# Advanced Algorithmic Techniques (COMP523)

Introduction to algorithms and basic complexity notions

# Algorithm

- ***A set of instructions for solving a problem or performing a computation.***

- **Origin of the name:** Latinisation of the name given by Persian scholar Muhammad ibn Musa al-Khwarizmi.

كتاب الخوارزمي

# Example: Searching

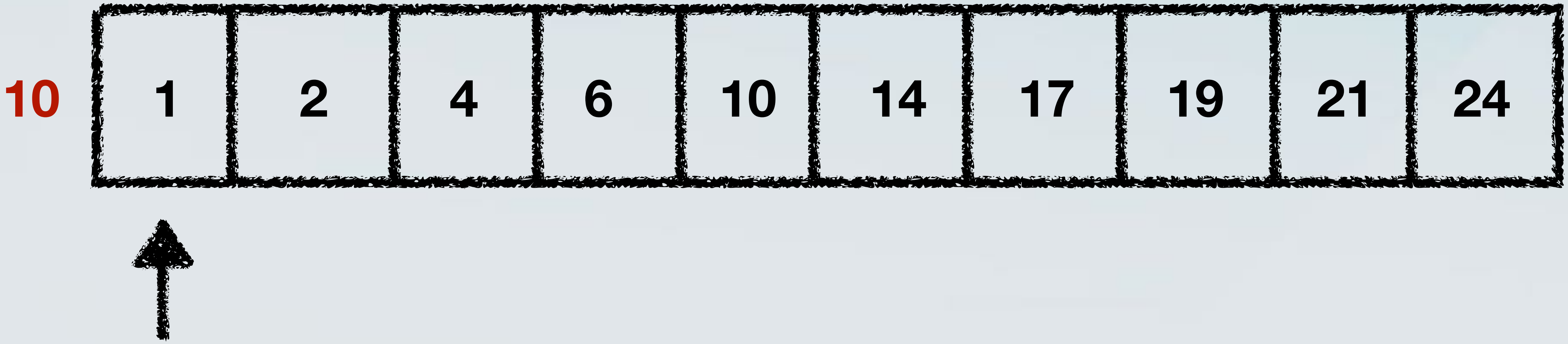
- Find if a number **x** exists in an **array** of **sorted numbers**.

**10**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 4 | 6 | 10 | 14 | 17 | 19 | 21 | 24 |

# Example: Searching

- Find if a number **x** exists in an **array** of **sorted numbers**.

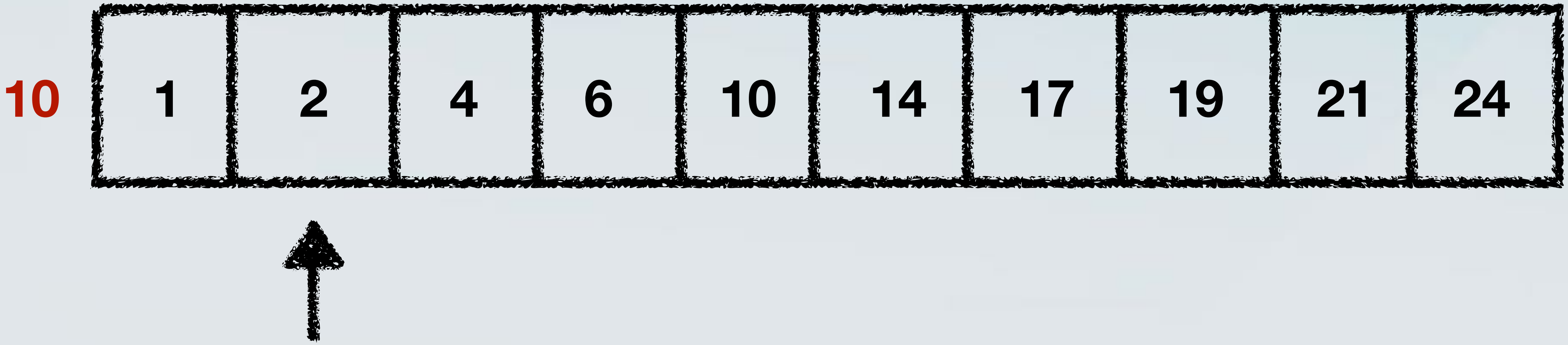


**10**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 4 | 6 | 10 | 14 | 17 | 19 | 21 | 24 |

# Example: Searching

- Find if a number **x** exists in an **array** of **sorted numbers**.

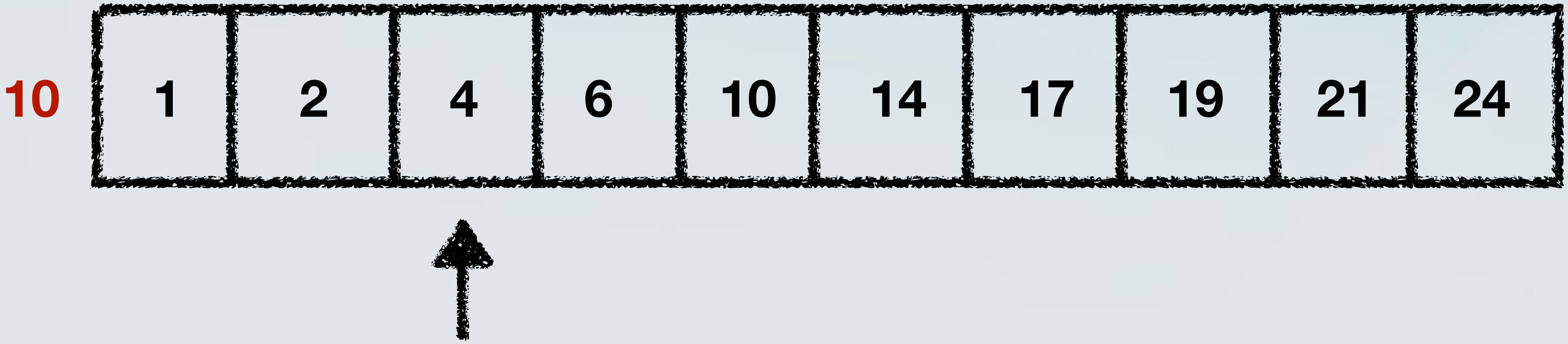


10

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 4 | 6 | 10 | 14 | 17 | 19 | 21 | 24 |

# Example: Searching

- Find if a number **x** exists in an **array** of **sorted numbers**.

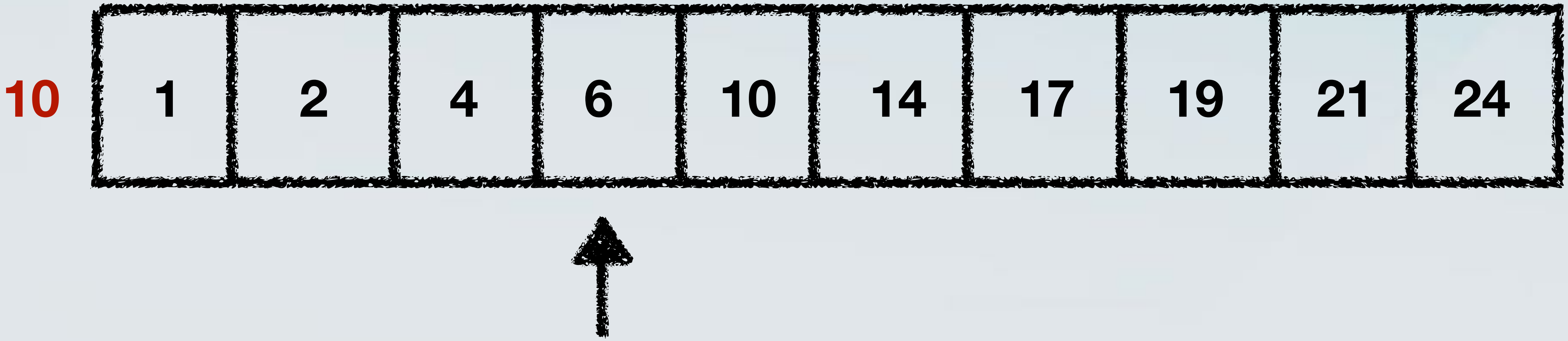


10

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 4 | 6 | 10 | 14 | 17 | 19 | 21 | 24 |

# Example: Searching

- Find if a number **x** exists in an **array** of **sorted numbers**.

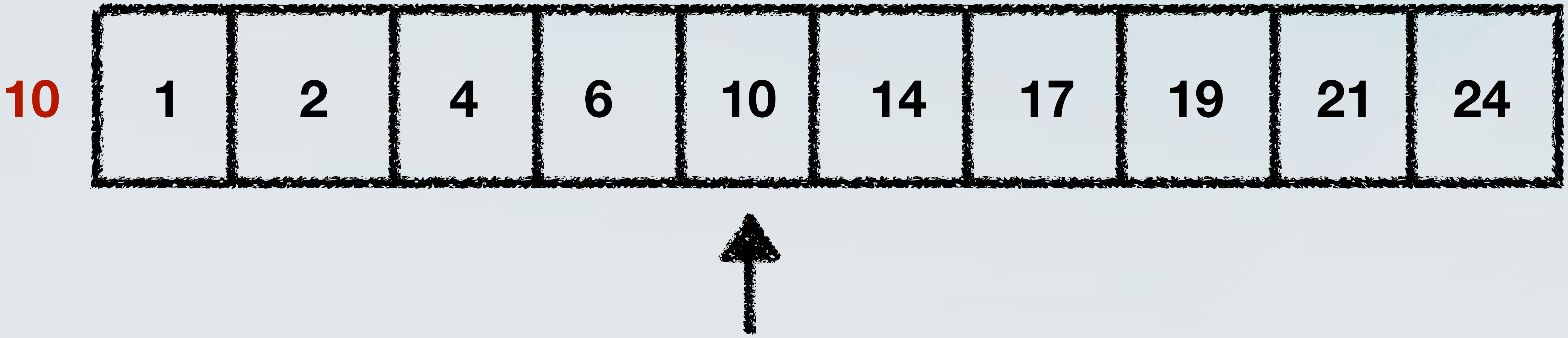


**10**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 4 | 6 | 10 | 14 | 17 | 19 | 21 | 24 |

# Example: Searching

- Find if a number **x** exists in an **array** of **sorted numbers**.

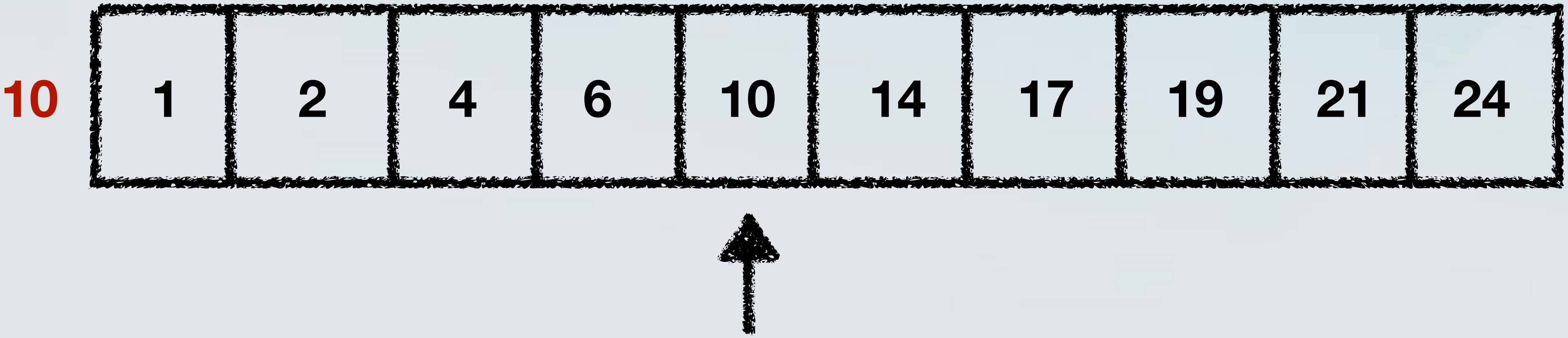


10

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 4 | 6 | 10 | 14 | 17 | 19 | 21 | 24 |

# Example: Searching

- Find if a number **x** exists in an **array** of **sorted numbers**.

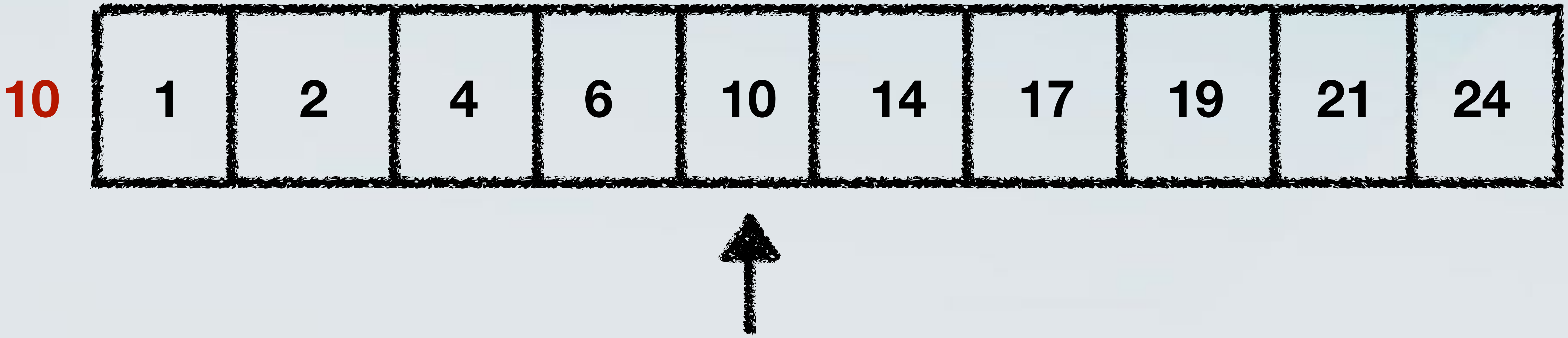


10

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 4 | 6 | 10 | 14 | 17 | 19 | 21 | 24 |

# Example: Searching

- Find if a number **x** exists in an **array** of **sorted numbers**.



10

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 4 | 6 | 10 | 14 | 17 | 19 | 21 | 24 |

- Yes, the number was found in the array!

