

Analysis of Insertion Sort

Antonio Carzaniga

Faculty of Informatics
Università della Svizzera italiana

February 25, 2025

- Sorting
- Insertion Sort
- Analysis

- **Input:** a sequence $A = \langle a_1, a_2, \dots, a_n \rangle$

■ **Input:** a sequence $A = \langle a_1, a_2, \dots, a_n \rangle$

Output: a sequence $\langle b_1, b_2, \dots, b_n \rangle$ such that

- ▶ $\langle b_1, b_2, \dots, b_n \rangle$ is a *permutation* of $\langle a_1, a_2, \dots, a_n \rangle$

■ **Input:** a sequence $A = \langle a_1, a_2, \dots, a_n \rangle$

Output: a sequence $\langle b_1, b_2, \dots, b_n \rangle$ such that

- ▶ $\langle b_1, b_2, \dots, b_n \rangle$ is a *permutation* of $\langle a_1, a_2, \dots, a_n \rangle$
- ▶ $\langle b_1, b_2, \dots, b_n \rangle$ is *sorted*

$$b_1 \leq b_2 \leq \dots \leq b_n$$

■ **Input:** a sequence $A = \langle a_1, a_2, \dots, a_n \rangle$

Output: a sequence $\langle b_1, b_2, \dots, b_n \rangle$ such that

- ▶ $\langle b_1, b_2, \dots, b_n \rangle$ is a *permutation* of $\langle a_1, a_2, \dots, a_n \rangle$
- ▶ $\langle b_1, b_2, \dots, b_n \rangle$ is *sorted*

$$b_1 \leq b_2 \leq \dots \leq b_n$$

■ Typically, A is implemented as an array

■ **Input:** a sequence $A = \langle a_1, a_2, \dots, a_n \rangle$

Output: a sequence $\langle b_1, b_2, \dots, b_n \rangle$ such that

- ▶ $\langle b_1, b_2, \dots, b_n \rangle$ is a *permutation* of $\langle a_1, a_2, \dots, a_n \rangle$
- ▶ $\langle b_1, b_2, \dots, b_n \rangle$ is *sorted*

$$b_1 \leq b_2 \leq \dots \leq b_n$$

■ Typically, A is implemented as an array

$A =$

6	8	3	2	7	6	11	5	9	4
---	---	---	---	---	---	----	---	---	---

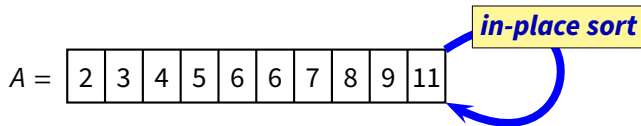
■ **Input:** a sequence $A = \langle a_1, a_2, \dots, a_n \rangle$

Output: a sequence $\langle b_1, b_2, \dots, b_n \rangle$ such that

- ▶ $\langle b_1, b_2, \dots, b_n \rangle$ is a *permutation* of $\langle a_1, a_2, \dots, a_n \rangle$
- ▶ $\langle b_1, b_2, \dots, b_n \rangle$ is *sorted*

$$b_1 \leq b_2 \leq \dots \leq b_n$$

■ Typically, A is implemented as an array



- **Idea:** it is like sorting a hand of cards

■ **Idea:** it is like sorting a hand of cards

- ▶ scan the sequence left to right
- ▶ pick the value at the current position a_j
- ▶ insert it in its correct position in the sequence $\langle a_1, a_2, \dots, a_{j-1} \rangle$ so as to maintain a sorted subsequence $\langle a_1, a_2, \dots, a_j \rangle$

■ **Idea:** it is like sorting a hand of cards

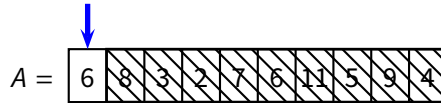
- ▶ scan the sequence left to right
- ▶ pick the value at the current position a_j
- ▶ insert it in its correct position in the sequence $\langle a_1, a_2, \dots, a_{j-1} \rangle$ so as to maintain a sorted subsequence $\langle a_1, a_2, \dots, a_j \rangle$

$A =$

6	8	3	2	7	6	11	5	9	4
---	---	---	---	---	---	----	---	---	---

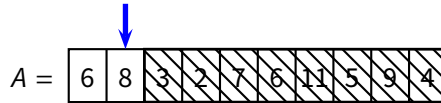
■ **Idea:** it is like sorting a hand of cards

- ▶ scan the sequence left to right
- ▶ pick the value at the current position a_j
- ▶ insert it in its correct position in the sequence $\langle a_1, a_2, \dots, a_{j-1} \rangle$ so as to maintain a sorted subsequence $\langle a_1, a_2, \dots, a_j \rangle$



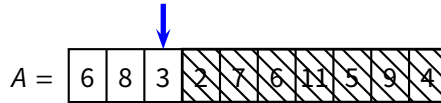
■ **Idea:** it is like sorting a hand of cards

- ▶ scan the sequence left to right
- ▶ pick the value at the current position a_j
- ▶ insert it in its correct position in the sequence $\langle a_1, a_2, \dots, a_{j-1} \rangle$ so as to maintain a sorted subsequence $\langle a_1, a_2, \dots, a_j \rangle$



