

# Exercise Sheet 2 – Solutions

## Advanced Programming and Algorithms

---

### Problem 1 – Running Time (Selection Sort)

a) Comparison to insertion sort:

**Selection sort** repeatedly selects the smallest remaining element in the unsorted suffix and swaps it into position at index j.

**Insertion sort** builds the sorted prefix incrementally by inserting the next item into its correct position within the already-sorted part.

b) Best-case and worst-case inputs:

**Best case:** An already sorted array, e.g., [1, 2, 3, 4, 5].

**Worst case:** A reverse-sorted array, e.g., [5, 4, 3, 2, 1].

Selection sort does not adapt to input order; best and worst cases have the same number of comparisons.

c) Asymptotic worst-case running time analysis:

analyze the lines of the provided pseudocode.

- Line 1 (outer for  $j = 0$  to  $n-1$ ): executes  $n$  iterations.
- Lines 3–5 (inner loop and min update): for fixed  $j$ , the inner loop runs  $n-j-1$  times; in total  $\sum_{j=0}^{n-1} (n-j-1) = n(n-1)/2$  comparisons and up to as many min-updates.
- Lines 6–8 (swap using a key): executed once per outer iteration  $\rightarrow 3$  assignments per  $j$   $\rightarrow 3n$  in total.

Hence,  $T(n) = n(n-1)/2 + 3n + O(n) = (1/2)n^2 + O(n)$ .

Choose  $f(n) = n^2$ . Then for all  $n \geq 1$ ,  $T(n) \leq 10 \cdot n^2$ , thus  $T(n) \in O(n^2)$ .

d) Average running time:

*Selection sort performs the same number of comparisons for any input order, so the average running time is  $O(n^2)$ .*

## Problem 2 – Refactoring (Python Implementation)

Reference implementation with readability and reusability improvements:

```
def selection_sort(arr):
    """In-place selection sort (ascending).

    Args:
        arr (list): Mutable sequence of comparable items.

    Returns:
        list: The same list, sorted in place.
    """
    n = len(arr)
    for j in range(n):
        min_idx = j
        for i in range(j + 1, n):
            if arr[i] < arr[min_idx]:
                min_idx = i
        # single swap at the end of pass j
        arr[j], arr[min_idx] = arr[min_idx], arr[j]
    return arr
```

Readability & reusability considerations:

- Consistency: spacing, indentation, and docstring.
- Expressivity: descriptive names (min\_idx vs. min).
- Function extendability: single, clear purpose; short body; easy to test.
- Avoiding redundancy: one swap per outer iteration; no extra data structures.

*These changes do not alter asymptotic complexity: best, worst, and average remain  $O(n^2)$ .*

## Problem 3 – Space Complexity

a) Formal notion of space complexity:

The space complexity  $S(n)$  of an algorithm is the function mapping input size  $n$  to the maximum additional memory used during execution, beyond the input itself (auxiliary space plus constant overhead).

b) Space usage of selection sort:

- In-place algorithm: operates directly on the input array.
- Uses a constant number of scalar variables (indices, key for swap).
- No auxiliary arrays or recursion stacks.

*Therefore, auxiliary space is  $O(1)$ ; total space including the input is  $O(n)$ .*

## Problem 4 – Correctness (Preview)

Pseudocode:

```
do_something(a, b):
    x ← a
    for i ← 1 to b do
        x ← x + a + (i - 1)
        x ← x + a + i
    return x
```

a) Computation problem solved:

The algorithm returns  $x = a + \sum_{i=1}^b (a + (i-1) + a + i) = a + \sum_{i=1}^b (2a + 2i - 1)$ .

Closed form:  $x = a + 2ab + b(b+1) - b = 2ab + b^2 + a$ . (Since  $\sum_{i=1}^b i = b(b+1)/2$ .)

b) Loop invariant:

At the beginning of iteration  $i$  ( $1 \leq i \leq b+1$ ),  $x = a + \sum_{k=1}^{i-1} (2a + 2k - 1)$ .

c) Proof of the invariant:

- Initialization: Before the loop ( $i = 1$ ),  $x = a$ , which matches the sum over an empty set.
- Maintenance: If the invariant holds for iteration  $i$ , the two updates add  $(2a + 2i - 1)$ , yielding the stated form for  $i+1$ .
- Termination: After  $b$  iterations ( $i = b + 1$ ), we obtain  $x = a + \sum_{k=1}^b (2a + 2k - 1)$ .

d) Correctness:

By the invariant and the closed form, on termination the algorithm returns  $x = 2ab + b^2 + a$ , thus solving the described computation problem.