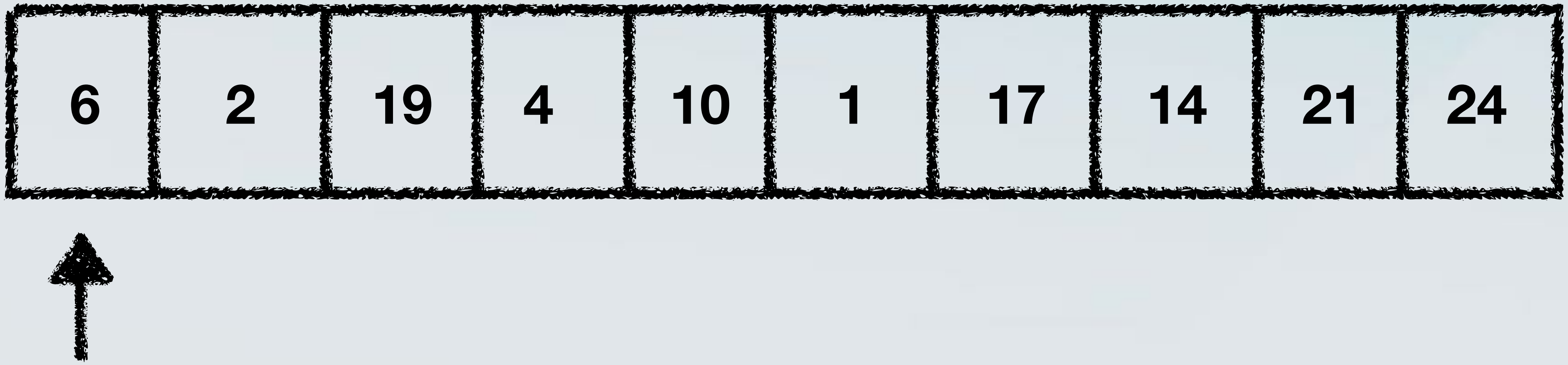
# Example: Sorting

- Given a sequence of numbers, put them in increasing order.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 6 | 2 | 19 | 4 | 10 | 1 | 17 | 14 | 21 | 24 |

# Example: Sorting

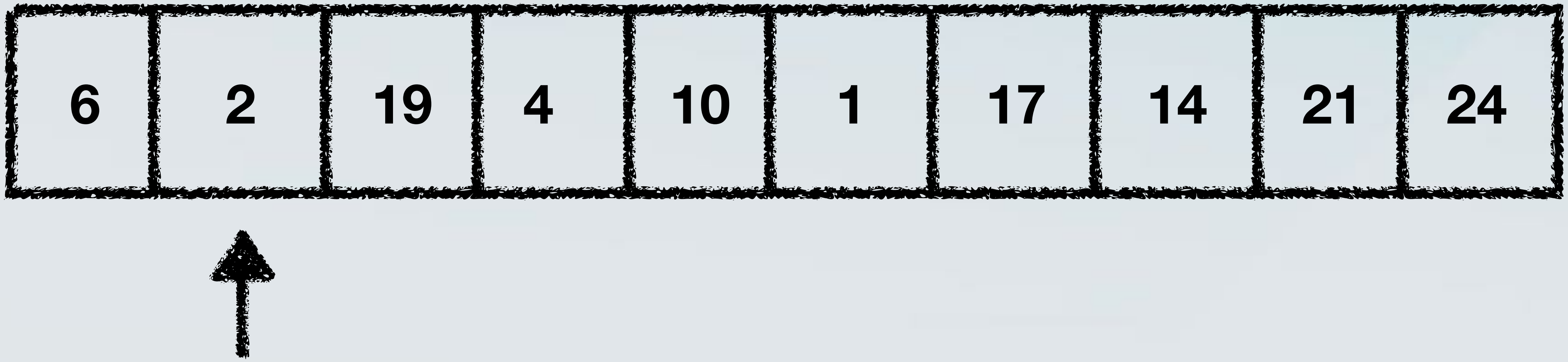
- Given a sequence of numbers, put them in increasing order.



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 6 | 2 | 19 | 4 | 10 | 1 | 17 | 14 | 21 | 24 |

# Example: Sorting

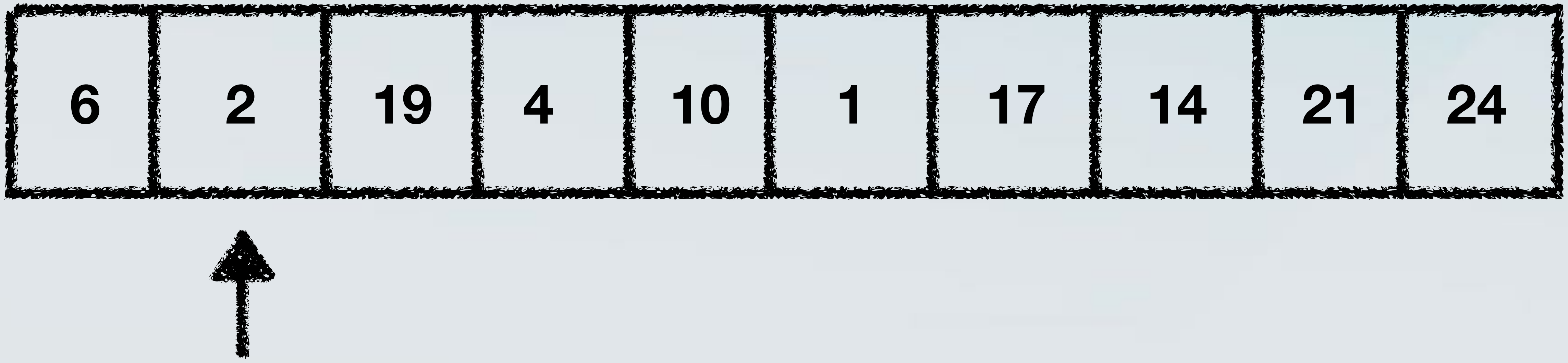
- Given a sequence of numbers, put them in increasing order.



|  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|
| 6 | 2 | 19 | 4 | 10 | 1 | 17 | 14 | 21 | 24 |

# Example: Sorting

- Given a sequence of numbers, put them in increasing order.

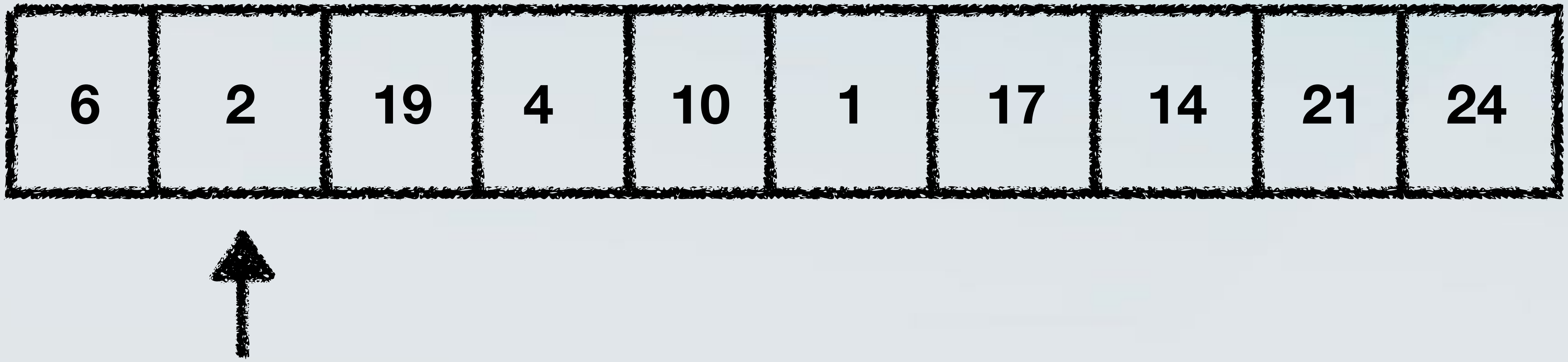


| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 6 | 2 | 19 | 4 | 10 | 1 | 17 | 14 | 21 | 24 |

**Is 2 < 6?**

# Example: Sorting

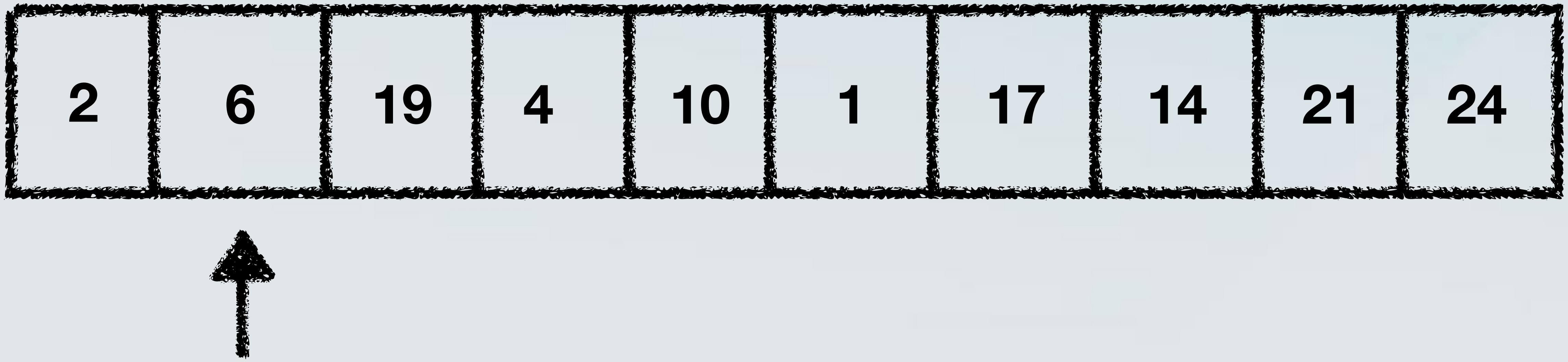
- Given a sequence of numbers, put them in increasing order.



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 6 | 2 | 19 | 4 | 10 | 1 | 17 | 14 | 21 | 24 |

# Example: Sorting

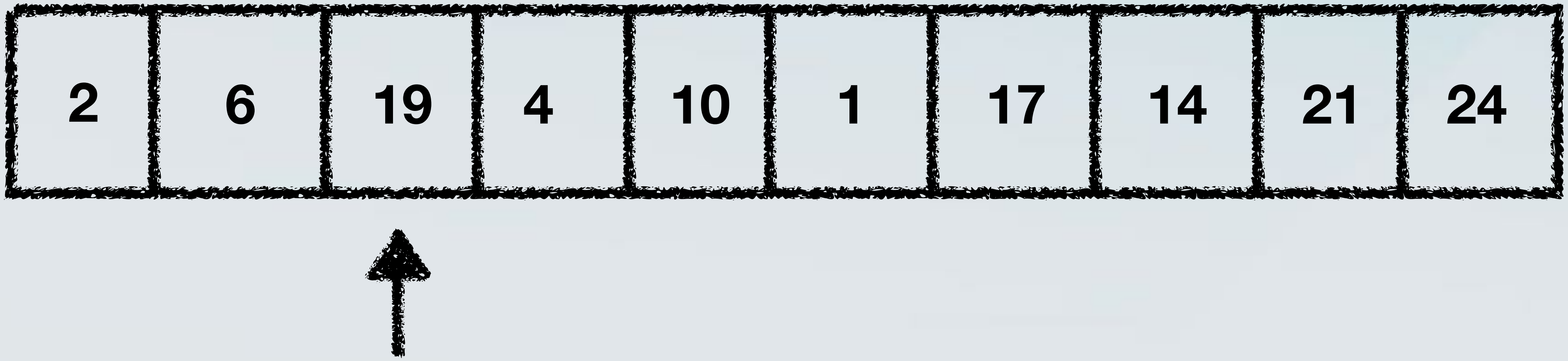
- Given a sequence of numbers, put them in increasing order.



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 6 | 19 | 4 | 10 | 1 | 17 | 14 | 21 | 24 |

# Example: Sorting

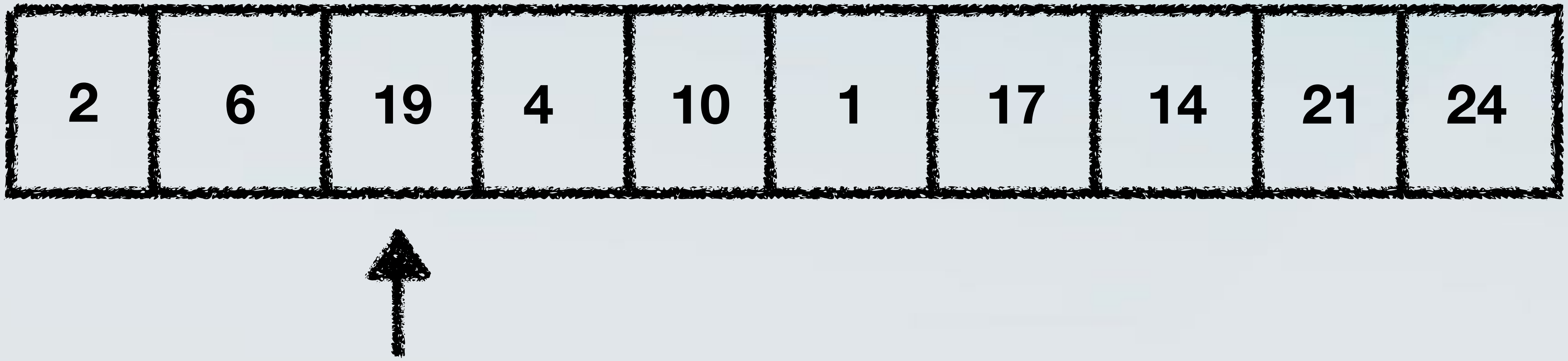
- Given a sequence of numbers, put them in increasing order.



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 6 | 19 | 4 | 10 | 1 | 17 | 14 | 21 | 24 |

# Example: Sorting

- Given a sequence of numbers, put them in increasing order.