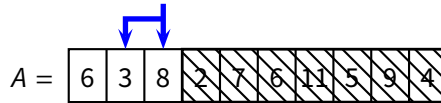
■ **Idea:** it is like sorting a hand of cards

- ▶ scan the sequence left to right
- ▶ pick the value at the current position a_j
- ▶ insert it in its correct position in the sequence $\langle a_1, a_2, \dots, a_{j-1} \rangle$ so as to maintain a sorted subsequence $\langle a_1, a_2, \dots, a_j \rangle$



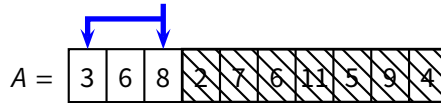
■ **Idea:** it is like sorting a hand of cards

- ▶ scan the sequence left to right
- ▶ pick the value at the current position a_j
- ▶ insert it in its correct position in the sequence $\langle a_1, a_2, \dots, a_{j-1} \rangle$ so as to maintain a sorted subsequence $\langle a_1, a_2, \dots, a_j \rangle$



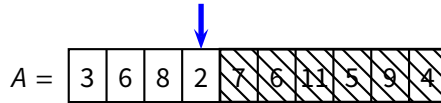
■ **Idea:** it is like sorting a hand of cards

- ▶ scan the sequence left to right
- ▶ pick the value at the current position a_j
- ▶ insert it in its correct position in the sequence $\langle a_1, a_2, \dots, a_{j-1} \rangle$ so as to maintain a sorted subsequence $\langle a_1, a_2, \dots, a_j \rangle$



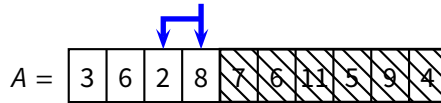
■ **Idea:** it is like sorting a hand of cards

- ▶ scan the sequence left to right
- ▶ pick the value at the current position a_j
- ▶ insert it in its correct position in the sequence $\langle a_1, a_2, \dots, a_{j-1} \rangle$ so as to maintain a sorted subsequence $\langle a_1, a_2, \dots, a_j \rangle$



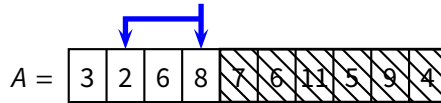
■ **Idea:** it is like sorting a hand of cards

- ▶ scan the sequence left to right
- ▶ pick the value at the current position a_j
- ▶ insert it in its correct position in the sequence $\langle a_1, a_2, \dots, a_{j-1} \rangle$ so as to maintain a sorted subsequence $\langle a_1, a_2, \dots, a_j \rangle$



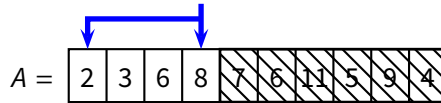
■ **Idea:** it is like sorting a hand of cards

- ▶ scan the sequence left to right
- ▶ pick the value at the current position a_j
- ▶ insert it in its correct position in the sequence $\langle a_1, a_2, \dots, a_{j-1} \rangle$ so as to maintain a sorted subsequence $\langle a_1, a_2, \dots, a_j \rangle$



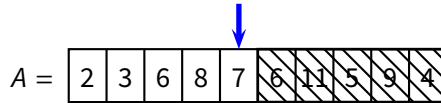
■ **Idea:** it is like sorting a hand of cards

- ▶ scan the sequence left to right
- ▶ pick the value at the current position a_j
- ▶ insert it in its correct position in the sequence $\langle a_1, a_2, \dots, a_{j-1} \rangle$ so as to maintain a sorted subsequence $\langle a_1, a_2, \dots, a_j \rangle$



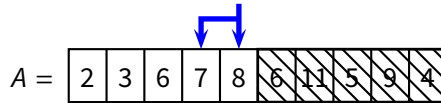
■ **Idea:** it is like sorting a hand of cards

- ▶ scan the sequence left to right
- ▶ pick the value at the current position a_j
- ▶ insert it in its correct position in the sequence $\langle a_1, a_2, \dots, a_{j-1} \rangle$ so as to maintain a sorted subsequence $\langle a_1, a_2, \dots, a_j \rangle$



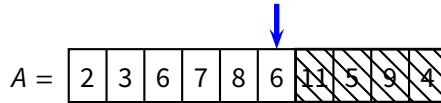
■ **Idea:** it is like sorting a hand of cards

- ▶ scan the sequence left to right
- ▶ pick the value at the current position a_j
- ▶ insert it in its correct position in the sequence $\langle a_1, a_2, \dots, a_{j-1} \rangle$ so as to maintain a sorted subsequence $\langle a_1, a_2, \dots, a_j \rangle$



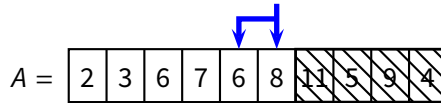
■ **Idea:** it is like sorting a hand of cards

