CSE 548 / AMS 542: Analysis of Algorithms

Prerequisites Review 1
(Insertion Sort and Selection Sort)

Rezaul Chowdhury
Department of Computer Science
SUNY Stony Brook
Fall 2023

Insertion Sort

Input array

	1	2	3	4	5	6	7	8	9	10
A	7	4	1	15	9	5	12	3	18	11

Let **State *j*** be a state of the array A in which all numbers that were originally in $A[1..j]$ are placed in sorted order (i.e., nondecreasing order of value) in $A[1..j]$.

State 1

Input array

	1	2	3	4	5	6	7	8	9	10
A	7	4	1	15	9	5	12	3	18	11

sorted (1–5) unchanged (6–10)

State 5

Suppose somehow we have this:

sorted (1–5) unchanged (6–10)

	1	2	3	4	5	6	7	8	9	10
A	1	4	7	9	15	5	12	3	18	11

State 6

Now from state 5 we want to reach this:

sorted (1–6) unchanged (7–10)

	1	2	3	4	5	6	7	8	9	10
A	1	4	5	7	9	15	12	3	18	11

Insertion Sort

State 5

Suppose somehow we have this:

sorted (1–5) | unchanged (6–10)

	1	2	3	4	5	6	7	8	9	10
A	1	4	7	9	15	5	12	3	18	11

To reach state 6 put $A[6]$ in sorted order among the sorted numbers in $A[1..5]$.

State 6

Now from state 5 we want reach this:

sorted (1–6) | unchanged (7–10)

	1	2	3	4	5	6	7	8	9	10
A	1	4	5	7	9	15	12	3	18	11

Insertion Sort

State 5

Suppose somehow we have this:

	1	2	3	4	5	6	7	8	9	10
A	1	4	7	9	15	5	12	3	18	11

Compare $A[6]$ with $A[5]$.

Since $A[5] > A[6]$, swap.

	1	2	3	4	5	6	7	8	9	10
A	1	4	7	9	5	15	12	3	18	11

Compare $A[5]$ with $A[4]$.

Since $A[4] > A[5]$, swap.

	1	2	3	4	5	6	7	8	9	10
A	1	4	7	5	9	15	12	3	18	11

Compare $A[4]$ with $A[3]$.

Since $A[3] > A[4]$, swap.

	1	2	3	4	5	6	7	8	9	10
A	1	4	5	7	9	15	12	3	18	11

Compare $A[3]$ with $A[2]$.

Since $A[2] \leq A[3]$, stop.

State 6

	1	2	3	4	5	6	7	8	9	10
A	1	4	5	7	9	15	12	3	18	11

Insertion Sort

Input array **A**

1	2	3	4	5	6	7	8	9	10
7	4	1	15	9	5	12	3	18	11

State 1

$A[1]$ is trivially sorted **A**

1	2	3	4	5	6	7	8	9	10
7	4	1	15	9	5	12	3	18	11

↓ Insert $A[2]$ in sorted order into $A[1]$

State 2

$A[1..2]$ is now sorted **A**

1	2	3	4	5	6	7	8	9	10
4	7	1	15	9	5	12	3	18	11

↓ Insert $A[3]$ in sorted order into $A[1..2]$

State 3

$A[1..3]$ is now sorted **A**

1	2	3	4	5	6	7	8	9	10
1	4	7	15	9	5	12	3	18	11

↓ Insert $A[4]$ in sorted order into $A[1..3]$

State 4

$A[1..4]$ is now sorted **A**

1	2	3	4	5	6	7	8	9	10
1	4	7	15	9	5	12	3	18	11

↓ Insert $A[5]$ in sorted order into $A[1..4]$

State 5

$A[1..5]$ is now sorted **A**

1	2	3	4	5	6	7	8	9	10
1	4	7	9	15	5	12	3	18	11

Insertion Sort

State 5

$A[1..5]$ is now sorted

	1	2	3	4	5	6	7	8	9	10
A	1	4	7	9	15	5	12	3	18	11

Insert $A[6]$ in sorted order into $A[1..5]$

State 6

$A[1..6]$ is now sorted

	1	2	3	4	5	6	7	8	9	10
A	1	4	5	7	9	15	12	3	18	11

Insert $A[7]$ in sorted order into $A[1..6]$

State 7

$A[1..7]$ is now sorted

	1	2	3	4	5	6	7	8	9	10
A	1	4	5	7	9	12	15	3	18	11

Insert $A[8]$ in sorted order into $A[1..7]$

State 8

$A[1..8]$ is now sorted

	1	2	3	4	5	6	7	8	9	10
A	1	3	4	5	7	9	12	15	18	11

Insert $A[9]$ in sorted order into $A[1..8]$

State 9

$A[1..9]$ is now sorted

	1	2	3	4	5	6	7	8	9	10
A	1	3	4	5	7	9	12	15	18	11

Insert $A[10]$ in sorted order into $A[1..9]$

State 10

$A[1..10]$ is now sorted

	1	2	3	4	5	6	7	8	9	10
A	1	3	4	5	7	9	11	12	15	18

Insertion Sort

Input: An array $A[\,1:\ n\,]$ of n numbers.

Output: Elements of $A[\,1:\ n\,]$ rearranged in non-decreasing order of value.

INSERTION-SORT (A)

```
for j = 2 to A.length
    // insert A[j] into the sorted sequence A[1..j − 1]
    i = j − 1
    while i > 0 and A[i] > A[i + 1]
        A[i + 1] ↔ A[i]          // swap A[i] and A[i + 1]
        i = i − 1
```

Worst Case Runtime of Insertion Sort (Upper Bound)

INSERTION-SORT (A)	cost	times
for $j = 2$ **to** $A.length$	c_1	n
// insert $A[j]$ into the sorted sequence $A[1..j-1]$	0	$n-1$
$i = j - 1$	c_3	$n-1$
while $i > 0$ **and** $A[i] > A[i+1]$	c_4	$\sum_{2 \le j \le n} j$
$A[i+1] \leftrightarrow A[i]$ // swap $A[i]$ and $A[i+1]$	c_5	$\sum_{2 \le j \le n} (j-1)$
$i = i - 1$	c_6	$\sum_{2 \le j \le n} (j-1)$