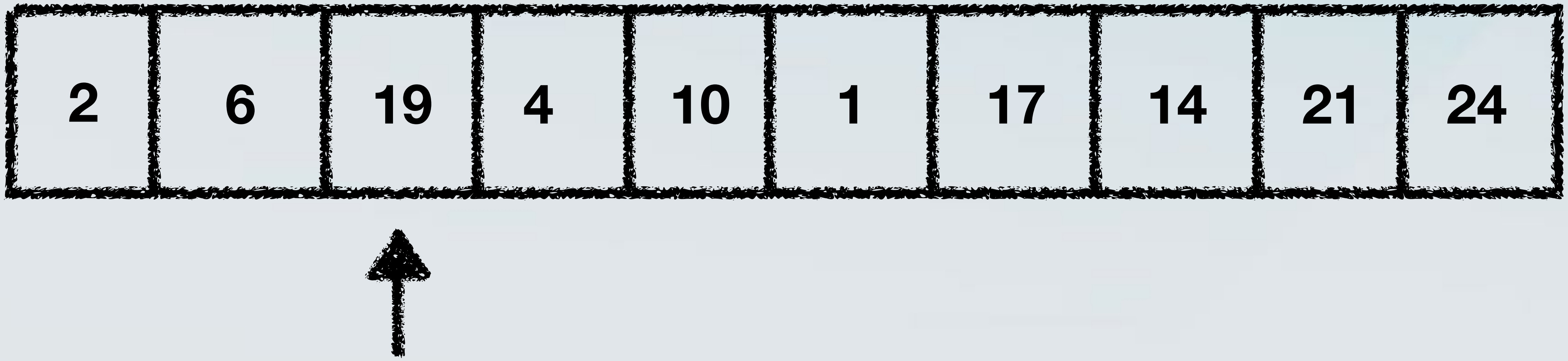


| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **2** | **6** | **19** | **4** | **10** | **1** | **17** | **14** | **21** | **24** |

**Is 19 < 6?**

# Example: Sorting

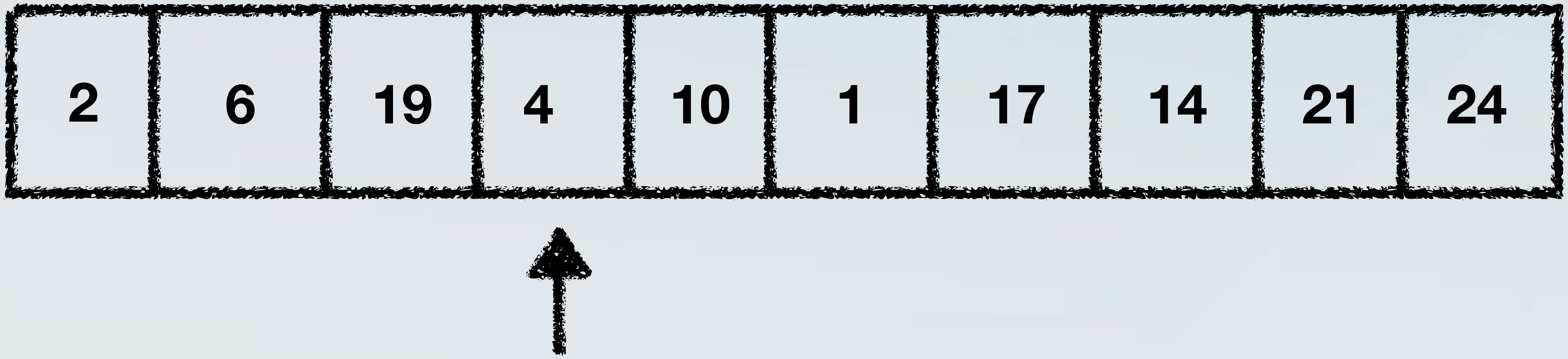
- Given a sequence of numbers, put them in increasing order.



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 6 | 19 | 4 | 10 | 1 | 17 | 14 | 21 | 24 |

# Example: Sorting

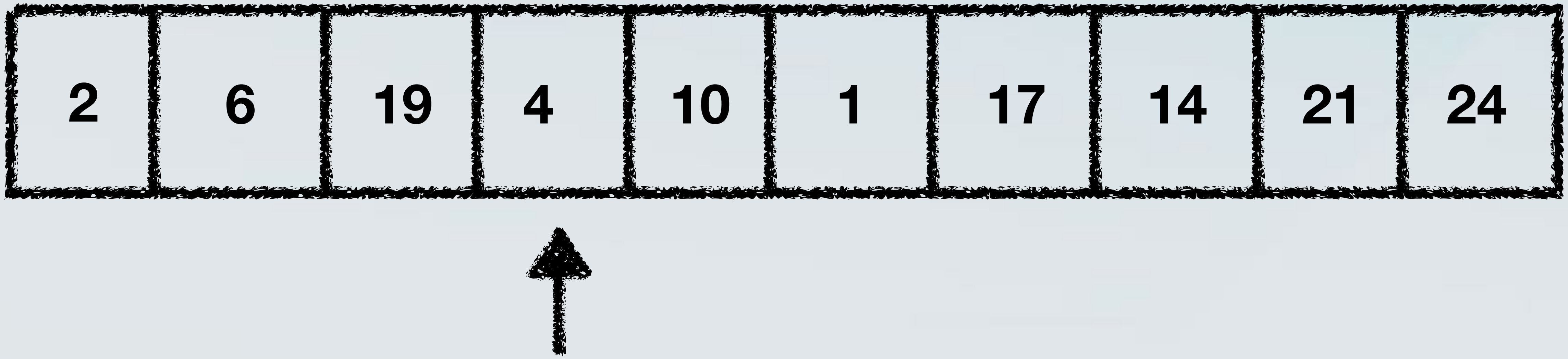
- Given a sequence of numbers, put them in increasing order.



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 6 | 19 | 4 | 10 | 1 | 17 | 14 | 21 | 24 |

# Example: Sorting

- Given a sequence of numbers, put them in increasing order.

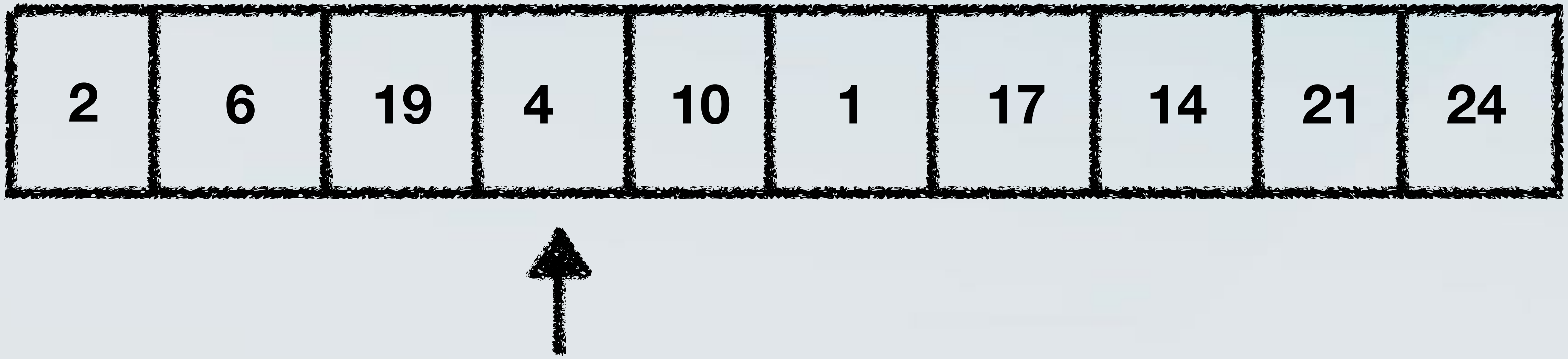


| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 6 | 19 | 4 | 10 | 1 | 17 | 14 | 21 | 24 |

**Is 4 < 19?**

# Example: Sorting

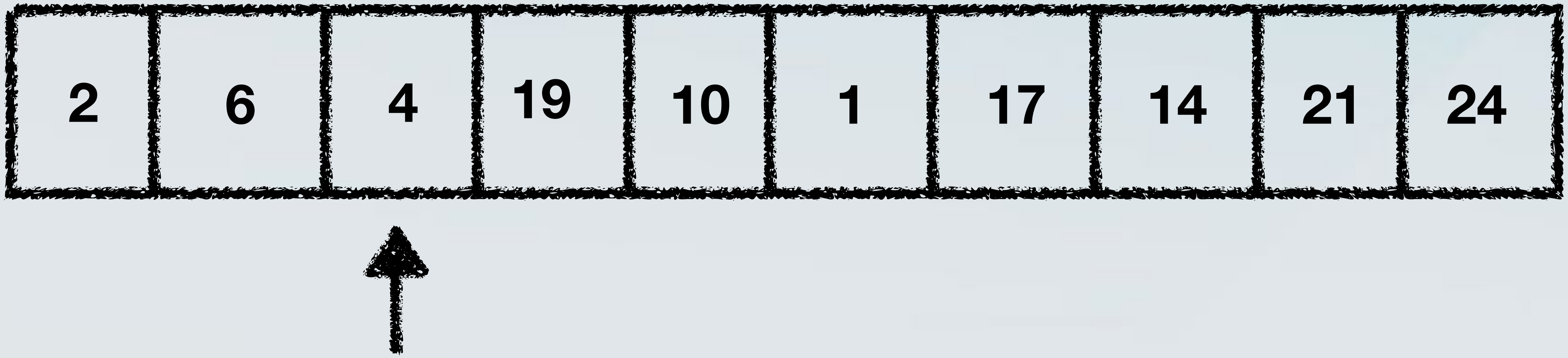
- Given a sequence of numbers, put them in increasing order.



|  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 6 | 19 | 4 | 10 | 1 | 17 | 14 | 21 | 24 |

# Example: Sorting

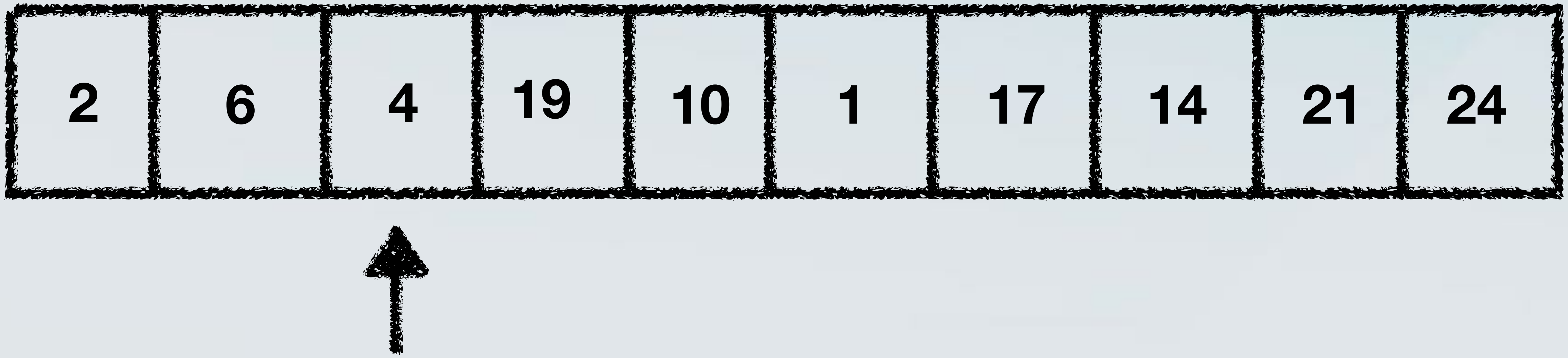
- Given a sequence of numbers, put them in increasing order.



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 6 | 4 | 19 | 10 | 1 | 17 | 14 | 21 | 24 |

# Example: Sorting

- Given a sequence of numbers, put them in increasing order.

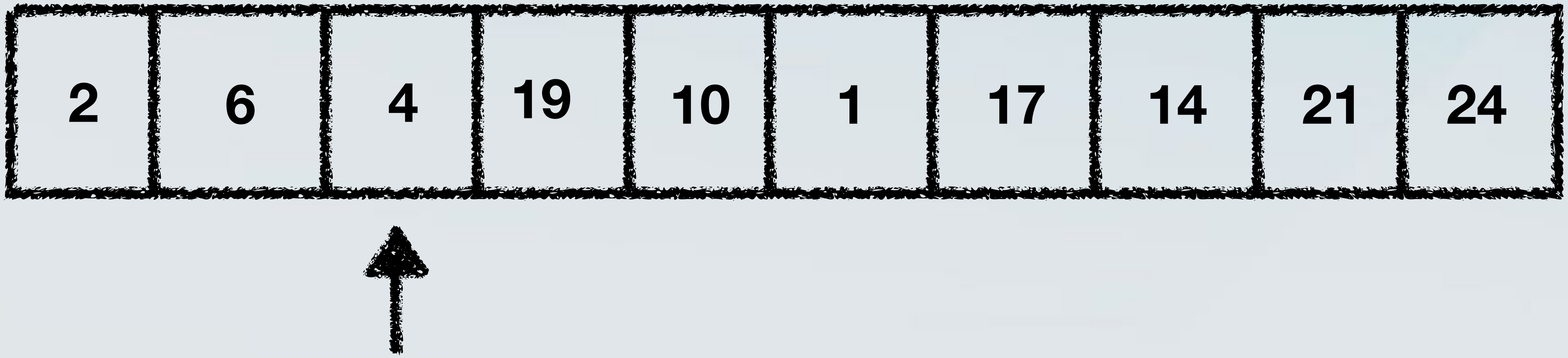


| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 6 | 4 | 19 | 10 | 1 | 17 | 14 | 21 | 24 |

**Is 4 < 6?**

# Example: Sorting

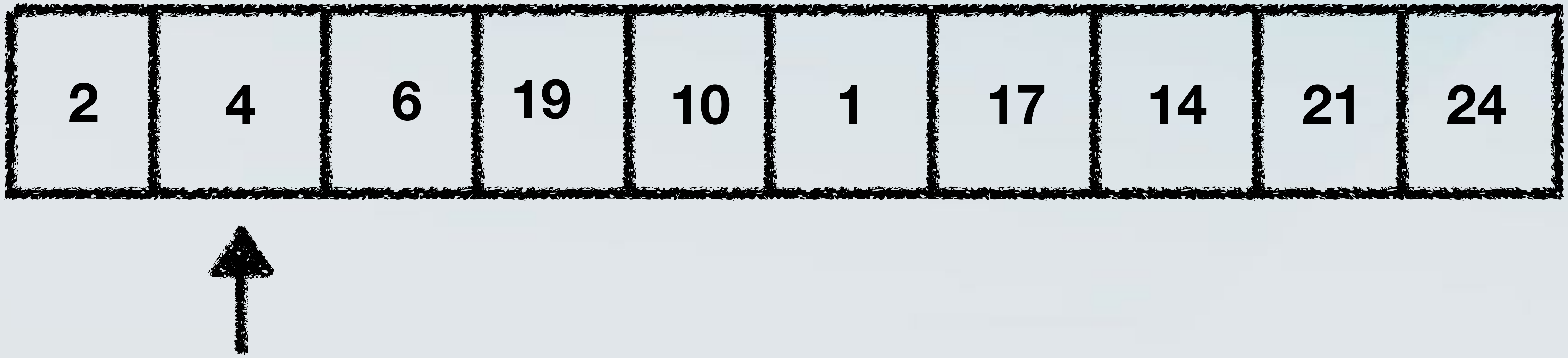
- Given a sequence of numbers, put them in increasing order.