- ▶ scan the sequence left to right
- ▶ pick the value at the current position a_j
- ▶ insert it in its correct position in the sequence $\langle a_1, a_2, \dots, a_{j-1} \rangle$ so as to maintain a sorted subsequence $\langle a_1, a_2, \dots, a_j \rangle$



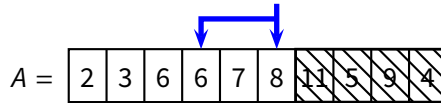
■ **Idea:** it is like sorting a hand of cards

- ▶ scan the sequence left to right
- ▶ pick the value at the current position a_j
- ▶ insert it in its correct position in the sequence $\langle a_1, a_2, \dots, a_{j-1} \rangle$ so as to maintain a sorted subsequence $\langle a_1, a_2, \dots, a_j \rangle$



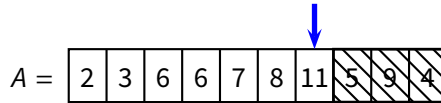
■ **Idea:** it is like sorting a hand of cards

- ▶ scan the sequence left to right
- ▶ pick the value at the current position a_j
- ▶ insert it in its correct position in the sequence $\langle a_1, a_2, \dots, a_{j-1} \rangle$ so as to maintain a sorted subsequence $\langle a_1, a_2, \dots, a_j \rangle$



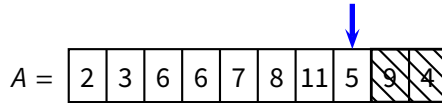
■ **Idea:** it is like sorting a hand of cards

- ▶ scan the sequence left to right
- ▶ pick the value at the current position a_j
- ▶ insert it in its correct position in the sequence $\langle a_1, a_2, \dots, a_{j-1} \rangle$ so as to maintain a sorted subsequence $\langle a_1, a_2, \dots, a_j \rangle$



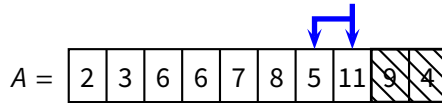
■ **Idea:** it is like sorting a hand of cards

- ▶ scan the sequence left to right
- ▶ pick the value at the current position a_j
- ▶ insert it in its correct position in the sequence $\langle a_1, a_2, \dots, a_{j-1} \rangle$ so as to maintain a sorted subsequence $\langle a_1, a_2, \dots, a_j \rangle$



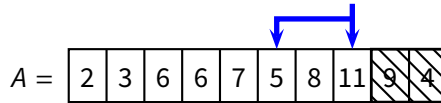
■ **Idea:** it is like sorting a hand of cards

- ▶ scan the sequence left to right
- ▶ pick the value at the current position a_j
- ▶ insert it in its correct position in the sequence $\langle a_1, a_2, \dots, a_{j-1} \rangle$ so as to maintain a sorted subsequence $\langle a_1, a_2, \dots, a_j \rangle$



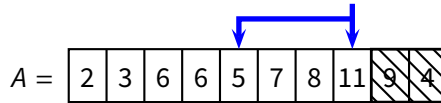
■ **Idea:** it is like sorting a hand of cards

- ▶ scan the sequence left to right
- ▶ pick the value at the current position a_j
- ▶ insert it in its correct position in the sequence $\langle a_1, a_2, \dots, a_{j-1} \rangle$ so as to maintain a sorted subsequence $\langle a_1, a_2, \dots, a_j \rangle$



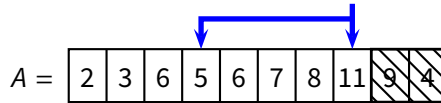
■ **Idea:** it is like sorting a hand of cards

- ▶ scan the sequence left to right
- ▶ pick the value at the current position a_j
- ▶ insert it in its correct position in the sequence $\langle a_1, a_2, \dots, a_{j-1} \rangle$ so as to maintain a sorted subsequence $\langle a_1, a_2, \dots, a_j \rangle$



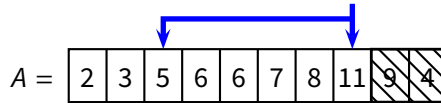
■ **Idea:** it is like sorting a hand of cards

- ▶ scan the sequence left to right
- ▶ pick the value at the current position a_j
- ▶ insert it in its correct position in the sequence $\langle a_1, a_2, \dots, a_{j-1} \rangle$ so as to maintain a sorted subsequence $\langle a_1, a_2, \dots, a_j \rangle$



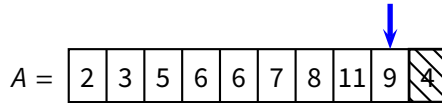
■ **Idea:** it is like sorting a hand of cards

- ▶ scan the sequence left to right
- ▶ pick the value at the current position a_j
- ▶ insert it in its correct position in the sequence $\langle a_1, a_2, \dots, a_{j-1} \rangle$ so as to maintain a sorted subsequence $\langle a_1, a_2, \dots, a_j \rangle$



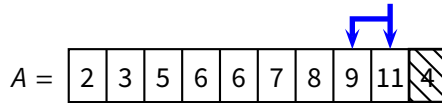
■ **Idea:** it is like sorting a hand of cards

