 16n(n-1)/2 + 13(n-1) + 4 \\ &= 8n^2 + 5n - 9\end{aligned}$$

   NB: As stated earlier, our choice of assigning each operation and keyword one "step" was arbitrary — we could just as easily assign each line or statement that was independent of n a step. In other words, in the accounting above, making each line worth 1 step (rather than some being 7 or 2) would make the accounting simpler, and not change the quality of the outcome: $\mathcal{O}(n^2)$.