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 6 | 4 | 19 | 10 | 1 | 17 | 14 | 21 | 24 |

# Example: Sorting

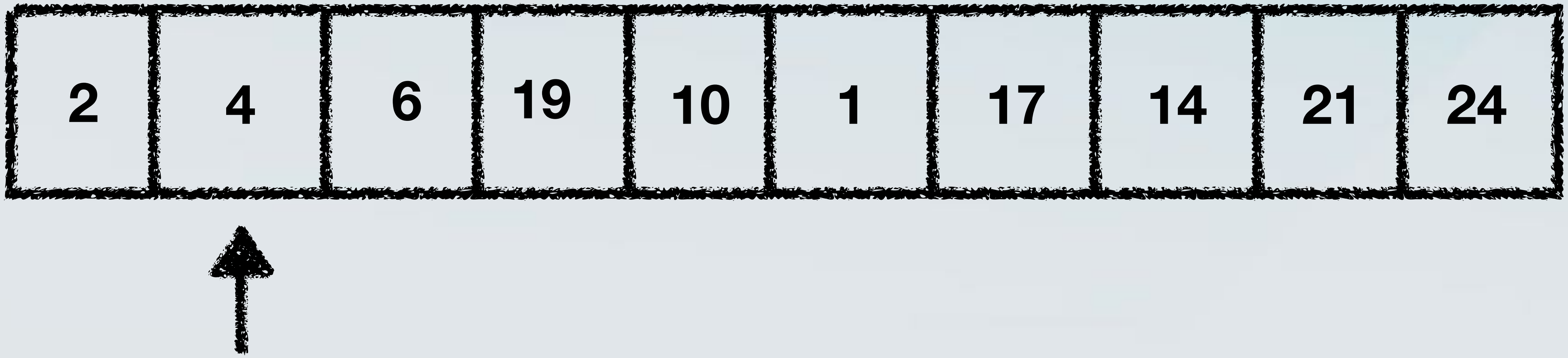
- Given a sequence of numbers, put them in increasing order.



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 4 | 6 | 19 | 10 | 1 | 17 | 14 | 21 | 24 |

# Example: Sorting

- Given a sequence of numbers, put them in increasing order.

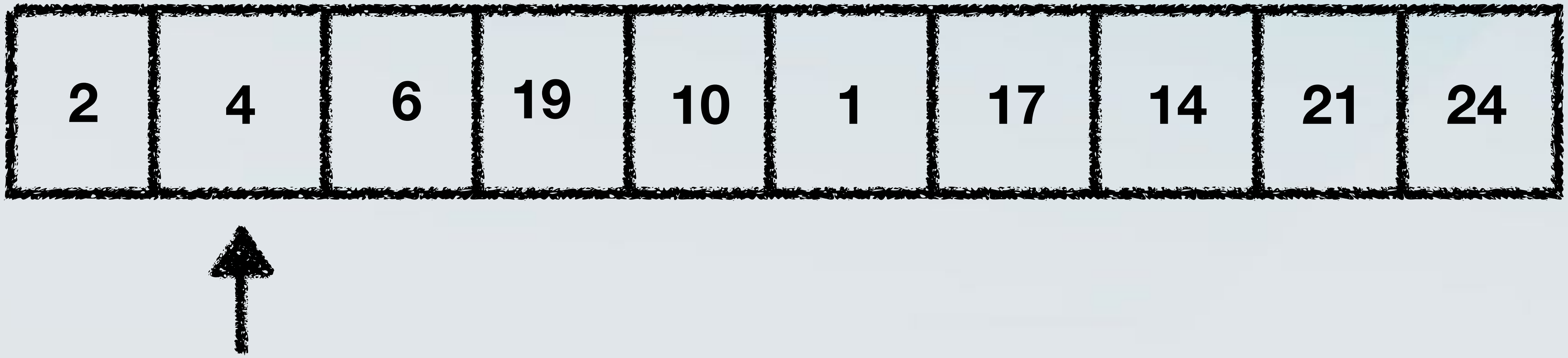


| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 4 | 6 | 19 | 10 | 1 | 17 | 14 | 21 | 24 |

**Is 4 < 2?**

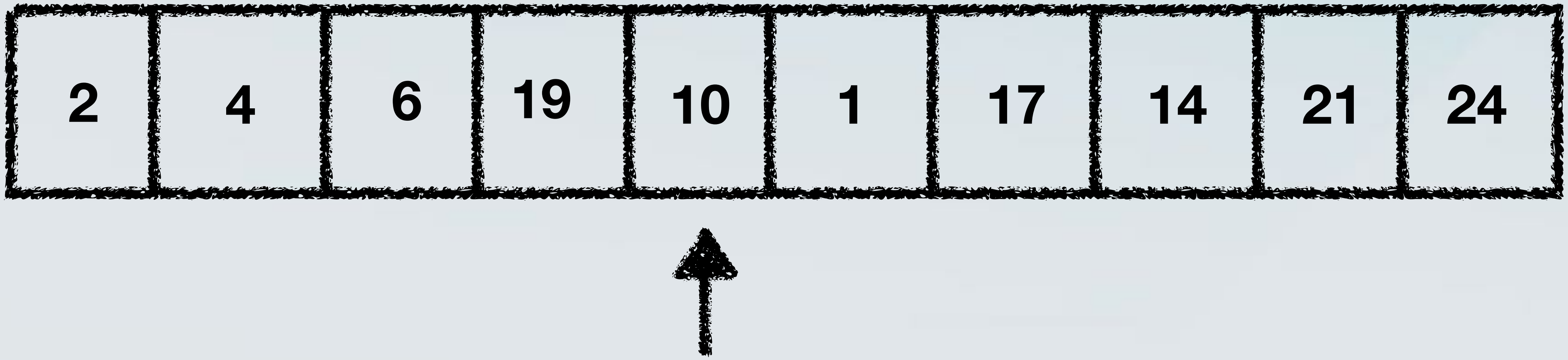
# Example: Sorting

- Given a sequence of numbers, put them in increasing order.



# Example: Sorting

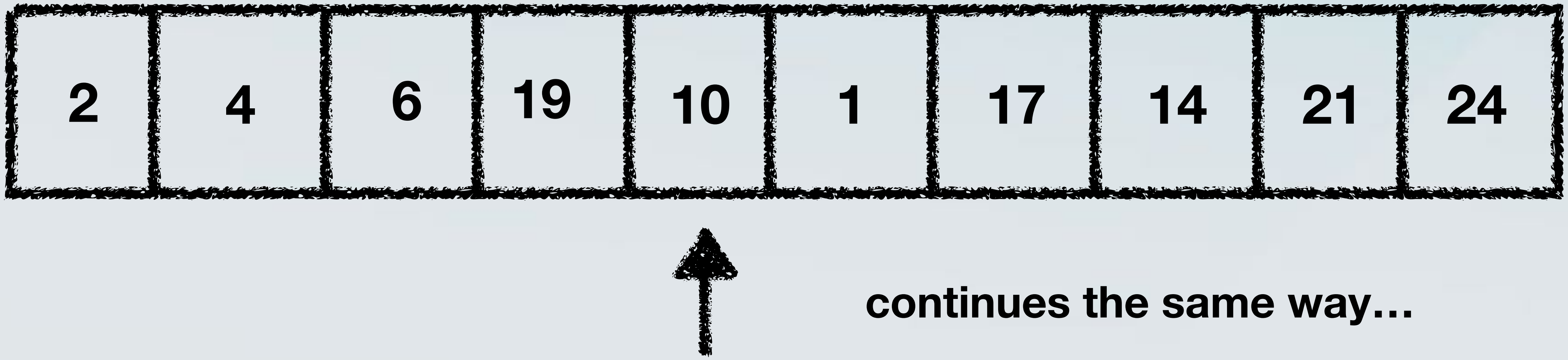
- Given a sequence of numbers, put them in increasing order.



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 4 | 6 | 19 | 10 | 1 | 17 | 14 | 21 | 24 |

# Example: Sorting

- Given a sequence of numbers, put them in increasing order.

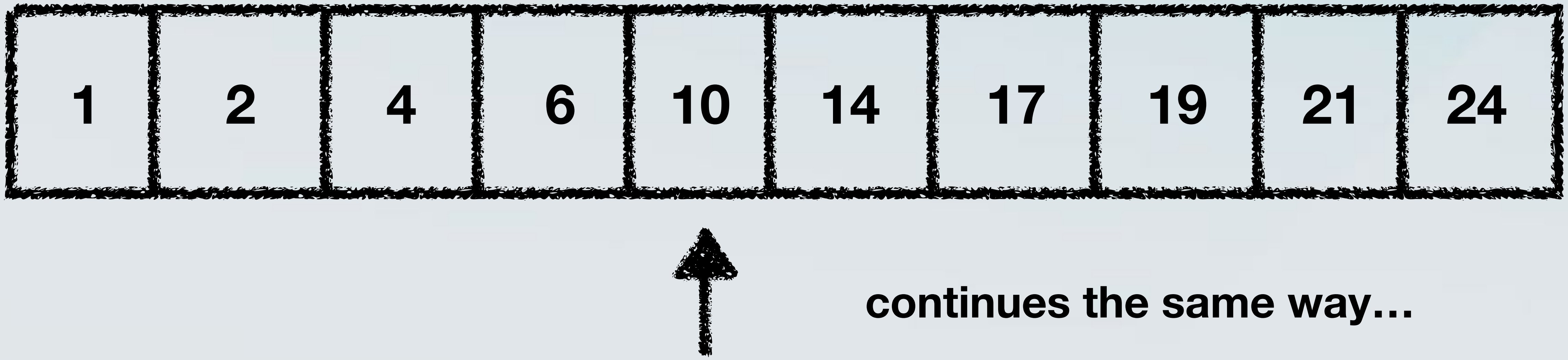


| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 4 | 6 | 19 | 10 | 1 | 17 | 14 | 21 | 24 |

**continues the same way…**

# Example: Sorting

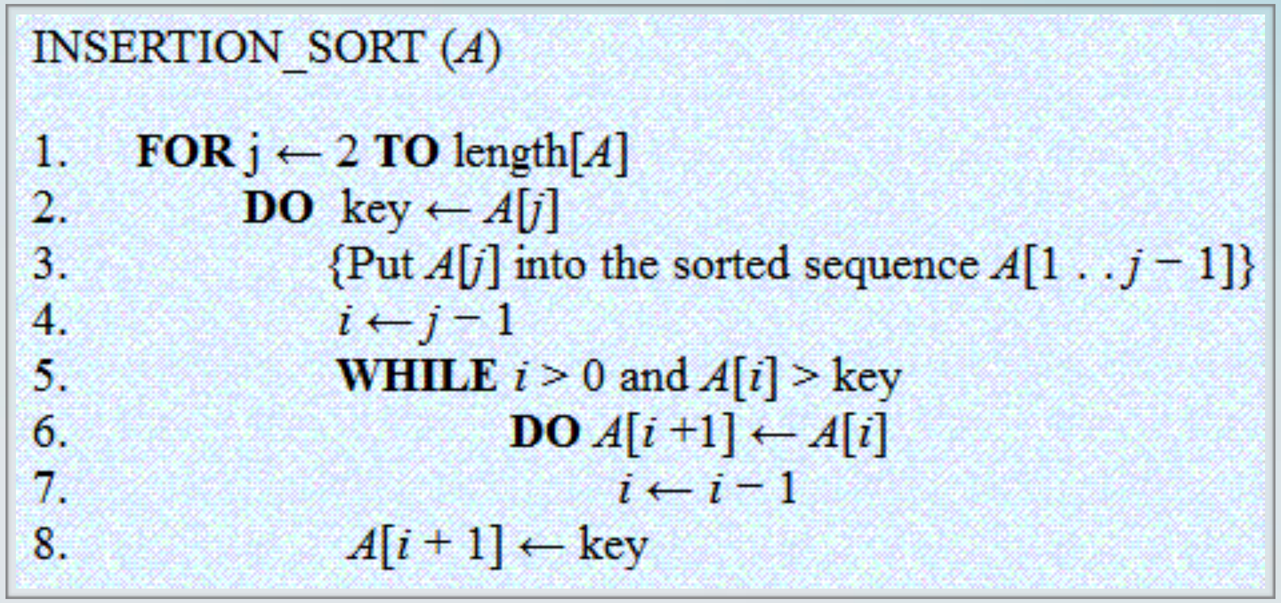
- Given a sequence of numbers, put them in increasing order.



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 4 | 6 | 10 | 14 | 17 | 19 | 21 | 24 |

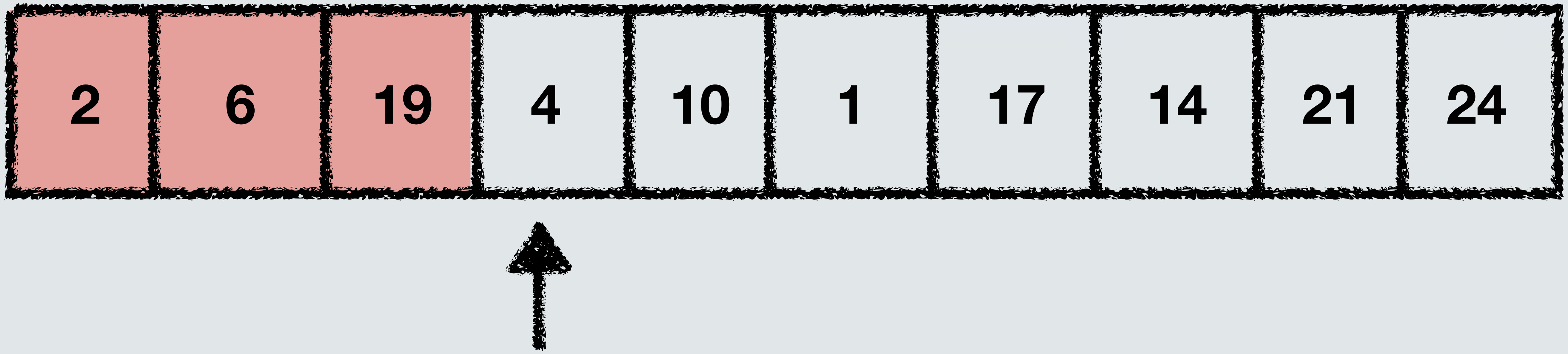
**continues the same way…**

# Describing algorithms: Pseudocode



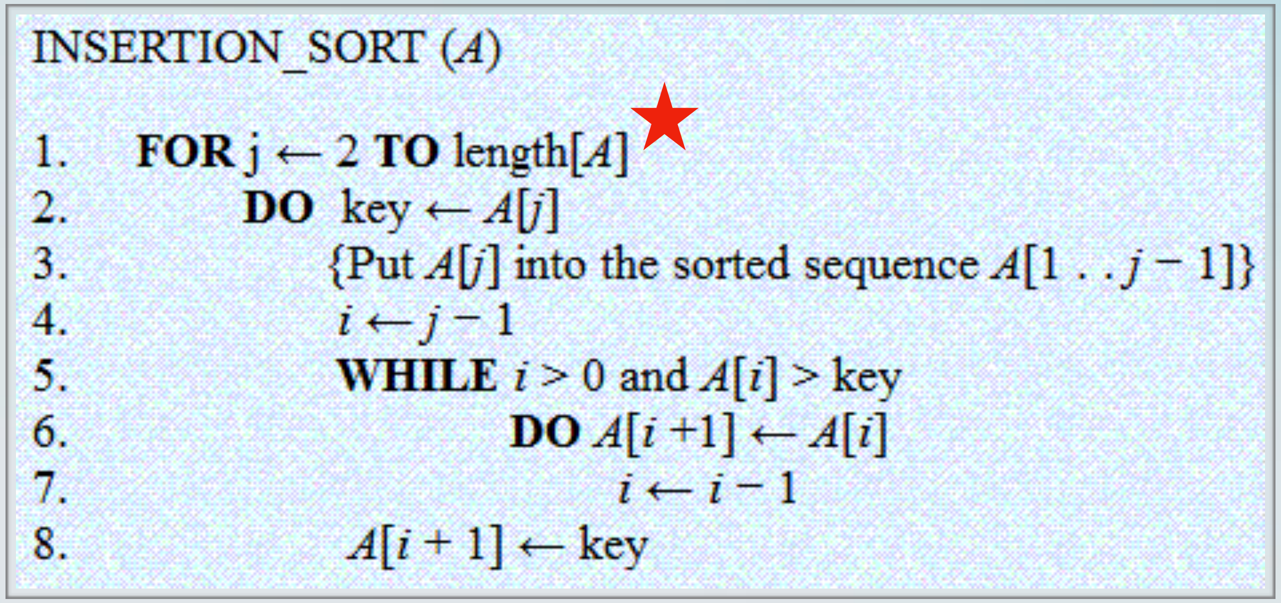
```
INSERTION_SORT ($A$)

FOR j ← 2 TO length[$A$]
    DO key ← $A[j]$
        {Put $A[j]$ into the sorted sequence $A[1 . . j - 1]$}
        $i \leftarrow j - 1$
        WHILE $i > 0$ and $A[i] >$ key
            DO $A[i+1] \leftarrow A[i]$
                $i \leftarrow i - 1$
        $A[i + 1] \leftarrow$ key
```