- ▶ scan the sequence left to right
- ▶ pick the value at the current position a_j
- ▶ insert it in its correct position in the sequence $\langle a_1, a_2, \dots, a_{j-1} \rangle$ so as to maintain a sorted subsequence $\langle a_1, a_2, \dots, a_j \rangle$



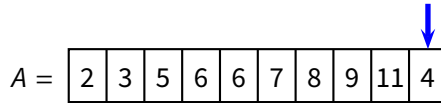
■ **Idea:** it is like sorting a hand of cards

- ▶ scan the sequence left to right
- ▶ pick the value at the current position a_j
- ▶ insert it in its correct position in the sequence $\langle a_1, a_2, \dots, a_{j-1} \rangle$ so as to maintain a sorted subsequence $\langle a_1, a_2, \dots, a_j \rangle$



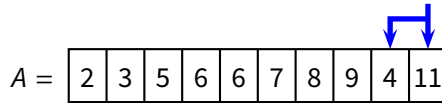
■ **Idea:** it is like sorting a hand of cards

- ▶ scan the sequence left to right
- ▶ pick the value at the current position a_j
- ▶ insert it in its correct position in the sequence $\langle a_1, a_2, \dots, a_{j-1} \rangle$ so as to maintain a sorted subsequence $\langle a_1, a_2, \dots, a_j \rangle$



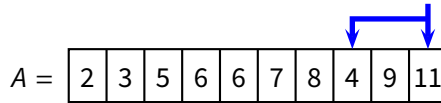
■ **Idea:** it is like sorting a hand of cards

- ▶ scan the sequence left to right
- ▶ pick the value at the current position a_j
- ▶ insert it in its correct position in the sequence $\langle a_1, a_2, \dots, a_{j-1} \rangle$ so as to maintain a sorted subsequence $\langle a_1, a_2, \dots, a_j \rangle$



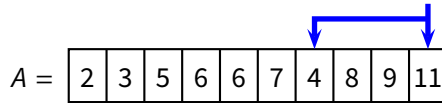
■ **Idea:** it is like sorting a hand of cards

- ▶ scan the sequence left to right
- ▶ pick the value at the current position a_j
- ▶ insert it in its correct position in the sequence $\langle a_1, a_2, \dots, a_{j-1} \rangle$ so as to maintain a sorted subsequence $\langle a_1, a_2, \dots, a_j \rangle$



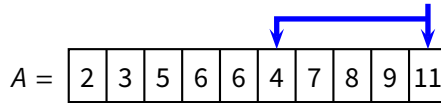
■ **Idea:** it is like sorting a hand of cards

- ▶ scan the sequence left to right
- ▶ pick the value at the current position a_j
- ▶ insert it in its correct position in the sequence $\langle a_1, a_2, \dots, a_{j-1} \rangle$ so as to maintain a sorted subsequence $\langle a_1, a_2, \dots, a_j \rangle$



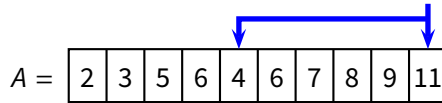
■ **Idea:** it is like sorting a hand of cards

- ▶ scan the sequence left to right
- ▶ pick the value at the current position a_j
- ▶ insert it in its correct position in the sequence $\langle a_1, a_2, \dots, a_{j-1} \rangle$ so as to maintain a sorted subsequence $\langle a_1, a_2, \dots, a_j \rangle$



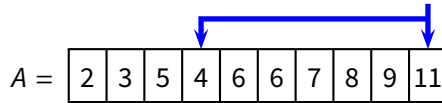
■ **Idea:** it is like sorting a hand of cards

- ▶ scan the sequence left to right
- ▶ pick the value at the current position a_j
- ▶ insert it in its correct position in the sequence $\langle a_1, a_2, \dots, a_{j-1} \rangle$ so as to maintain a sorted subsequence $\langle a_1, a_2, \dots, a_j \rangle$



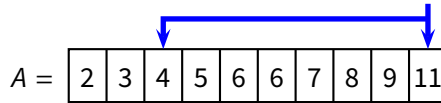
■ **Idea:** it is like sorting a hand of cards

- ▶ scan the sequence left to right
- ▶ pick the value at the current position a_j
- ▶ insert it in its correct position in the sequence $\langle a_1, a_2, \dots, a_{j-1} \rangle$ so as to maintain a sorted subsequence $\langle a_1, a_2, \dots, a_j \rangle$



■ **Idea:** it is like sorting a hand of cards

- ▶ scan the sequence left to right
- ▶ pick the value at the current position a_j
- ▶ insert it in its correct position in the sequence $\langle a_1, a_2, \dots, a_{j-1} \rangle$ so as to maintain a sorted subsequence $\langle a_1, a_2, \dots, a_j \rangle$



■ **Idea:** it is like sorting a hand of cards

- ▶ scan the sequence left to right
- ▶ pick the value at the current position a_j
- ▶ insert it in its correct position in the sequence $\langle a_1, a_2, \dots, a_{j-1} \rangle$ so as to maintain a sorted subsequence $\langle a_1, a_2, \dots, a_j \rangle$