Running time, $T(n) \le c_1 n + c_3(n-1)$

$$+c_4 \sum_{j=2}^{n} j + c_5 \sum_{j=2}^{n} (j-1) + c_6 \sum_{j=2}^{n} (j-1)$$

$$= 0.5(c_4 + c_5 + c_6)n^2 + 0.5(2c_1 + 2c_3 + c_4 - c_5 - c_6)n - (c_3 + c_4)$$

$$\Rightarrow T(n) = O\left(n^2\right)$$

Best Case Runtime of Insertion Sort (Lower Bound)

Insertion-Sort (A)	cost	times
for $j = 2$ **to** $A.length$	c_1	n
// insert $A[j]$ into the sorted sequence $A[1..j-1]$	0	$n-1$
$i = j - 1$	c_3	$n-1$
while $i > 0$ **and** $A[i] > A[i+1]$	c_4	$n-1$
$A[i+1] \leftrightarrow A[i]$ // swap $A[i]$ and $A[i+1]$	c_5	0
$i = i - 1$	c_6	0

Running time, $T(n) \geq c_1 n + c_3(n-1) + c_4(n-1)$

$$= (c_1 + c_3 + c_4)n - (c_3 + c_4)$$

$$\Rightarrow T(n) = \Omega(n)$$

Insertion Sort
(Slightly Optimized but Same Asymptotic Bounds)

Input: An array $A[\,1 : n\,]$ of n numbers.

Output: Elements of $A[\,1 : n\,]$ rearranged in non-decreasing order of value.

INSERTION-SORT (A)

for $j = 2$ **to** $A.length$
  $key = A[j]$
  // insert $A[j]$ into the sorted sequence $A[1..j-1]$
  $i = j - 1$
  while $i > 0$ *and* $A[i] > key$
    $A[i+1] = A[i]$
    $i = i - 1$
  $A[i+1] = key$

Selection Sort

Input array

	1	2	3	4	5	6	7	8	9	10
A	7	4	1	15	9	5	12	3	18	11

Let **State *j*** be a state of the array A in which the smallest j numbers of $A[1..n]$ are placed in sorted order (i.e., nondecreasing order of value) in $A[1..j]$, and the remaining numbers placed in arbitrary order in $A[j+1..n]$.

State 0

Input array

	1	2	3	4	5	6	7	8	9	10
A	7	4	1	15	9	5	12	3	18	11

smallest 4 numbers in sorted order

remaining 6 numbers

State 4

Suppose somehow we have this:

	1	2	3	4	5	6	7	8	9	10
A	1	3	4	5	9	15	12	7	18	11

smallest 5 numbers in sorted order

remaining 5 numbers

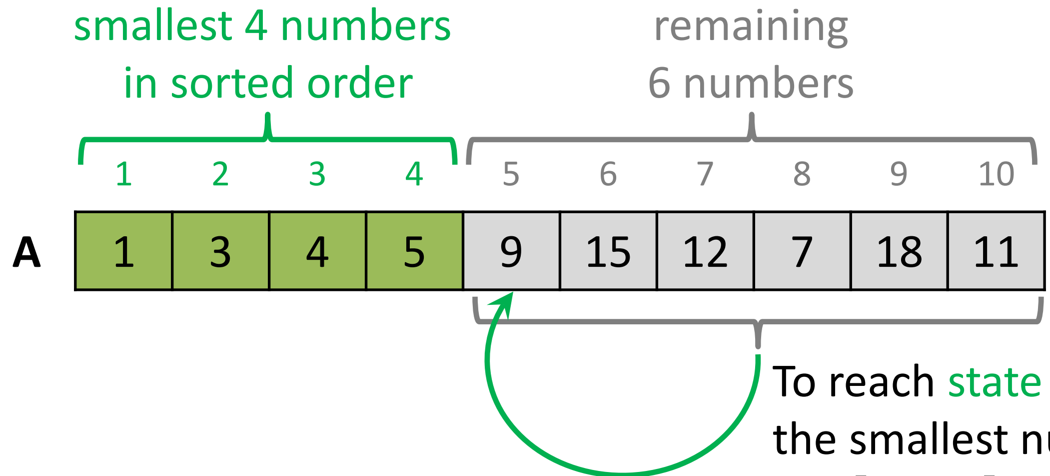
State 5

Now from state 4 we want to reach this:

	1	2	3	4	5	6	7	8	9	10
A	1	3	4	5	7	15	12	9	18	11

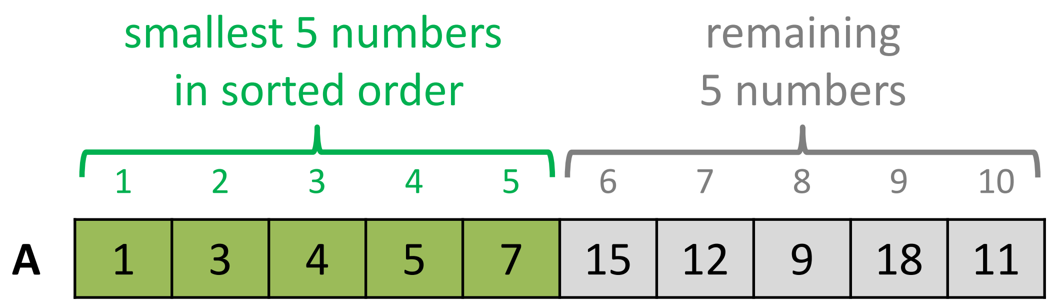
Selection Sort

State 4
Suppose somehow we have this:



To reach state 5 find the smallest number in $A[5..10]$ and swap that with $A[5]$.