- The algorithm maintains a sorted array in each iteration (each time the for loop is executed).



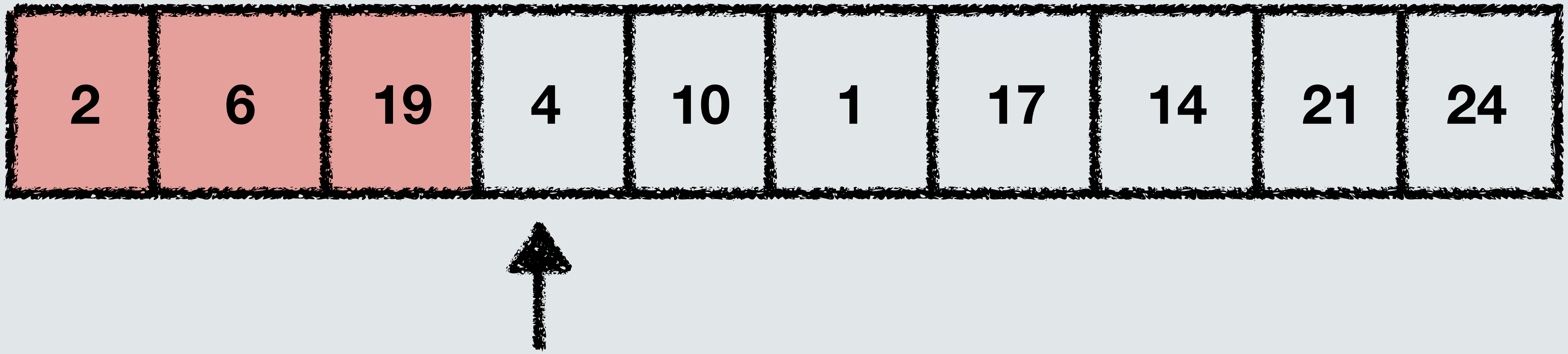
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 6 | 19 | 4 | 10 | 1 | 17 | 14 | 21 | 24 |

# Describing algorithms: Pseudocode



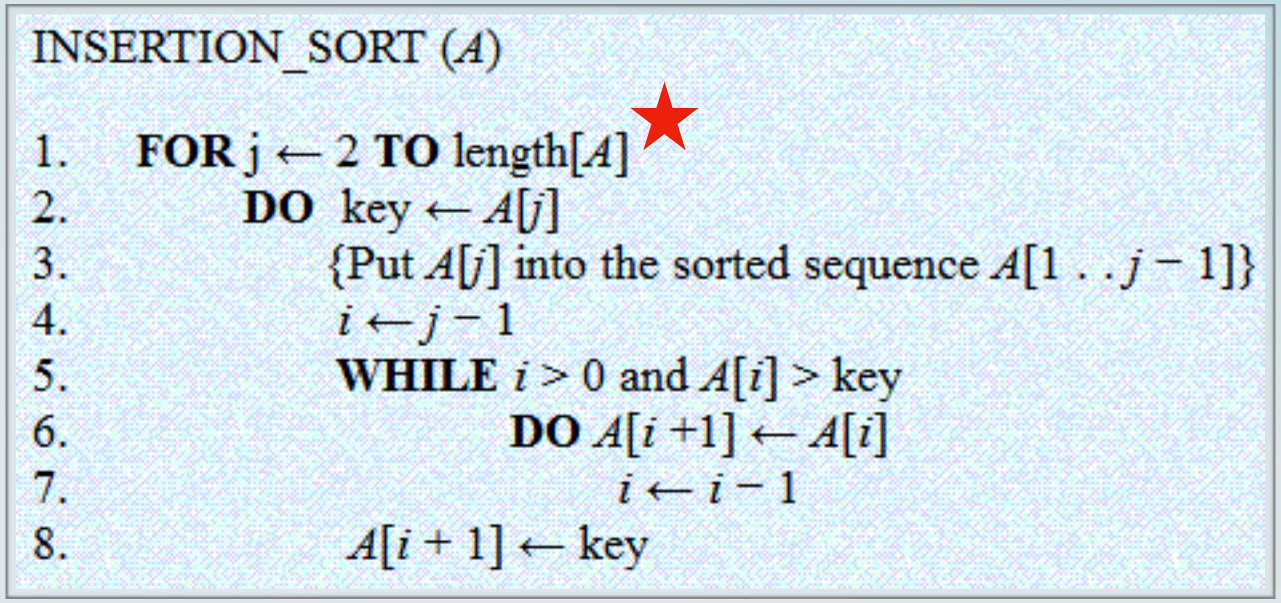
```
INSERTION_SORT (A)

FOR j ← 2 TO length[A]
        DO  key ← A[j]
             {Put A[j] into the sorted sequence A[1 . . j − 1]}
             i ← j − 1
             WHILE i > 0 and A[i] > key
                        DO A[i +1] ← A[i]
                              i ← i − 1
              A[i + 1] ← key
```

- The algorithm maintains a sorted array in each iteration (each time the for loop is executed).



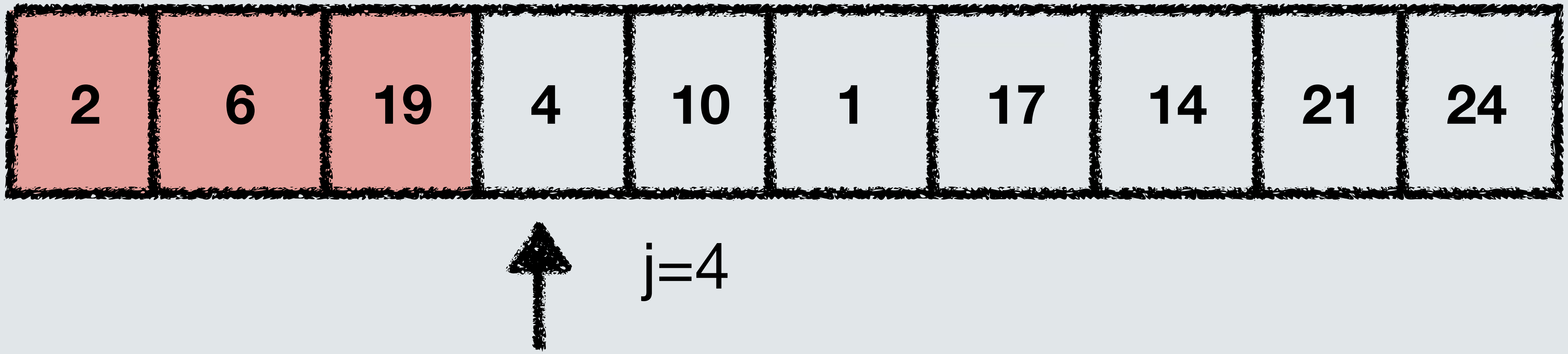
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 6 | 19 | 4 | 10 | 1 | 17 | 14 | 21 | 24 |

# Describing algorithms: Pseudocode



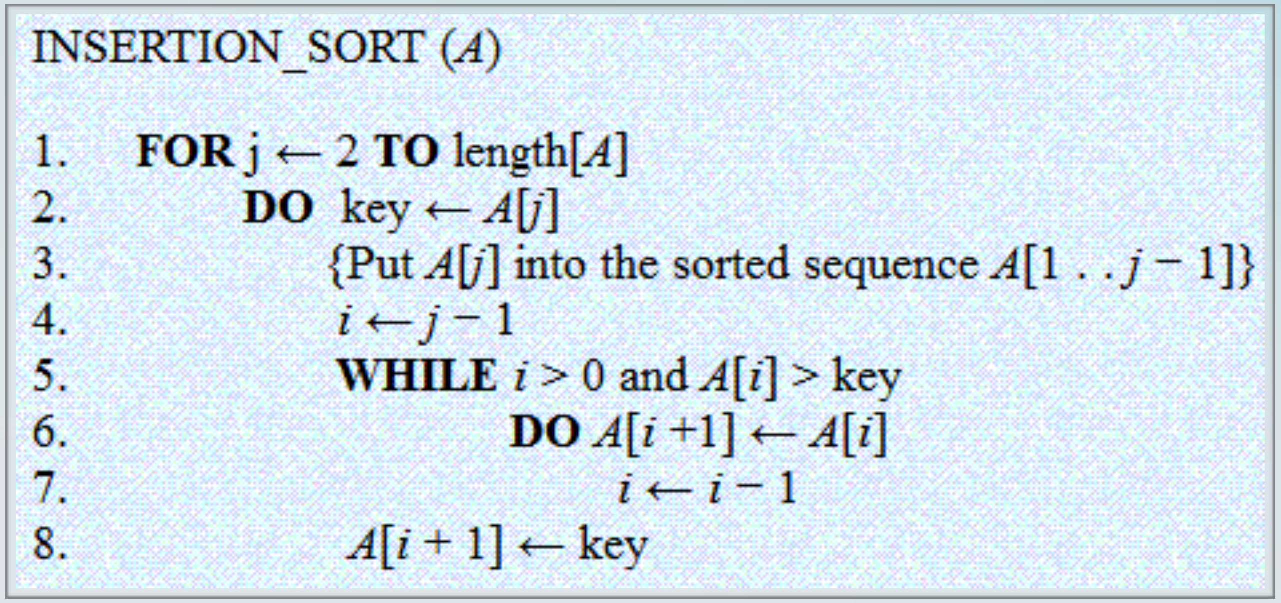
```
INSERTION_SORT ($A$)

FOR j ← 2 TO length[$A$]
    DO key ← $A[j]$
        {Put $A[j]$ into the sorted sequence $A[1..j-1]$}
        $i \leftarrow j-1$
        WHILE $i > 0$ and $A[i] >$ key
            DO $A[i+1] \leftarrow A[i]$
                $i \leftarrow i-1$
        $A[i+1] \leftarrow$ key
```

- The algorithm maintains a sorted array in each iteration (each time the for loop is executed).



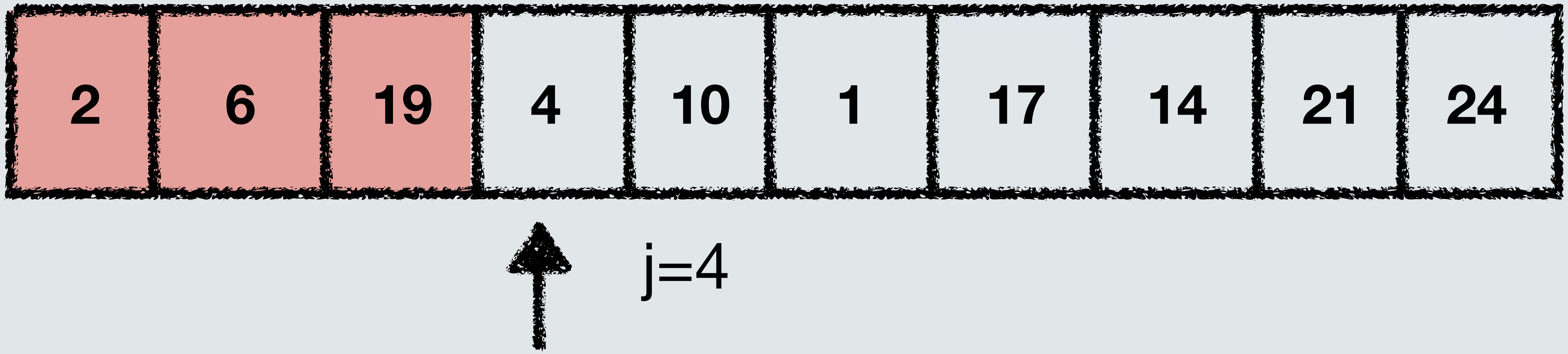
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 6 | 19 | 4 | 10 | 1 | 17 | 14 | 21 | 24 |

# Describing algorithms: Pseudocode



```
INSERTION_SORT (A)

FOR j ← 2 TO length[A]
    DO  key ← A[j]
        {Put A[j] into the sorted sequence A[1 . . j − 1]}
        i ← j − 1
        WHILE i > 0 and A[i] > key
            DO A[i +1] ← A[i]
               i ← i − 1
        A[i + 1] ← key
```

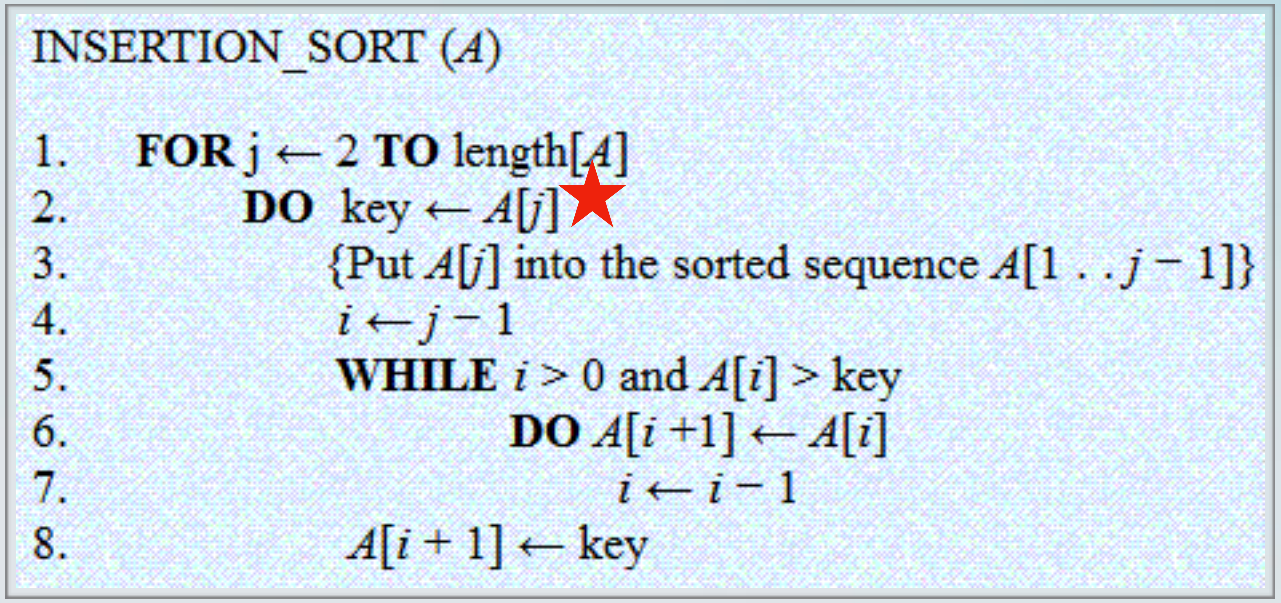
- The algorithm maintains a sorted array in each iteration (each time the for loop is executed).