$A =$

2	3	4	5	6	6	7	8	9	11
---	---	---	---	---	---	---	---	---	----

INSERTION-SORT(A)

```
1  for  $i = 2$  to  $\text{length}(A)$ 
2       $j = i$ 
3      while  $j > 1$  and  $A[j - 1] > A[j]$ 
4          swap  $A[j]$  and  $A[j - 1]$ 
5           $j = j - 1$ 
```

INSERTION-SORT(A)

```
1  for  $i = 2$  to  $\text{length}(A)$ 
2       $j = i$ 
3      while  $j > 1$  and  $A[j - 1] > A[j]$ 
4          swap  $A[j]$  and  $A[j - 1]$ 
5           $j = j - 1$ 
```

- Is **INSERTION-SORT** *correct*?
- What is the time complexity of **INSERTION-SORT**?
- Can we do better?

Complexity of INSERTION-SORT

INSERTION-SORT(A)

```
1  for  $i = 2$  to  $length(A)$ 
2       $j = i$ 
3      while  $j > 1$  and  $A[j - 1] > A[j]$ 
4          swap  $A[j]$  and  $A[j - 1]$ 
5           $j = j - 1$ 
```

Complexity of INSERTION-SORT

```
INSERTION-SORT(A)
1  for  $i = 2$  to  $length(A)$ 
2       $j = i$ 
3      while  $j > 1$  and  $A[j - 1] > A[j]$ 
4          swap  $A[j]$  and  $A[j - 1]$ 
5           $j = j - 1$ 
```

- Outer loop (lines 1–5) runs exactly $n - 1$ times (with $n = length(A)$)
- What about the inner loop (lines 3–5)?
 - ▶ best, worst, and average case?

Complexity of INSERTION-SORT (2)

INSERTION-SORT(A)

```
1  for  $i = 2$  to  $length(A)$ 
2       $j = i$ 
3      while  $j > 1$  and  $A[j - 1] > A[j]$ 
4          swap  $A[j]$  and  $A[j - 1]$ 
5           $j = j - 1$ 
```

- **Best case:**

Complexity of INSERTION-SORT (2)

INSERTION-SORT(A)

```
1  for  $i = 2$  to  $length(A)$ 
2       $j = i$ 
3      while  $j > 1$  and  $A[j - 1] > A[j]$ 
4          swap  $A[j]$  and  $A[j - 1]$ 
5           $j = j - 1$ 
```

- **Best case:** the inner loop is *never* executed
 - ▶ what case is this?

Complexity of INSERTION-SORT (2)

INSERTION-SORT(A)

```
1  for  $i = 2$  to  $length(A)$ 
2       $j = i$ 
3      while  $j > 1$  and  $A[j - 1] > A[j]$ 
4          swap  $A[j]$  and  $A[j - 1]$ 
5           $j = j - 1$ 
```

- **Best case:** the inner loop is *never* executed

- ▶ what case is this?

- **Worst case:**

Complexity of INSERTION-SORT (2)

```
INSERTION-SORT(A)
1  for  $i = 2$  to  $\text{length}(A)$ 
2       $j = i$ 
3      while  $j > 1$  and  $A[j - 1] > A[j]$ 
4          swap  $A[j]$  and  $A[j - 1]$ 
5           $j = j - 1$ 
```

- **Best case:** the inner loop is *never* executed
 - ▶ what case is this?
- **Worst case:** the inner loop is executed exactly $j - 1$ times for every iteration of the outer loop
 - ▶ what case is this?

Complexity of INSERTION-SORT (3)

- The worst-case complexity is when the inner loop is executed exactly $j - 1$ times, so

$$T(n) = \sum_{j=2}^n (j - 1)$$

Complexity of INSERTION-SORT (3)

- The worst-case complexity is when the inner loop is executed exactly $j - 1$ times, so

$$T(n) = \sum_{j=2}^n (j - 1)$$

$T(n)$ is the *arithmetic series* $\sum_{k=1}^{n-1} k$, so

$$T(n) = \frac{n(n - 1)}{2}$$

$$T(n) = \Theta(n^2)$$

Complexity of INSERTION-SORT (3)

- The worst-case complexity is when the inner loop is executed exactly $j - 1$ times, so

$$T(n) = \sum_{j=2}^n (j - 1)$$

$T(n)$ is the *arithmetic series* $\sum_{k=1}^{n-1} k$, so

$$T(n) = \frac{n(n - 1)}{2}$$

$$T(n) = \Theta(n^2)$$

- Best-case is $T(n) = \Theta(n)$

Complexity of INSERTION-SORT (3)

- The worst-case complexity is when the inner loop is executed exactly $j - 1$ times, so

$$T(n) = \sum_{j=2}^n (j - 1)$$

$T(n)$ is the *arithmetic series* $\sum_{k=1}^{n-1} k$, so

$$T(n) = \frac{n(n - 1)}{2}$$

$$T(n) = \Theta(n^2)$$

- Best-case is $T(n) = \Theta(n)$
- Average-case is $T(n) = \Theta(n^2)$

- Does **INSERTION-SORT** terminate for all valid inputs?