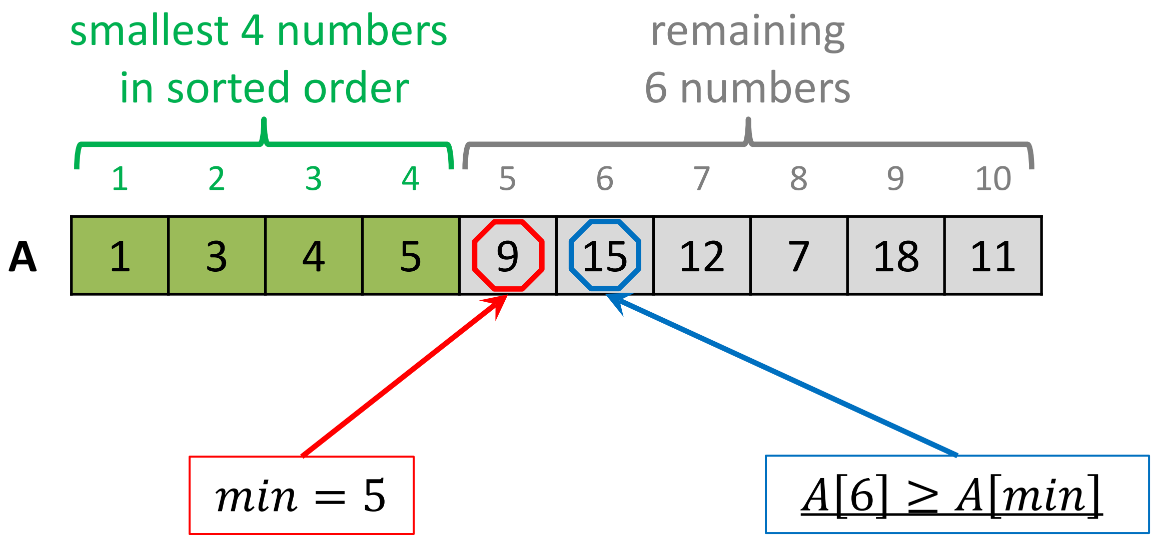
State 5
Now from state 4 we want reach this:



Selection Sort

State 4

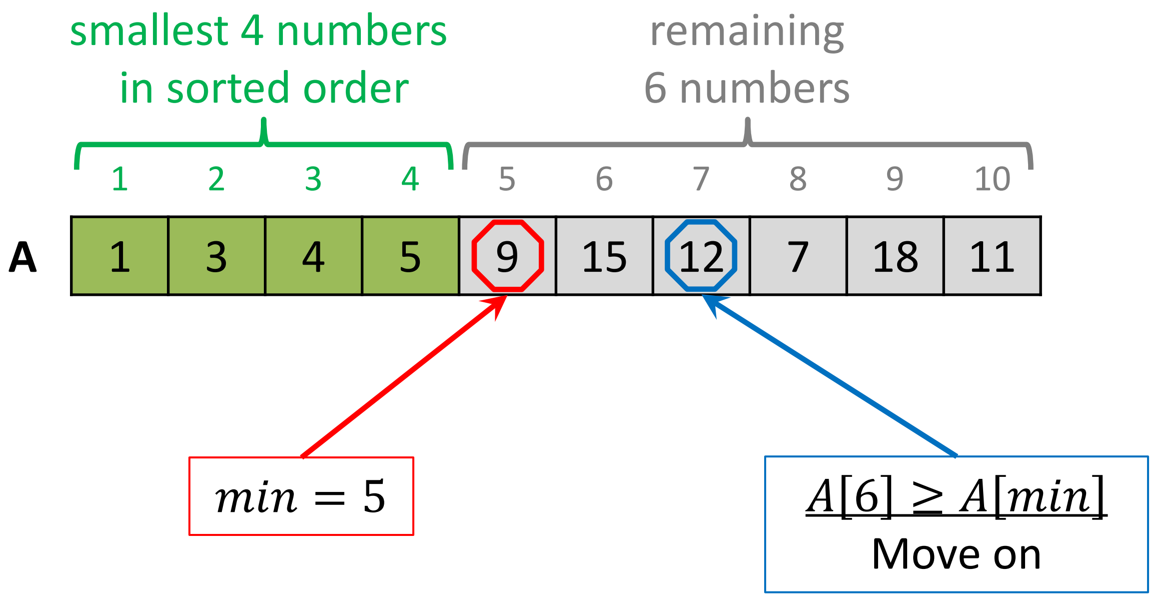
Suppose somehow we have this:



Selection Sort

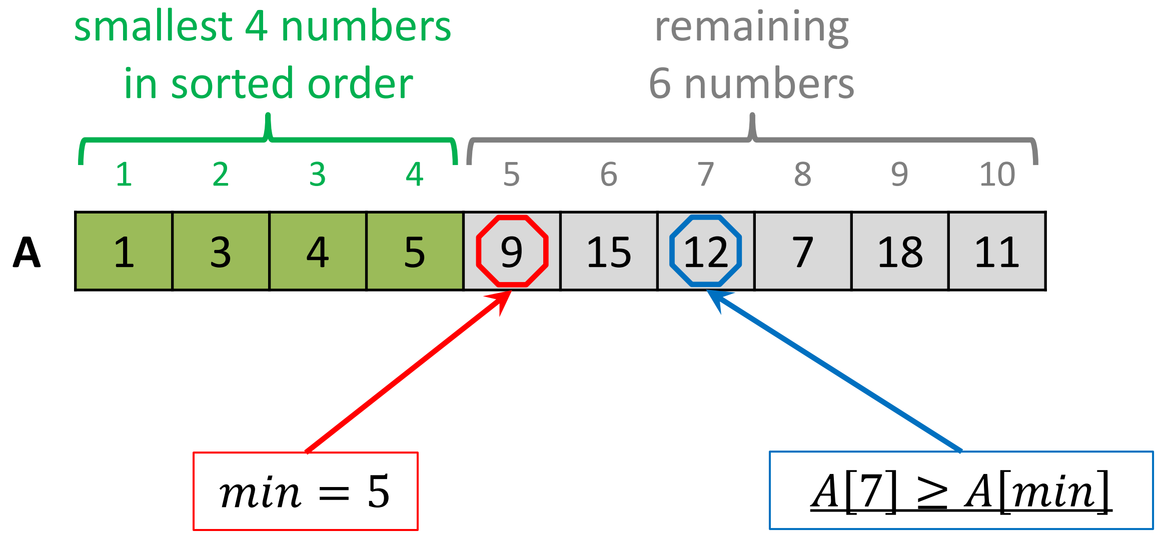
State 4

Suppose somehow we have this:



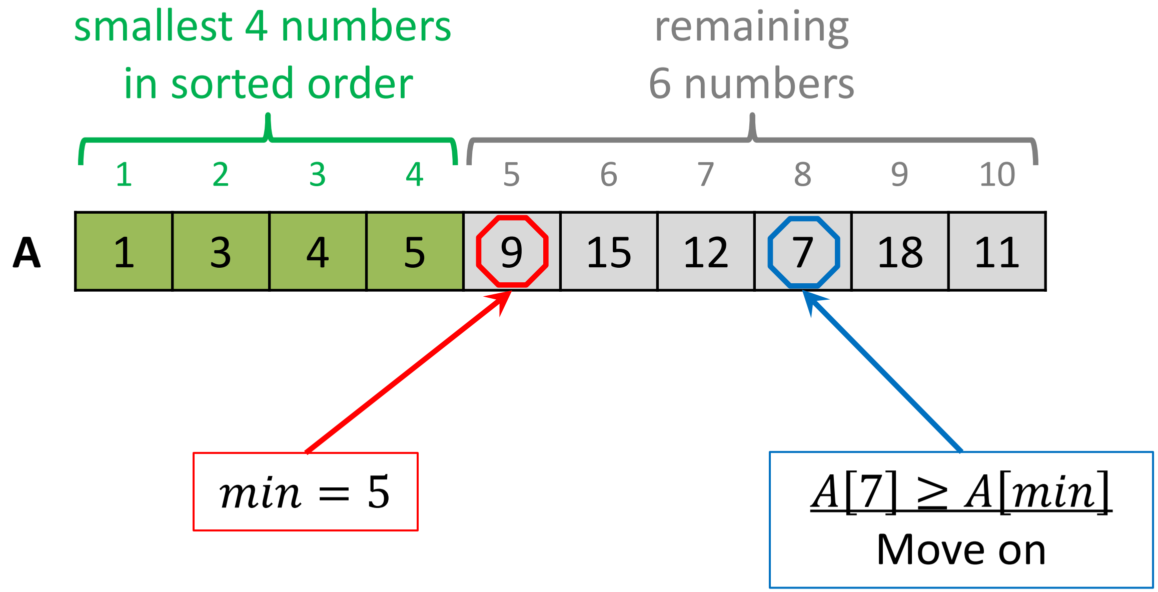
Selection Sort

State 4
Suppose somehow we have this:



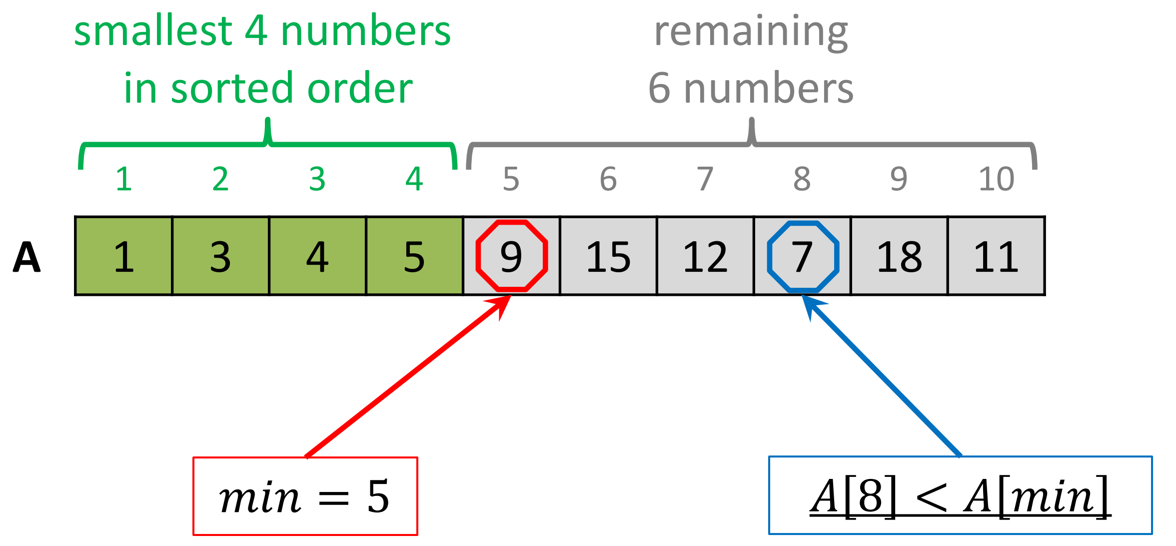
Selection Sort

State 4
Suppose somehow we have this:



Selection Sort

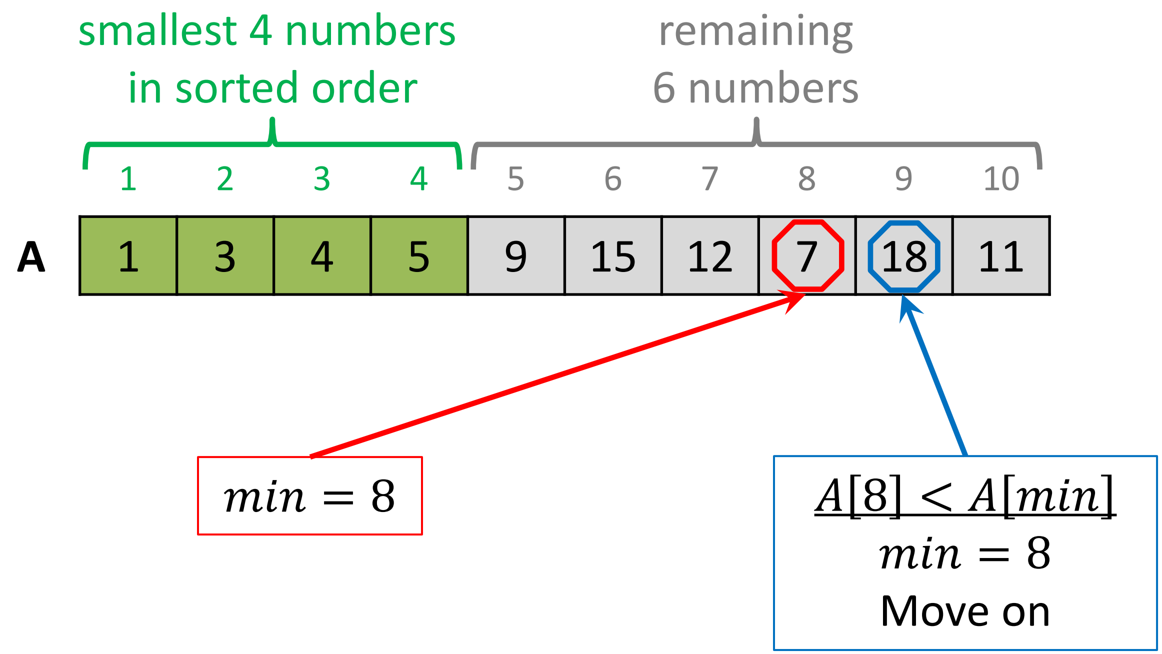
State 4
Suppose somehow we have this:



Selection Sort

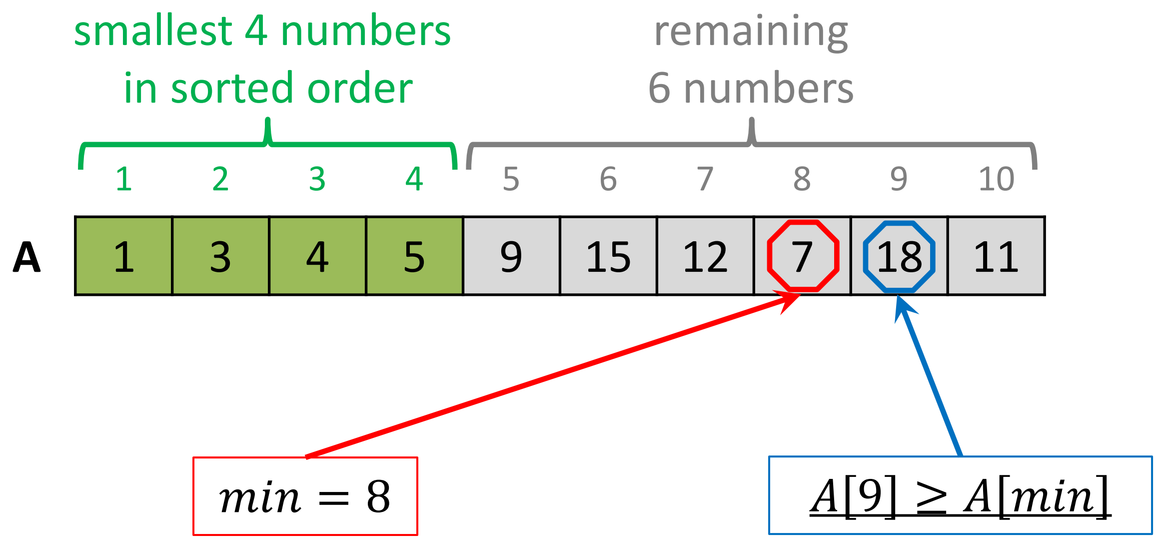
State 4

Suppose somehow we have this:



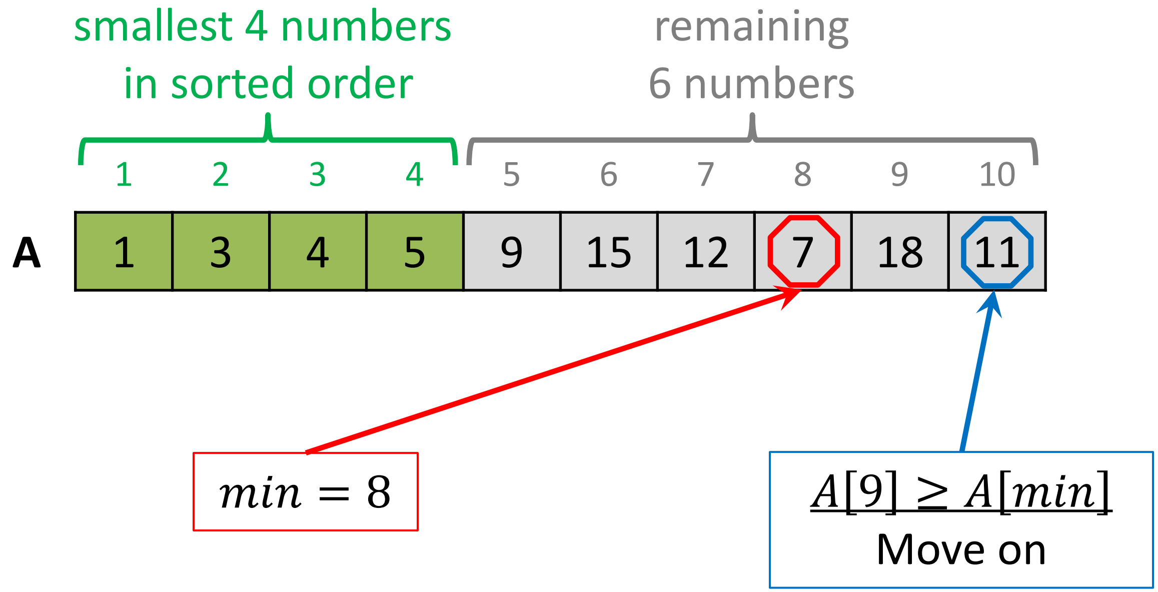
Selection Sort

State 4
Suppose somehow we have this:



Selection Sort

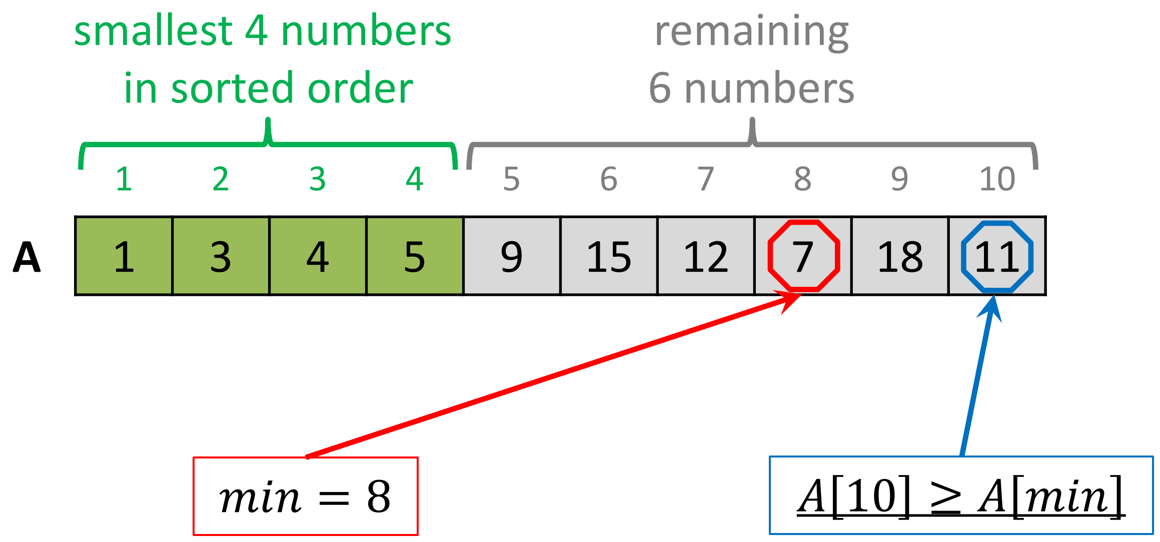
State 4
Suppose somehow
we have this:



Selection Sort

State 4

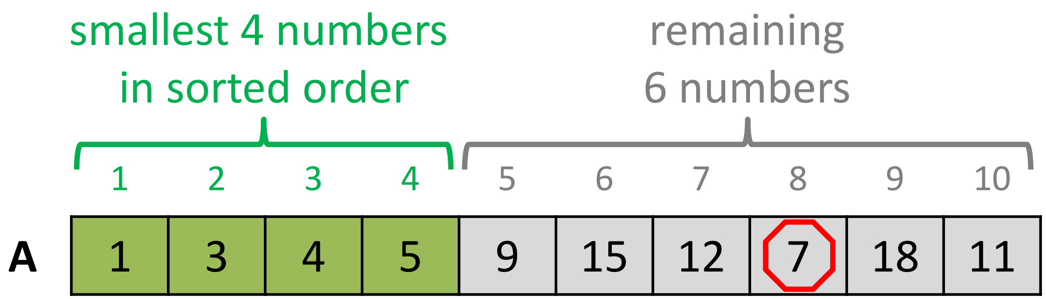
Suppose somehow we have this:



Selection Sort

State 4

Suppose somehow we have this:



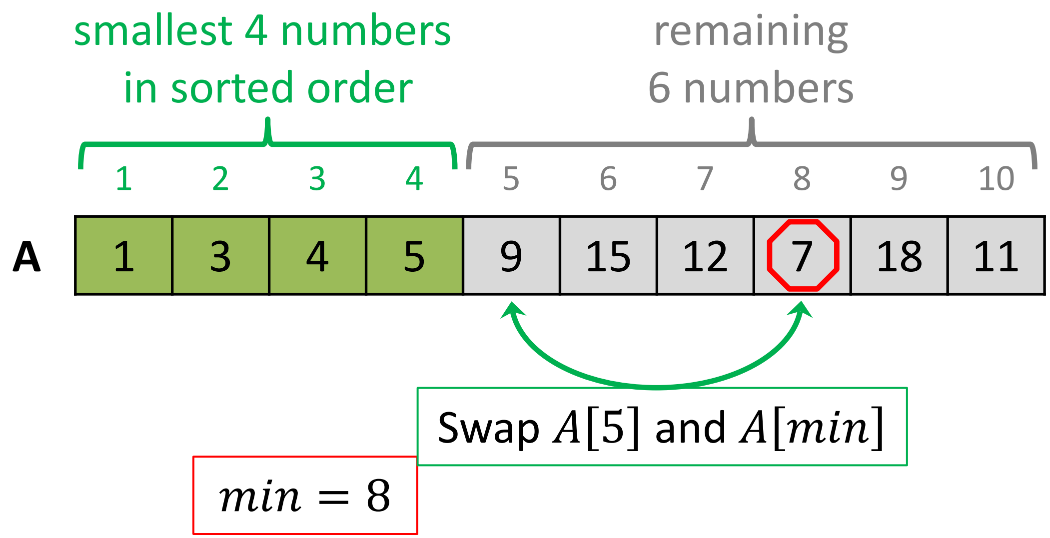
$min = 8$

$\underline{A[10] \geq A[min]}$

Done scanning

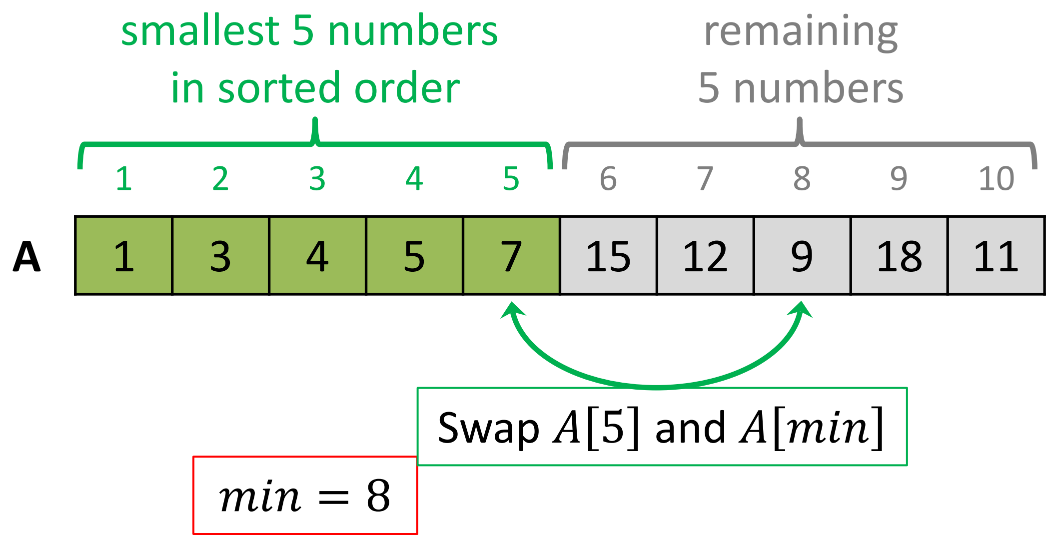
Selection Sort

State 4
Suppose somehow we have this:



Selection Sort

State 5



Selection Sort

State 0

Input array

	1	2	3	4	5	6	7	8	9	10
A	7	4	1	15	9	5	12	3	18	11

↓ Swap $A[1]$ with the smallest number in $A[1..10]$

State 1

$A[1]$ is now sorted

	1	2	3	4	5	6	7	8	9	10
A	1	4	7	15	9	5	12	3	18	11

↓ Swap $A[2]$ with the smallest number in $A[2..10]$

State 2

$A[1..2]$ is now sorted

	1	2	3	4	5	6	7	8	9	10
A	1	3	7	15	9	5	12	4	18	11

↓ Swap $A[3]$ with the smallest number in $A[3..10]$

State 3

$A[1..3]$ is now sorted

	1	2	3	4	5	6	7	8	9	10
A	1	3	4	15	9	5	12	7	18	11

↓ Swap $A[4]$ with the smallest number in $A[4..10]$

State 4

$A[1..4]$ is now sorted

	1	2	3	4	5	6	7	8	9	10
A	1	3	4	5	9	15	12	7	18	11

↓ Swap $A[5]$ with the smallest number in $A[5..10]$

State 5

$A[1..5]$ is now sorted

	1	2	3	4	5	6	7	8	9	10
A	1	3	4	5	7	15	12	9	18	11

Selection Sort

State 5

$A[1..5]$ is now sorted

	1	2	3	4	5	6	7	8	9	10
A	1	3	4	5	7	15	12	9	18	11

Swap $A[6]$ with the smallest number in $A[6..10]$

State 6

$A[1..6]$ is now sorted

	1	2	3	4	5	6	7	8	9	10
A	1	3	4	5	7	9	12	15	18	11

Swap $A[7]$ with the smallest number in $A[7..10]$

State 7

$A[1..7]$ is now sorted

	1	2	3	4	5	6	7	8	9	10
A	1	3	4	5	7	9	11	15	18	12

Swap $A[8]$ with the smallest number in $A[8..10]$

State 8

$A[1..8]$ is now sorted

	1	2	3	4	5	6	7	8	9	10
A	1	3	4	5	7	9	11	12	18	15

Swap $A[9]$ with the smallest number in $A[9..10]$

State 9

$A[1..9]$ is now sorted

	1	2	3	4	5	6	7	8	9	10
A	1	3	4	5	7	9	11	12	15	18

Do nothing!

State 10

$A[1..10]$ is now sorted

	1	2	3	4	5	6	7	8	9	10
A	1	3	4	5	7	9	11	12	15	18

Selection Sort

Input: An array $A[\,1:\ n\,]$ of n numbers.

Output: Elements of $A[\,1:\ n\,]$ rearranged in non-decreasing order of value.

```
SELECTION-SORT ( A )

for j = 1 to A.length − 1
    // find the index of an entry with the smallest value in A[j..A.length]
    min = j
    for i = j + 1 to A.length
        if A[i] < A[min]
            min = i
    // swap A[j] and A[min]
    A[j] ↔ A[min]
```

[Optional]
Proof of Correctness
of Insertion Sort

Loop Invariants

We use *loop invariants* to prove correctness of iterative algorithms

A loop invariant is associated with a given loop of an algorithm, and it is a formal statement about the relationship among variables of the algorithm such that

- **[Initialization]** It is true prior to the first iteration of the loop
- **[Maintenance]** If it is true before an iteration of the loop, it remains true before the next iteration
- **[Termination]** When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct

Loop Invariants for Insertion Sort

INSERTION-SORT (A)

$$
\begin{array}{l}
\textbf{for } j = 2 \textbf{ to } A.length \\
\quad key = A[j] \\
\quad \text{// insert } A[j] \text{ into the sorted sequence } A[1..j-1] \\
\quad i = j - 1 \\
\quad \textbf{while } i > 0 \textbf{ and } A[i] > key \\
\quad\quad A[i+1] = A[i] \\
\quad\quad i = i - 1 \\
\quad A[i+1] = key
\end{array}
$$

Loop Invariants for Insertion Sort

INSERTION-SORT (A)

for $j = 2$ **to** $A.length$

> **Invariant 1:** $A[1..j-1]$ consists of the elements originally in $A[1..j-1]$, but in sorted order

    $key = A[j]$

    // insert $A[j]$ into the sorted sequence $A[1..j-1]$

    $i = j - 1$

    while $i > 0$ *and* $A[i] > key$

        $A[i+1] = A[i]$

        $i = i - 1$

    $A[i+1] = key$

Loop Invariants for Insertion Sort

INSERTION-SORT (A)

for $j = 2$ **to** $A.length$

> **Invariant 1:** $A[1..j-1]$ consists of the elements originally in $A[1..j-1]$, but in sorted order

    $key = A[j]$

    // insert $A[j]$ into the sorted sequence $A[1..j-1]$

    $i = j - 1$

    while $i > 0$ **and** $A[i] > key$

> **Invariant 2:** $A[i..j]$ are each $\geq key$

        $A[i+1] = A[i]$

        $i = i - 1$

    $A[i+1] = key$

Loop Invariant 1: Initialization

```
INSERTION-SORT ( A )

1. for j = 2 to A.length
       Invariant 1: A[1..j − 1] consists of the elements
                    originally in A[1..j − 1], but in sorted order
2.     key = A[j]
3.     // insert A[j] into the sorted sequence A[1..j − 1]
4.     i = j − 1
5.     while i > 0 and A[i] > key
           Invariant 2: A[i..j] are each ≥ key
6.         A[i + 1] = A[i]
7.         i = i − 1
8.     A[i + 1] = key
```

At the start of the first iteration of the loop (in lines $1 - 8$): $j = 2$

Hence, subarray $A[1..j-1]$ consists of a single element $A[1]$, which is in fact the original element in $A[1]$.

The subarray consisting of a single element is trivially sorted.

Hence, the invariant holds initially.