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 6 | 19 | 4 | 10 | 1 | 17 | 14 | 21 | 24 |

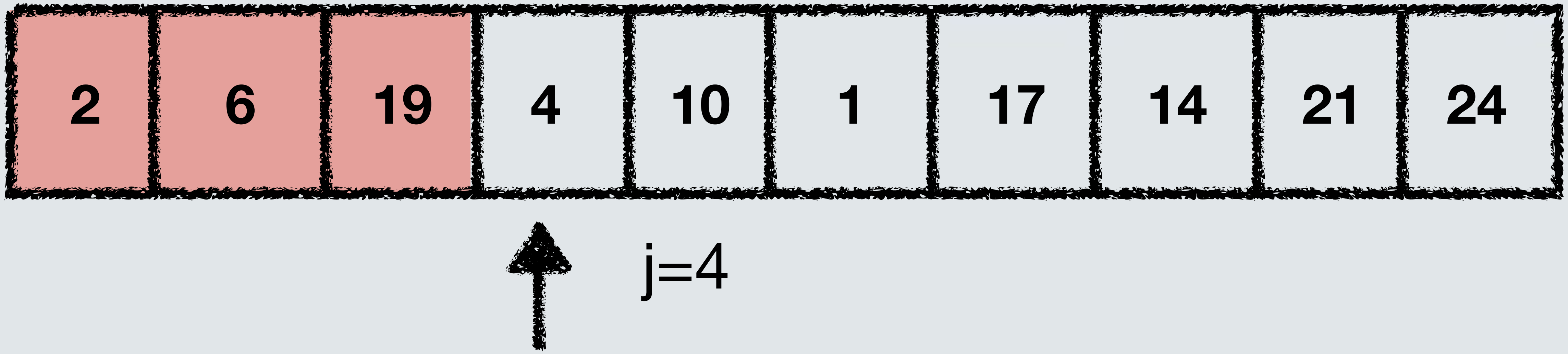
j=4

# Describing algorithms: Pseudocode

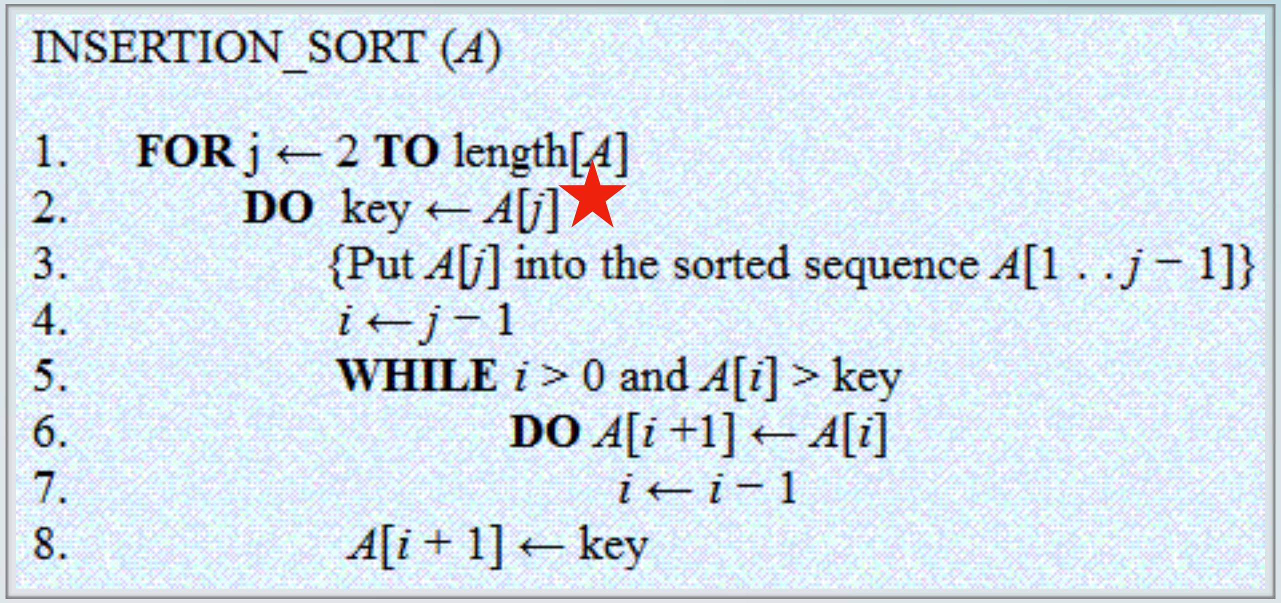


```
INSERTION_SORT ($A$)

FOR j ← 2 TO length[$A$]
    DO key ← $A[j]$ ★
        {Put $A[j]$ into the sorted sequence $A[1 . . j − 1]$}
        $i ← j − 1$
        WHILE $i > 0$ and $A[i]$ > key
            DO $A[i +1] ← A[i]$
                $i ← i − 1$
        $A[i + 1]$ ← key
```

- The algorithm maintains a sorted array in each iteration (each time the for loop is executed).

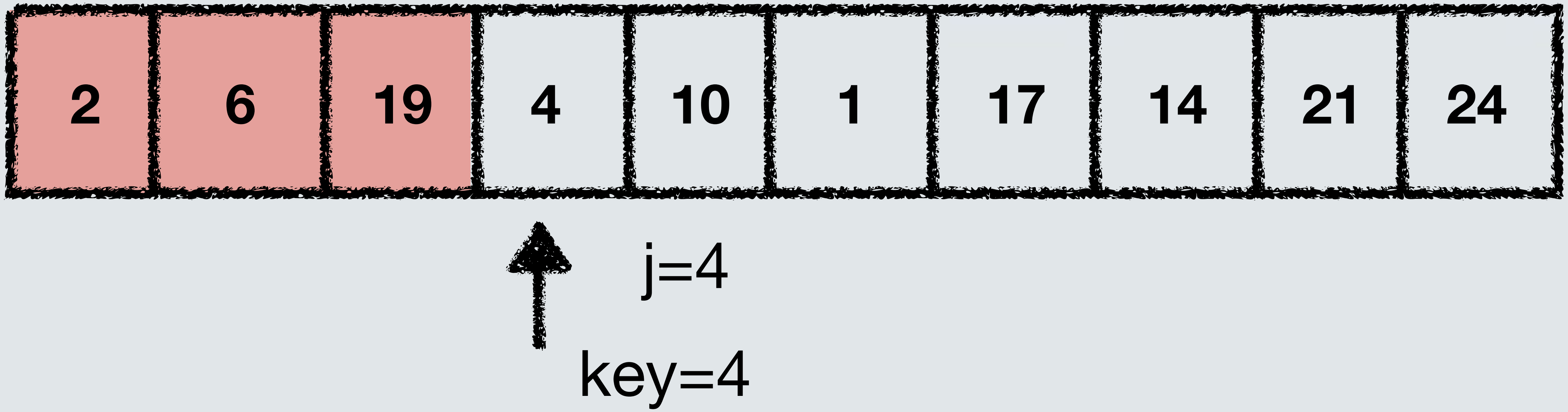


# Describing algorithms: Pseudocode



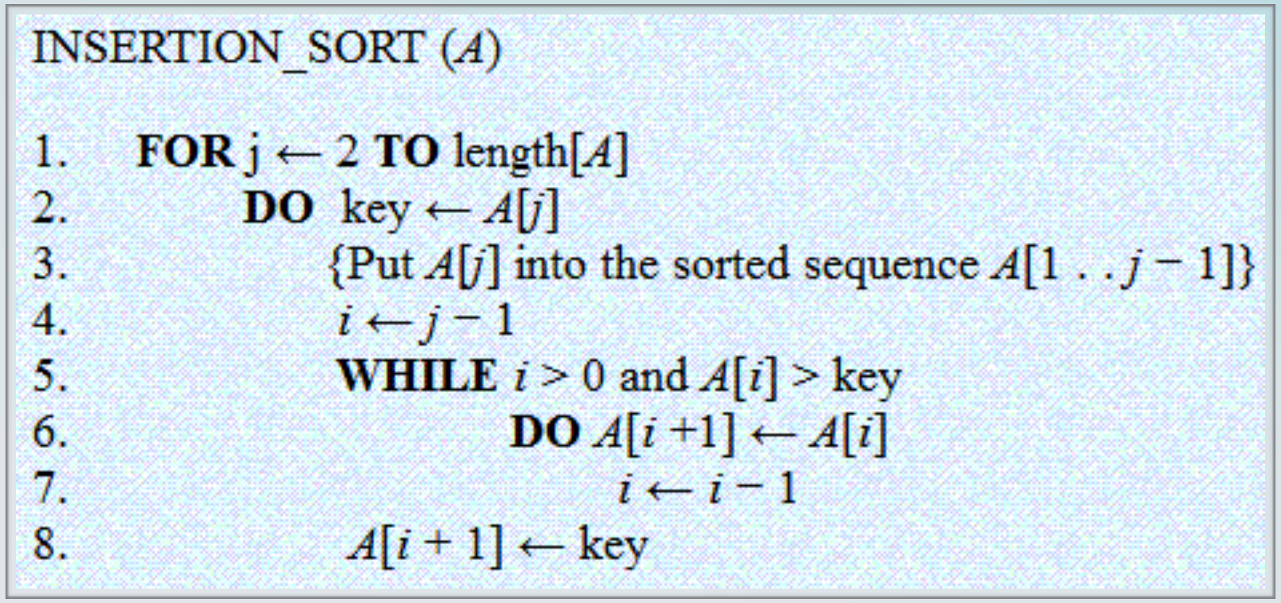
```
INSERTION_SORT ($A$)

FOR j ← 2 TO length[$A$]
    DO key ← $A[j]$ ★
        {Put $A[j]$ into the sorted sequence $A[1 . . j - 1]$}
        $i \leftarrow j - 1$
        WHILE $i > 0$ and $A[i] >$ key
            DO $A[i + 1] \leftarrow A[i]$
                $i \leftarrow i - 1$
        $A[i + 1]$ ← key
```

- The algorithm maintains a sorted array in each iteration (each time the for loop is executed).



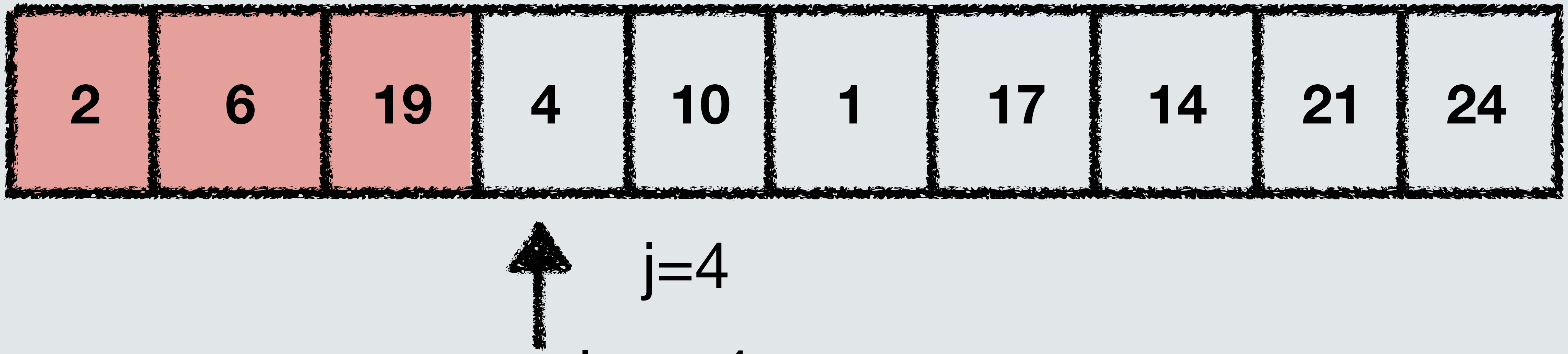
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 6 | 19 | 4 | 10 | 1 | 17 | 14 | 21 | 24 |

# Describing algorithms: Pseudocode



```
INSERTION_SORT (A)

FOR j ← 2 TO length[A]
    DO  key ← A[j]
        {Put A[j] into the sorted sequence A[1 . . j − 1]}
        i ← j − 1
        WHILE i > 0 and A[i] > key
              DO A[i +1] ← A[i]
                 i ← i − 1
        A[i + 1] ← key
```

- The algorithm maintains a sorted array in each iteration (each time the for loop is executed).