- Does **INSERTION-SORT** terminate for all valid inputs?
- If so, does it satisfy the conditions of the sorting problem?
 - ▶ A contains a *permutation* of the initial value of A
 - ▶ A is sorted: $A[1] \leq A[2] \leq \dots \leq A[\text{length}(A)]$

- Does **INSERTION-SORT** terminate for all valid inputs?
- If so, does it satisfy the conditions of the sorting problem?
 - ▶ A contains a *permutation* of the initial value of A
 - ▶ A is sorted: $A[1] \leq A[2] \leq \dots \leq A[\text{length}(A)]$
- We want ***a formal proof of correctness***
 - ▶ does not seem straightforward...

The Logic of Algorithmic Steps

Example 1: (straight-line program)

BIGGER(n)

1 // must return a value greater than n

2 $m = n * n + 1$

3 **return** m

Example 1: (straight-line program)

BIGGER(n)

```
1 // must return a value greater than n
2  $m = n * n + 1$ 
3 return  $m$ 
```

Example 2: (branching)

SortTwo(A)

```
1 // must sort (in-place) an array of 2 elements
2 if  $A[1] > A[2]$ 
3      $t = A[1]$ 
4      $A[1] = A[2]$ 
5      $A[2] = t$ 
```

Example 3: (nested branching)

MAXTHREE(a, b, c)

```
1 // find the maximum value in an array of 3 elements
2 if  $a > b$ 
3     if  $b > c$ 
4         return  $a$ 
5     else return  $c$ 
6 else if  $c > b$ 
7     return  $c$ 
8     else return  $b$ 
```

Example 3: (nested branching)

```
MAXTHREE(a, b, c)  
1 // find the maximum value in an array of 3 elements  
2 if a > b  
3     if b > c  
4         return a  
5     else return c  
6 else if c > b  
7     return c  
8     else return b
```

Is this algorithm correct?

Example 3: (nested branching)

```
MAXTHREE(a, b, c)  
1 // find the maximum value in an array of 3 elements  
2 if a > b  
3     if b > c  
4         return a  
5     else return c  
6 else if c > b  
7     return c  
8     else return b
```

Is this algorithm correct?

Why?

Example 4: (second variant)

MAXTHREE(a, b, c)

```
1 // find the maximum value in an array of 3 elements
2 if  $a > b$ 
3     if  $a > c$ 
4         return  $a$ 
5     else return  $c$ 
6 else if  $b > c$ 
7     return  $b$ 
8     else return  $c$ 
```

Example 4: (second variant)

```
MAXTHREE(a, b, c)
```

```
1 // find the maximum value in an array of 3 elements
```

```
2 if a > b
```

```
3     if a > c
```

```
4         return a
```

```
5     else return c
```

```
6 else if b > c
```

```
7     return b
```

```
8     else return c
```

Is this algorithm correct?

Example 4: (second variant)

```
MAXTHREE(a, b, c)  
1 // find the maximum value in an array of 3 elements  
2 if a > b  
3     if a > c  
4         return a  
5     else return c  
6 else if b > c  
7     return b  
8     else return c
```

Is this algorithm correct?

Prove it!

Example 5: (third variant)

```
MAXTHREE(a, b, c)
```

```
1 // find the maximum among 3 values
```

```
2 if a > b and a > c
```

```
3     return a
```

```
4 if b > c
```

```
5     return b
```

```
6 else return c
```

Example 5: (third variant)

```
MAXTHREE(a, b, c)  
1 // find the maximum among 3 values  
2 if a > b and a > c  
3     return a  
4 if b > c  
5     return b  
6 else return c
```

Is this algorithm correct?

Example 5: (third variant)

```
MAXTHREE(a, b, c)  
1 // find the maximum among 3 values  
2 if a > b and a > c  
3     return a  
4 if b > c  
5     return b  
6 else return c
```

Is this algorithm correct?

Prove it!

- The execution of a program amounts to the execution of a *sequence of instructions* we call it a ***path***

- The execution of a program amounts to the execution of a *sequence of instructions* we call it a ***path***
 - The path is determined by the evaluations of the various conditions tested in branch instructions along the path
- path condition:*** logical conjunction of all the branch conditions taken along the path

- The execution of a program amounts to the execution of a *sequence of instructions* we call it a ***path***
- The path is determined by the evaluations of the various conditions tested in branch instructions along the path
path condition: logical conjunction of all the branch conditions taken along the path
- The path condition of every path must imply the correctness condition

- The execution of a program amounts to the execution of a *sequence of instructions* we call it a ***path***
- The path is determined by the evaluations of the various conditions tested in branch instructions along the path
path condition: logical conjunction of all the branch conditions taken along the path
- The path condition of every path must imply the correctness condition

An algorithm must be correct for every possible execution path

- The execution of a program amounts to the execution of a *sequence of instructions* we call it a ***path***
- The path is determined by the evaluations of the various conditions tested in branch instructions along the path
path condition: logical conjunction of all the branch conditions taken along the path
- The path condition of every path must imply the correctness condition

An algorithm must be correct for every possible execution path

Problem: what happens when we have ***loops***?