Loop Invariant 1: Maintenance

```
INSERTION-SORT ( A )
for j = 2 to A.length
    Invariant 1: A[1..j − 1] consists of the elements
                 originally in A[1..j − 1], but in sorted order
    key = A[j]
    // insert A[j] into the sorted sequence A[1..j − 1]
    i = j − 1
    while i > 0 and A[i] > key
        Invariant 2: A[i..j] are each ≥ key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

We assume that invariant 1 holds before the start of the current iteration.

Hence, the following holds: $A[1..j-1]$ consists of the elements originally in $A[1..j-1]$, but in sorted order.

For invariant 1 to hold before the start of the next iteration, the following must hold at the end of the current iteration:

$A[1..j]$ consists of the elements originally in $A[1..j]$, but in sorted order.

We use invariant 2 to prove this.

Loop Invariant 1: Maintenance

Loop Invariant 2: Initialization

```
INSERTION-SORT ( A )

1.  for j = 2 to A.length
        Invariant 1: A[1..j − 1] consists of the elements
                     originally in A[1..j − 1], but in sorted order
2.      key = A[j]
3.      // insert A[j] into the sorted sequence A[1..j − 1]
4.      i = j − 1
5.      while i > 0 and A[i] > key
            Invariant 2: A[i..j] are each ≥ key
6.          A[i + 1] = A[i]
7.          i = i − 1
8.      A[i + 1] = key
```

At the start of the first iteration of the loop (in lines $5-7$): $i = j - 1$

Hence, subarray $A[i..j]$ consists of only two entries: $A[i]$ and $A[j]$.

We know the following:

— $A[i] > key$ (explicitly tested in line 5)
— $A[j] = key$ (from line 2)

Hence, invariant 2 holds initially.

Loop Invariant 1: Maintenance
Loop Invariant 2: Maintenance

```
INSERTION-SORT ( A )

1.  for j = 2 to A.length
        Invariant 1: A[1..j − 1] consists of the elements
                     originally in A[1..j − 1], but in sorted order
2.      key = A[j]
3.      // insert A[j] into the sorted sequence A[1..j − 1]
4.      i = j − 1
5.      while i > 0 and A[i] > key
          Invariant 2: A[i..j] are each ≥ key
6.          A[i + 1] = A[i]
7.          i = i − 1
8.      A[i + 1] = key
```

We assume that invariant 2 holds before the start of the current iteration.

Hence, the following holds: $A[i..j]$ are each $\geq key$.

Since line 6 copies $A[i]$ which is known to be $> key$ to $A[i+1]$ which also held a value $\geq key$, the following holds at the end of the current iteration: $A[i+1..j]$ are each $\geq key$.

Before the start of the next iteration the check $A[i] > key$ in line 5 ensures that invariant 2 continues to hold.

Loop Invariant 1: Maintenance

Loop Invariant 2: Maintenance

```
Insertion-Sort ( A )

1.  for j = 2 to A.length
        Invariant 1: A[1..j − 1] consists of the elements
                     originally in A[1..j − 1], but in sorted order
2.      key = A[j]
3.      // insert A[j] into the sorted sequence A[1..j − 1]
4.      i = j − 1
5.      while i > 0 and A[i] > key
            Invariant 2: A[i..j] are each ≥ key
6.          A[i + 1] = A[i]
7.          i = i − 1
8.      A[i + 1] = key
```

Observe that the inner loop (in lines $5 - 7$) does not destroy any data because though the first iteration overwrites $A[j]$, that $A[j]$ has already been saved in key in line 2.

As long as key is copied back into a location in $A[1..j]$ without destroying any other element in that subarray, we maintain the invariant that $A[1..j]$ contains the first j elements of the original list.

Loop Invariant 1: Maintenance
Loop Invariant 2: Termination

```
INSERTION-SORT ( A )

for j = 2 to A.length
    Invariant 1: A[1..j − 1] consists of the elements
                 originally in A[1..j − 1], but in sorted order
    key = A[j]
    // insert A[j] into the sorted sequence A[1..j − 1]
    i = j − 1
    while i > 0 and A[i] > key
        Invariant 2: A[i..j] are each ≥ key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

When the inner loop terminates we know the following.

- $A[1..i]$ is sorted with each element $\leq key$
  - if $i = 0$, true by default
  - if $i > 0$, true because $A[1..i]$ is sorted and $A[i] \leq key$
- $A[i+1..j]$ is sorted with each element $\geq key$ because the following held before i was decremented: $A[i..j]$ is sorted with each item $\geq key$
- $A[i+1] = A[i+2]$ if the loop was executed at least once, and $A[i+1] = key$ otherwise

Loop Invariant 1: Maintenance
Loop Invariant 2: Termination

```
INSERTION-SORT ( A )

1.  for j = 2 to A.length
        Invariant 1: A[1..j − 1] consists of the elements
                     originally in A[1..j − 1], but in sorted order
2.      key = A[j]
3.      // insert A[j] into the sorted sequence A[1..j − 1]
4.      i = j − 1
5.      while i > 0 and A[i] > key
            Invariant 2: A[i..j] are each ≥ key
6.          A[i + 1] = A[i]
7.          i = i − 1
8.      A[i + 1] = key
```

When the inner loop terminates we know the following.

– $A[1..i]$ is sorted with each element $\leq key$

– $A[i+1..j]$ is sorted with each element $\geq key$

– $A[i+1] = A[i+2]$ or $A[i+1] = key$

Given the facts above, line 8 does not destroy any data, and gives us $A[1..j]$ as the sorted permutation of the original data in $A[1..j]$.

Loop Invariant 1: Termination

```
INSERTION-SORT ( A )

for j = 2 to A.length
    Invariant 1: A[1..j − 1] consists of the elements
                 originally in A[1..j − 1], but in sorted order
    key = A[j]
    // insert A[j] into the sorted sequence A[1..j − 1]
    i = j − 1
    while i > 0 and A[i] > key
        Invariant 2: A[i..j] are each ≥ key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

When the outer loop terminates we know that $j = A.length + 1$.

Hence, $A[1..j-1]$ is the entire array $A[1..A.length]$, which is sorted and contains the original elements of $A[1..A.length]$.