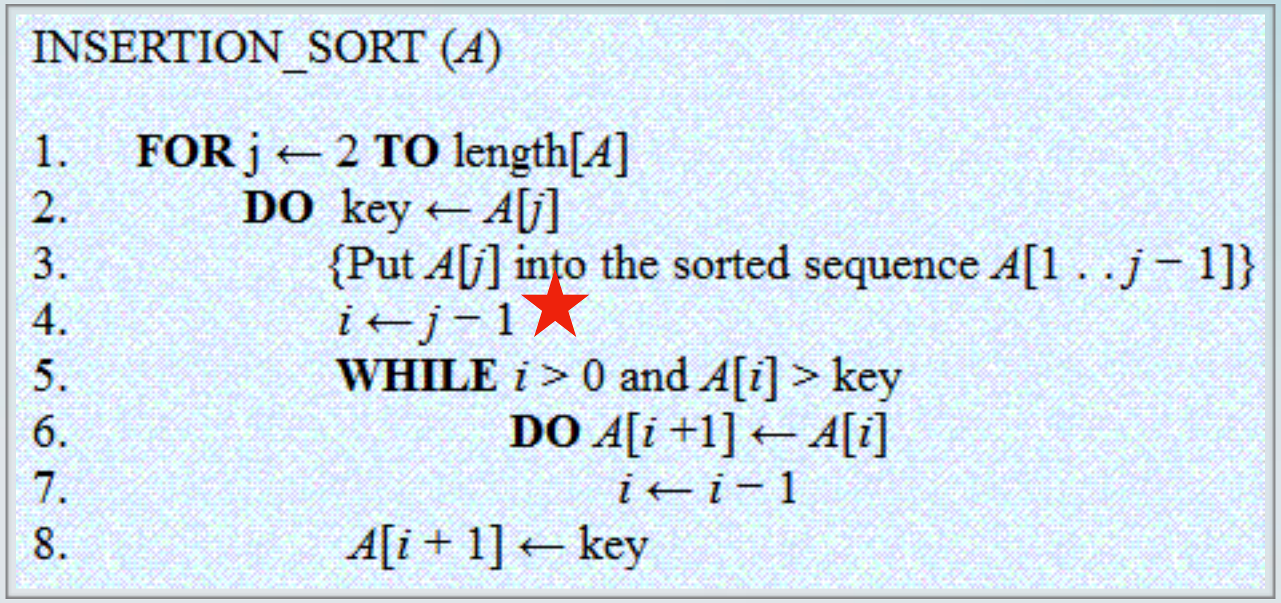


| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 6 | 19 | 4 | 10 | 1 | 17 | 14 | 21 | 24 |

j=4

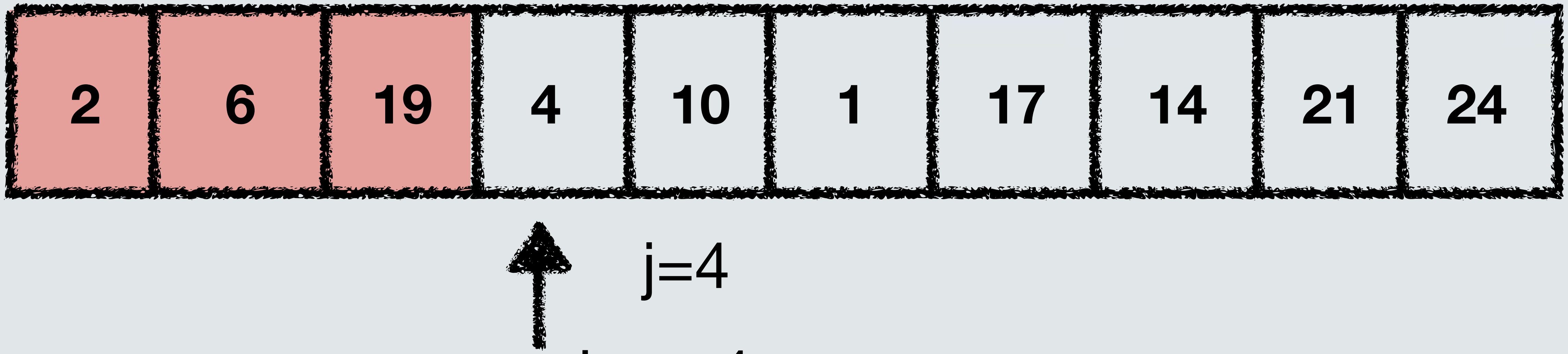
key=4

# Describing algorithms: Pseudocode



```
INSERTION_SORT ($A$)

FOR j ← 2 TO length[$A$]
    DO key ← $A[j]$
        {Put $A[j]$ into the sorted sequence $A[1..j-1]$}
        $i \leftarrow j - 1$ ★
        WHILE $i > 0$ and $A[i]$ > key
            DO $A[i+1] \leftarrow A[i]$
               $i \leftarrow i - 1$
        $A[i+1]$ ← key
```

- The algorithm maintains a sorted array in each iteration (each time the for loop is executed).

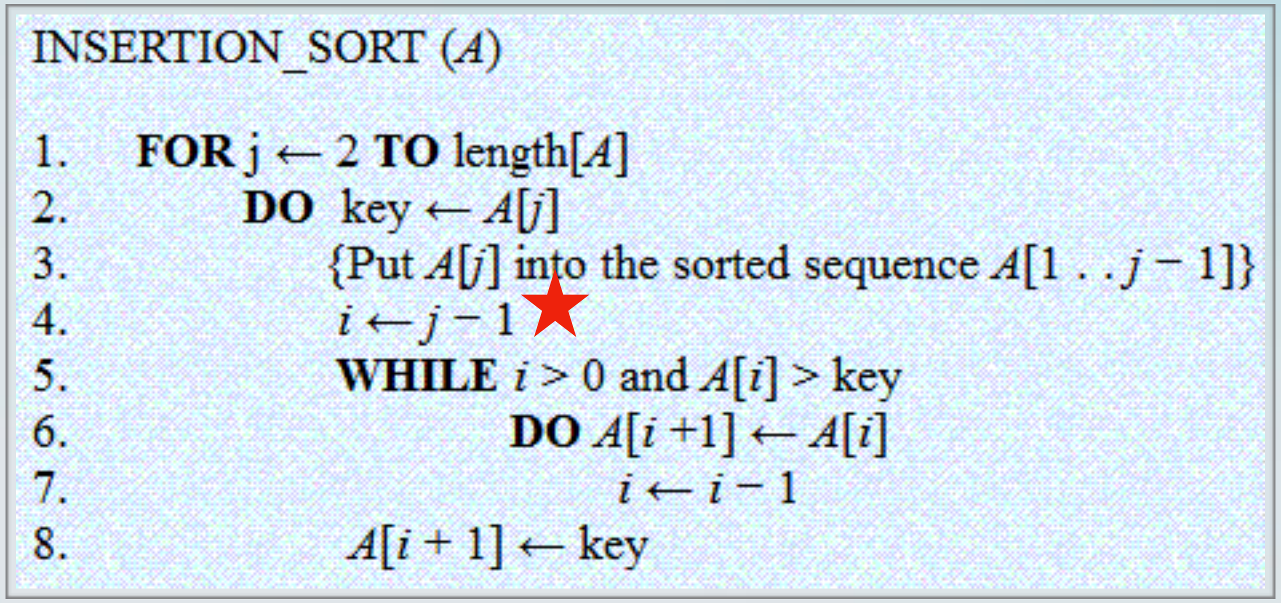


|  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 6 | 19 | 4 | 10 | 1 | 17 | 14 | 21 | 24 |

j=4

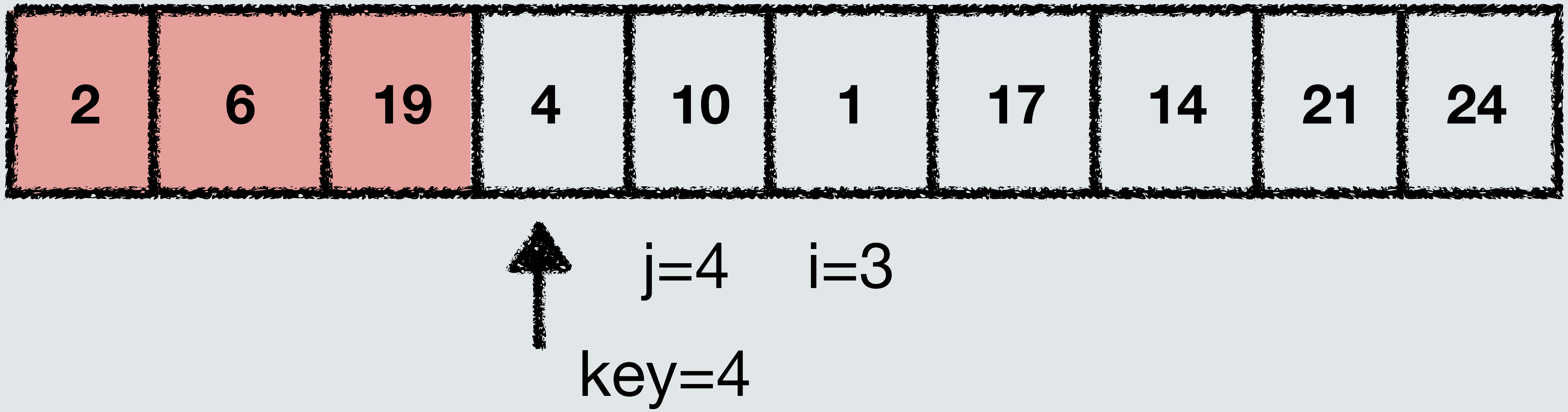
key=4

# Describing algorithms: Pseudocode



```
INSERTION_SORT ($A$)

FOR j ← 2 TO length[$A$]
        DO  key ← $A[j]$
              {Put $A[j]$ into the sorted sequence $A[1 . . j − 1]$}
              $i ← j − 1$ ★
              WHILE $i > 0$ and $A[i]$ > key
                        DO $A[i + 1] ← A[i]$
                              $i ← i − 1$
               $A[i + 1]$ ← key
```

- The algorithm maintains a sorted array in each iteration (each time the for loop is executed).

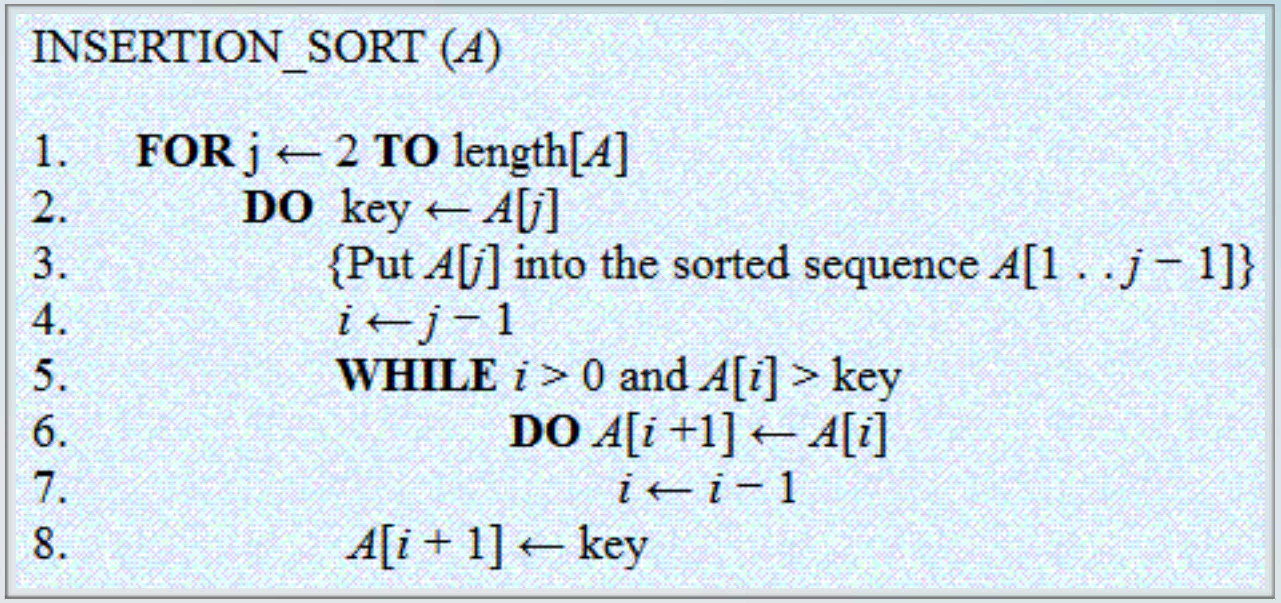


| 2 | 6 | 19 | 4 | 10 | 1 | 17 | 14 | 21 | 24 |
|---|---|---|---|---|---|---|---|---|---|

j=4 i=3

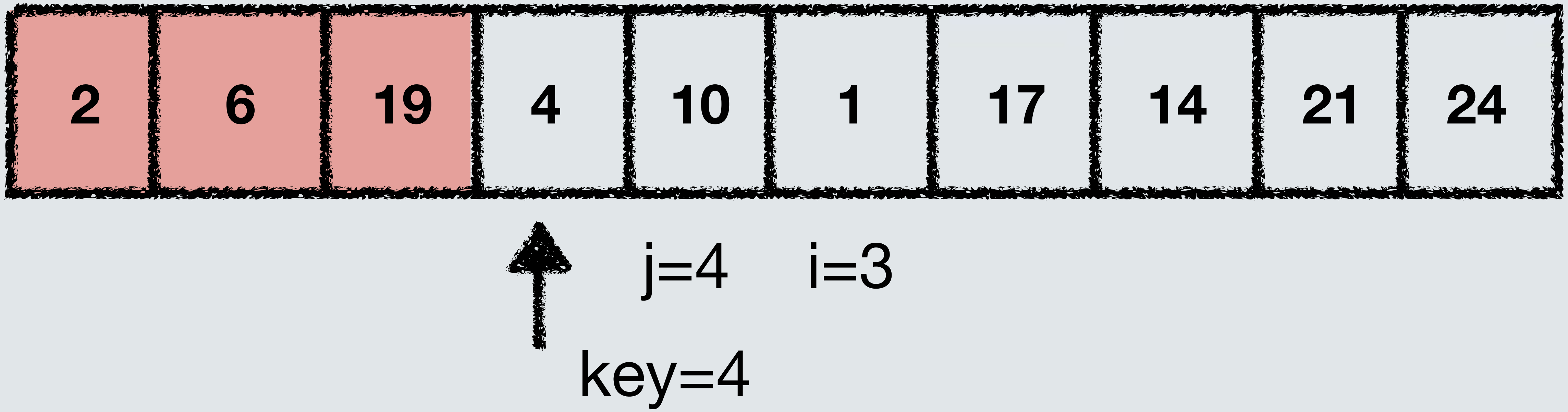
key=4

# Describing algorithms: Pseudocode



```
INSERTION_SORT (A)

FOR j ← 2 TO length[A]
    DO key ← A[j]
        {Put A[j] into the sorted sequence A[1 . . j − 1]}
        i ← j − 1
        WHILE i > 0 and A[i] > key
            DO A[i +1] ← A[i]
                i ← i − 1
        A[i + 1] ← key
```

- The algorithm maintains a sorted array in each iteration (each time the for loop is executed).

