For each of the following problems, consider the algorithm below.

**Algorithm 1: SS(A)**

<table>
<thead>
<tr>
<th>Input: Array $A$ of orderable elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Let $i = 0$</td>
</tr>
<tr>
<td><strong>while</strong> $i &lt;</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>$i = i + 1$</td>
</tr>
<tr>
<td><em>return</em></td>
</tr>
</tbody>
</table>

**Problem 1.**

a) What does Algorithm 1 do?

b) Describe the worst-case input to Algorithm 1.

c) What is the worst-case runtime of Algorithm 1? Why?

d) Describe the best-case input to Algorithm 1.

e) What is the best-case runtime of Algorithm 1? Why?

f) What is the average-case runtime of Algorithm 1? Why?

**Solution 1.**

a) It sorts $A$.

b) All inputs are equally bad for the algorithm.

c) The worst-case runtime is $O(n^2)$ since it must scan the entire remaining unsorted array $n$ times to find the minimum remaining element to sort next. This results in the runtime

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2},$$

which is $O(n^2)$. 
d) All inputs are equally good for the algorithm.

e) Since the algorithm behaves identically on all inputs (modulo which elements get
swapped), the best-case runtime is the same as the worst-case runtime.

f) The average-case runtime is \( O(n^2) \) because the best and the worst-case runtimes
are identical.

**Problem 2.** When tackling nested loops in a proof of correctness, it is almost always best
to state your loop invariants from the outside in. It is often also useful to trace through
the algorithm with an example input to see what is going on.

a) What do you know about the first \( i \) elements of \( A \) at the start of an iteration of the
outer while loop of Algorithm 1?

b) What do you know about the remaining \(|A| - i\) elements of \( A \)?

c) Give a useful loop invariant for the outer while loop.

**Solution 2.**

a) The first \( i \) elements are sorted.

b) The remaining \(|A| - i\) elements are all at least as big as the first \( i \) elements.

c) Define the loop invariant

\[
LIO(A, i) := \forall 0 \leq k < i - 1, A[k] \leq A[k + 1] \text{ and } \forall i \leq k < |A|, A[i - 1] \leq A[k].
\]

**Problem 3.**

a) What do you know about \( m \) at the start of an iteration of the inner while loop of
Algorithm 1?

b) Does the inner while loop modify the array?

c) Give a useful loop invariant for the inner while loop.

**Solution 3.**

a) It is the index of the minimum element between indices \( i \) and \( j - 1 \) (both inclusive).

b) Define the loop invariant

\[
LII(A, i, j, m) := LIO(A, i) \text{ and } m = \arg \min_{i \leq k < j} (A[k]).
\]

The assumption that the outer loop invariant is true is technically necessary in order
to prove the outer loop invariant is true. It is, in essence, an assertion that the inner
while loop hasn’t scrambled the array. However, it may be omitted by observing
that the inner while loop clearly does not modify the array, though this is a little
handwave-y.

**Problem 4.**
a) Prove that the outer loop invariant you gave for Algorithm 1 is true before the first iteration of the outer while loop.

b) Assuming that your outer loop invariant is true, prove that the inner loop invariant you gave for Algorithm 1 is true before the first iteration of the inner while loop.

c) Prove that your inner loop invariant is a loop invariant.

d) Prove that your outer loop invariant is a loop invariant using your inner loop invariant.

e) Prove that Algorithm 1 is correct using your outer loop invariant.

Solution 4.

a) Before the first iteration of the outer while loop, \( LIO(A, 0) \) is vacuously true. There are no values of \( k \) satisfying \( 0 \leq k < -1 \), and \( A[-1] \) does not exist.

b) Assume \( LIO(A, i) \). Then before the first iteration of the inner while loop, we have \( m = i \) and \( j = i + 1 \). Thus \( LII(A, i, i + 1, i) \) is true since \( LIO(A, i) \) holds and \( m \) is assigned to be the only valid choice of \( k \) satisfying \( i \leq k < i + 1 \).

c) Assume \( LII(A, i, j, m) \). Over the course of the iteration, both \( A \) and \( i \) do not change, so \( LIO(A, i) \) remains true. Then we have

\[
m' = \arg \min(A[m], A[j]), \quad j' = j + 1.
\]

So we get that

\[
m' = \arg \min \left( \arg \min_{i < k < j} (A[k], A[j]) \right) = \arg \min_{i < k < j + 1} (A[k]) = \arg \min_{i < k < j'} (A[k]).
\]

Thus \( LII(A, i, j', m') \) holds.

d) The inner while loop breaks (and it clearly must) when \( j = |A| \). \( LII(A, i, |A|, m) \) holds at this point, so

\[
m = \arg \min_{i < k < |A|} (A[k]).
\]

We then swap \( A[i] \) and \( A[m] \) but otherwise leave \( A \) unchanged. We also assign \( i' = i + 1 \).

This guarantees that \( A'[i] = A'[i' - 1] \leq A'[k] \) for every \( i' \leq k < |A| \) since \( LIO(A, i) \) held and we assigned \( A'[i' - 1] \) to be the smallest element of the second half of the array. Moreover, we also know that \( A'[i' - 1] \) is at least as large as \( A[i - 1] \), so we have \( \forall 0 \leq k < i' - 1, A[k] \leq A[k + 1] \).

Thus \( LIO(A', i') \) holds.

e) The outer while loop breaks (and it clearly must) when \( i = |A| \). At this point, \( LIO(A, |A|) \) must hold. The loop invariant guarantees that the first \( |A| \) elements of \( A \) are in ascending order, so it follows that \( A \) is sorted. As no further modifications are made to \( A \) after the outer while loop breaks, \( SS(A) \) correctly sorts \( A \).