Loop Invariants

- We formulate a *loop-invariant* condition C
 - ▶ C must remain true *through* the loop

- We formulate a *loop-invariant* condition C
 - ▶ C must remain true *through* the loop
 - ▶ C is relevant to the problem definition: we use C at the end of the loop to prove the correctness of the result

- We formulate a *loop-invariant* condition C
 - ▶ C must remain true *through* the loop
 - ▶ C is relevant to the problem definition: we use C at the end of the loop to prove the correctness of the result

- Then, we only need to prove that the algorithm terminates

Loop Invariants (2)

- Formulation: this is where we try to be smart
 - ▶ *the invariant must reflect the structure of the algorithm*
 - ▶ it must be the basis to prove the correctness of the solution

- Formulation: this is where we try to be smart
 - ▶ *the invariant must reflect the structure of the algorithm*
 - ▶ it must be the basis to prove the correctness of the solution
- Proof of validity (i.e., that C is indeed a loop invariant): typical *proof by induction*
 - ▶ **initialization:** we must prove that *the invariant C is true before entering the loop*
 - ▶ **maintenance:** we must prove that *if C is true at the beginning of a cycle **then** it remains true after one cycle*

Loop Invariant for INSERTION-SORT

INSERTION-SORT(A)

```
1  for  $i = 2$  to  $\text{length}(A)$ 
2       $j = i$ 
3      while  $j > 1$  and  $A[j - 1] > A[j]$ 
4          swap  $A[j]$  and  $A[j - 1]$ 
5           $j = j - 1$ 
```

Loop Invariant for INSERTION-SORT

INSERTION-SORT(A)

```
1  for  $i = 2$  to  $\text{length}(A)$ 
2       $j = i$ 
3      while  $j > 1$  and  $A[j - 1] > A[j]$ 
4          swap  $A[j]$  and  $A[j - 1]$ 
5           $j = j - 1$ 
```

- The main idea is to insert $A[i]$ in $A[1 \dots i - 1]$ so as to maintain a *sorted subsequence* $A[1 \dots i]$

Loop Invariant for INSERTION-SORT

```
INSERTION-SORT(A)
1  for  $i = 2$  to  $length(A)$ 
2       $j = i$ 
3      while  $j > 1$  and  $A[j - 1] > A[j]$ 
4          swap  $A[j]$  and  $A[j - 1]$ 
5           $j = j - 1$ 
```

- The main idea is to insert $A[i]$ in $A[1..i-1]$ so as to maintain a *sorted subsequence* $A[1..i]$
- **Invariant:** (outer loop) *the subarray $A[1..i-1]$ consists of the elements originally in $A[1..i-1]$ in sorted order*

Loop Invariant for INSERTION-SORT (2)

INSERTION-SORT(A)

```
1  for  $i = 2$  to  $length(A)$ 
2       $j = i$ 
3      while  $j > 1$  and  $A[j - 1] > A[j]$ 
4          swap  $A[j]$  and  $A[j - 1]$ 
5           $j = j - 1$ 
```

Loop Invariant for INSERTION-SORT (2)

```
INSERTION-SORT(A)
1  for  $i = 2$  to  $\text{length}(A)$ 
2       $j = i$ 
3      while  $j > 1$  and  $A[j - 1] > A[j]$ 
4          swap  $A[j]$  and  $A[j - 1]$ 
5           $j = j - 1$ 
```

- **Initialization:** $j = 2$, so $A[1..j - 1]$ is the single element $A[1]$
 - ▶ $A[1]$ contains the original element in $A[1]$
 - ▶ $A[1]$ is trivially sorted

Loop Invariant for INSERTION-SORT (3)

INSERTION-SORT(A)

```
1  for  $i = 2$  to  $length(A)$ 
2       $j = i$ 
3      while  $j > 1$  and  $A[j - 1] > A[j]$ 
4          swap  $A[j]$  and  $A[j - 1]$ 
5           $j = j - 1$ 
```

Loop Invariant for INSERTION-SORT (3)

```
INSERTION-SORT(A)
1  for  $i = 2$  to  $\text{length}(A)$ 
2       $j = i$ 
3      while  $j > 1$  and  $A[j - 1] > A[j]$ 
4          swap  $A[j]$  and  $A[j - 1]$ 
5           $j = j - 1$ 
```

- **Maintenance:** informally, if $A[1..i-1]$ is a permutation of the original $A[1..i-1]$ and $A[1..i-1]$ is sorted (invariant), then *if* we enter the inner loop:
 - ▶ shifts the subarray $A[k..i-1]$ by one position to the right
 - ▶ inserts *key*, which was originally in $A[i]$ at its proper position $1 \leq k \leq i-1$, in sorted order

Loop Invariant for INSERTION-SORT (4)

INSERTION-SORT(A)

```
1  for  $i = 2$  to  $length(A)$ 
2       $j = i$ 
3      while  $j > 1$  and  $A[j - 1] > A[j]$ 
4          swap  $A[j]$  and  $A[j - 1]$ 
5           $j = j - 1$ 
```

Loop Invariant for INSERTION-SORT (4)

INSERTION-SORT(A)

```
1  for  $i = 2$  to  $length(A)$ 
2       $j = i$ 
3      while  $j > 1$  and  $A[j - 1] > A[j]$ 
4          swap  $A[j]$  and  $A[j - 1]$ 
5           $j = j - 1$ 
```

- **Termination:** INSERTION-SORT terminates with $i = length(A) + 1$; the invariant states that

Loop Invariant for INSERTION-SORT (4)

```
INSERTION-SORT(A)
1  for  $i = 2$  to  $length(A)$ 
2       $j = i$ 
3      while  $j > 1$  and  $A[j - 1] > A[j]$ 
4          swap  $A[j]$  and  $A[j - 1]$ 
5           $j = j - 1$ 
```

- **Termination:** **INSERTION-SORT** terminates with $i = length(A) + 1$; the invariant states that
 - ▶ $A[1..i-1]$ is a permutation of the original $A[1..i-1]$
 - ▶ $A[1..i-1]$ is sorted

Given the termination condition, $A[1..i-1]$ is the whole A

So **INSERTION-SORT** is *correct*!

- You are given a problem P and an algorithm A
 - ▶ P formally defines a *correctness* condition
 - ▶ assume, for simplicity, that A consists of one loop
-

- You are given a problem P and an algorithm A
 - ▶ P formally defines a *correctness* condition
 - ▶ assume, for simplicity, that A consists of one loop
-

1. Formulate an invariant C

- You are given a problem P and an algorithm A
 - ▶ P formally defines a *correctness* condition
 - ▶ assume, for simplicity, that A consists of one loop
-

1. Formulate an invariant C

2. **Initialization**

(for all valid inputs)

- ▶ prove that C holds right before the first execution of the first instruction of the loop

- You are given a problem P and an algorithm A
 - ▶ P formally defines a *correctness* condition
 - ▶ assume, for simplicity, that A consists of one loop
-

1. Formulate an invariant C

2. **Initialization** (for all valid inputs)

- ▶ prove that C holds right before the first execution of the first instruction of the loop

3. **Management** (for all valid inputs)

- ▶ prove that if C holds right before the first instruction of the loop, then it holds also at the end of the loop

- You are given a problem P and an algorithm A
 - ▶ P formally defines a *correctness* condition
 - ▶ assume, for simplicity, that A consists of one loop
-

1. Formulate an invariant C

2. **Initialization** (for all valid inputs)

- ▶ prove that C holds right before the first execution of the first instruction of the loop

3. **Management** (for all valid inputs)

- ▶ prove that if C holds right before the first instruction of the loop, then it holds also at the end of the loop

4. **Termination** (for all valid inputs)

- ▶ prove that the loop terminates, with some exit condition X

- You are given a problem P and an algorithm A
 - ▶ P formally defines a *correctness* condition
 - ▶ assume, for simplicity, that A consists of one loop
-

1. Formulate an invariant C

2. **Initialization** (for all valid inputs)

- ▶ prove that C holds right before the first execution of the first instruction of the loop

3. **Management** (for all valid inputs)

- ▶ prove that if C holds right before the first instruction of the loop, then it holds also at the end of the loop

4. **Termination** (for all valid inputs)

- ▶ prove that the loop terminates, with some exit condition X

5. Prove that $X \wedge C \Rightarrow P$, which means that A is correct

Exercise: Analyze Selection-Sort

SELECTION-SORT(A)

```
1   $n = \text{length}(A)$ 
2  for  $i = 1$  to  $n - 1$ 
3       $\text{smallest} = i$ 
4      for  $j = i + 1$  to  $n$ 
5          if  $A[j] < A[\text{smallest}]$ 
6               $\text{smallest} = j$ 
7      swap  $A[i]$  and  $A[\text{smallest}]$ 
```

Exercise: Analyze Selection-Sort

SELECTION-SORT(A)

```
1   $n = \text{length}(A)$ 
2  for  $i = 1$  to  $n - 1$ 
3       $\text{smallest} = i$ 
4      for  $j = i + 1$  to  $n$ 
5          if  $A[j] < A[\text{smallest}]$ 
6               $\text{smallest} = j$ 
7      swap  $A[i]$  and  $A[\text{smallest}]$ 
```

■ Correctness?

- ▶ loop invariant?

■ Complexity?

- ▶ worst, best, and average case?

Exercise: Analyze Bubblesort

BUBBLESORT(A)

```
1  for  $i = 1$  to  $\text{length}(A)$   
2      for  $j = \text{length}(A)$  downto  $i + 1$   
3          if  $A[j] < A[j - 1]$   
4              swap  $A[j]$  and  $A[j - 1]$ 
```

Exercise: Analyze Bubblesort

BUBBLESORT(A)

```
1  for  $i = 1$  to  $length(A)$ 
2      for  $j = length(A)$  downto  $i + 1$ 
3          if  $A[j] < A[j - 1]$ 
4              swap  $A[j]$  and  $A[j - 1]$ 
```

■ Correctness?

- ▶ loop invariant?

■ Complexity?

- ▶ worst, best, and average case?