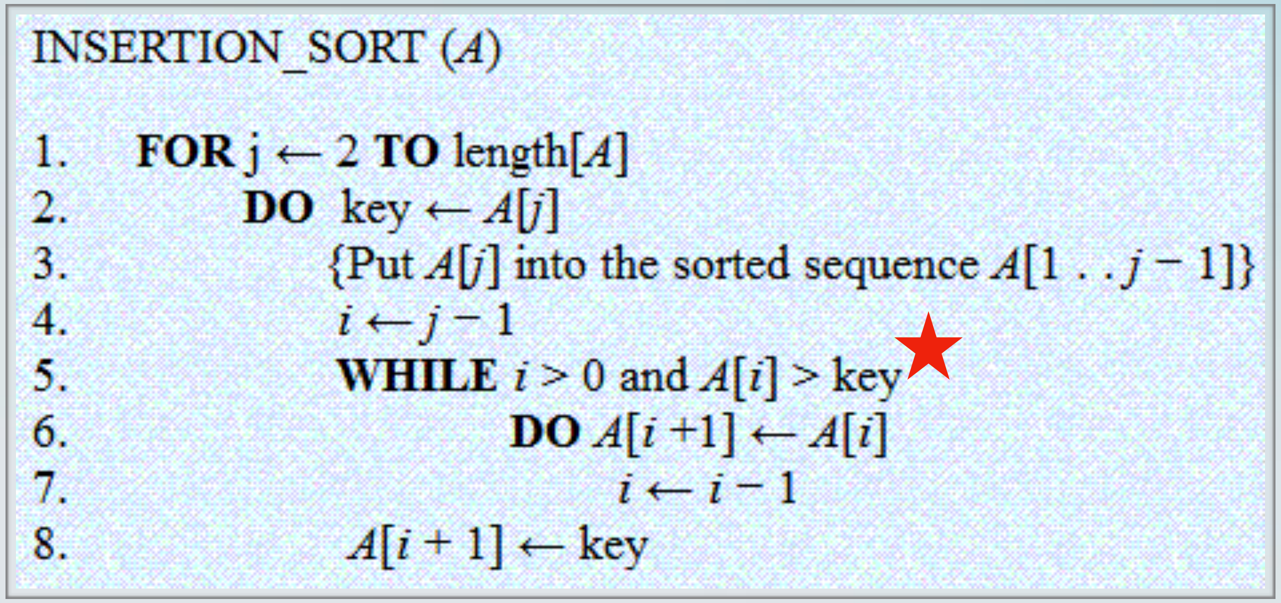


| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 6 | 19 | 4 | 10 | 1 | 17 | 14 | 21 | 24 |

j=4 i=3

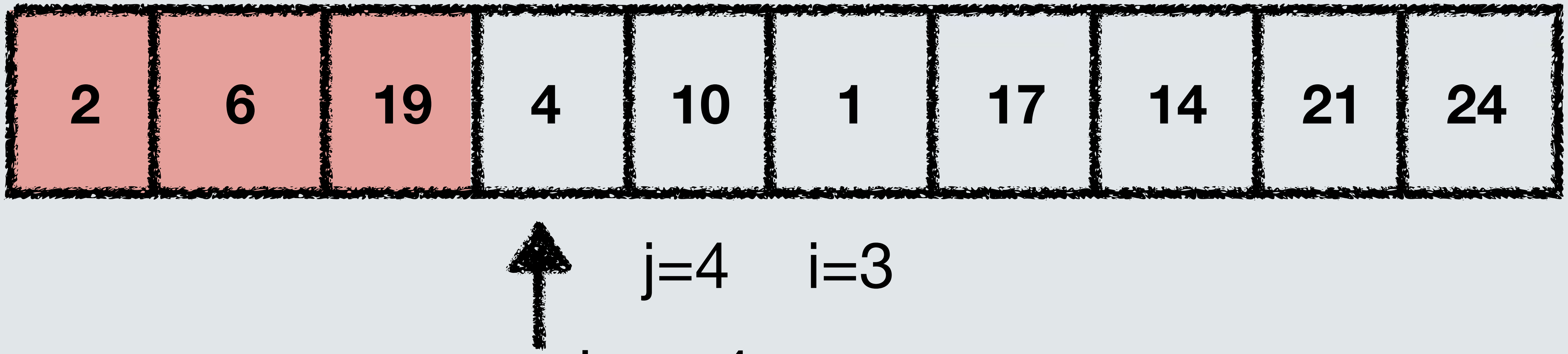
key=4

# Describing algorithms: Pseudocode



```
INSERTION_SORT ($A$)

FOR j ← 2 TO length[$A$]
    DO key ← $A[j]$
        {Put $A[j]$ into the sorted sequence $A[1 . . j − 1]$}
        $i ← j − 1$
        WHILE $i > 0$ and $A[i]$ > key
            DO $A[i + 1] ← A[i]$
               $i ← i − 1$
        $A[i + 1]$ ← key
```

- The algorithm maintains a sorted array in each iteration (each time the for loop is executed).

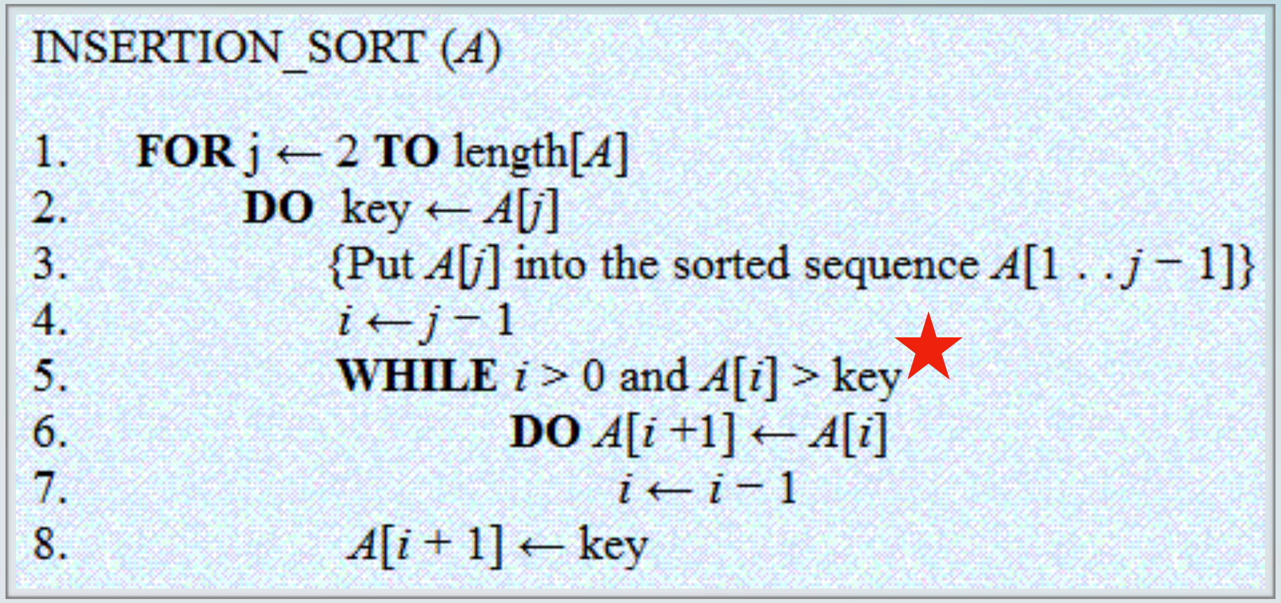


| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 6 | 19 | 4 | 10 | 1 | 17 | 14 | 21 | 24 |

j=4 i=3

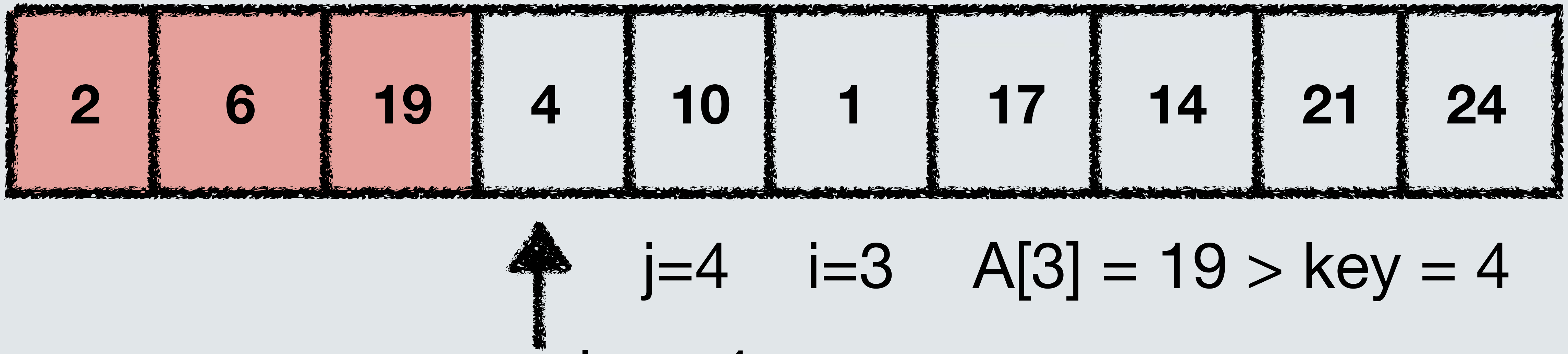
key=4

# Describing algorithms: Pseudocode



```
INSERTION_SORT (A)

FOR j ← 2 TO length[A]
    DO key ← A[j]
        {Put A[j] into the sorted sequence A[1 . . j − 1]}
        i ← j − 1
        WHILE i > 0 and A[i] > key
            DO A[i +1] ← A[i]
                i ← i − 1
        A[i + 1] ← key
```

- The algorithm maintains a sorted array in each iteration (each time the for loop is executed).

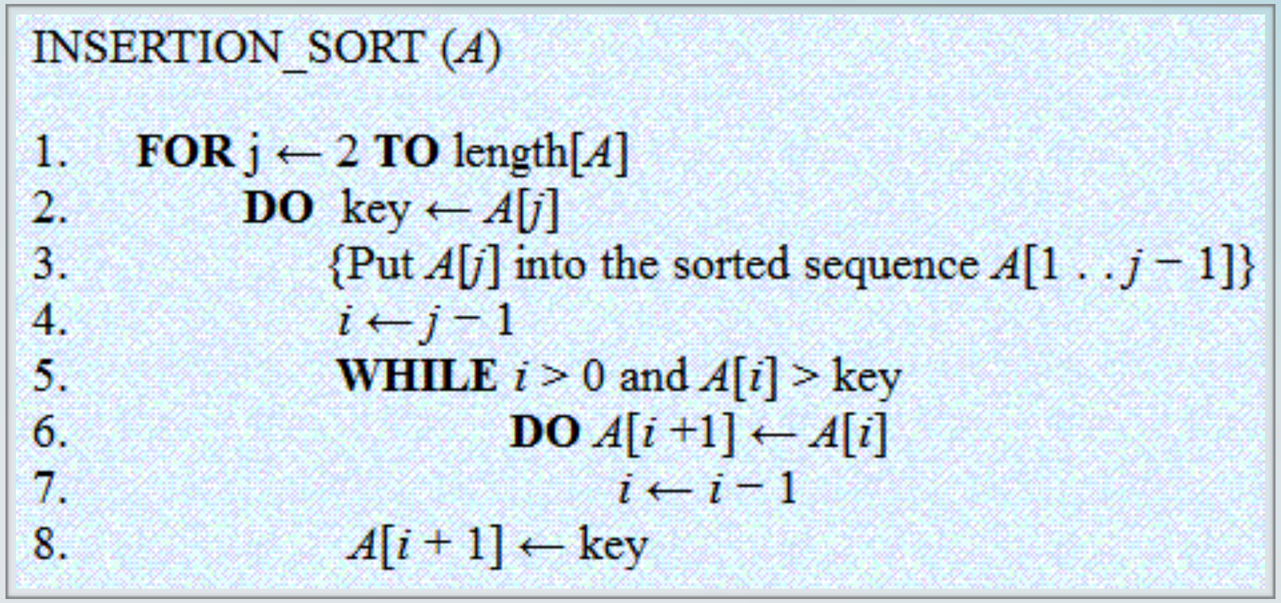


| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 6 | 19 | 4 | 10 | 1 | 17 | 14 | 21 | 24 |

j=4 i=3 A[3] = 19 > key = 4

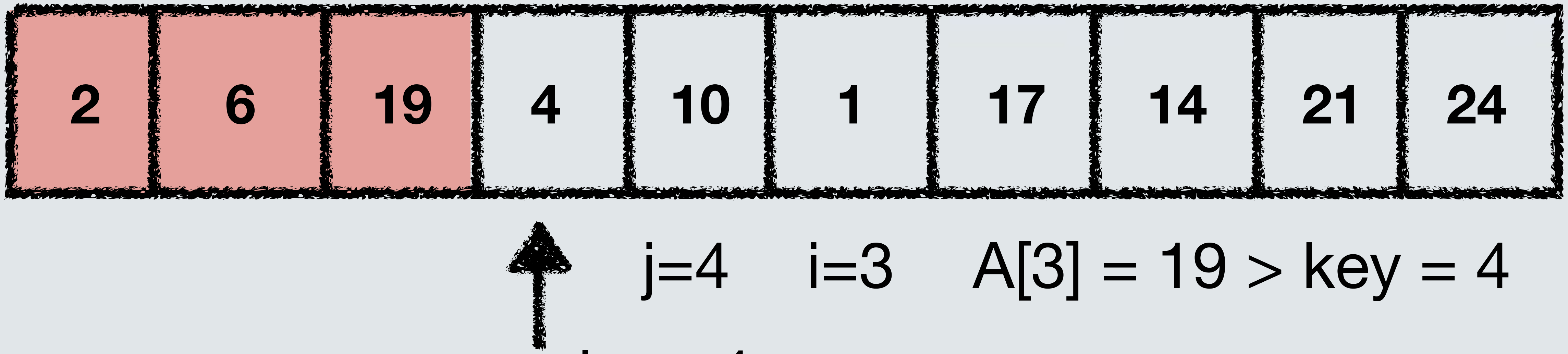
key=4

# Describing algorithms: Pseudocode



```
INSERTION_SORT (A)

FOR j ← 2 TO length[A]
    DO key ← A[j]
        {Put A[j] into the sorted sequence A[1 . . j − 1]}
        i ← j − 1
        WHILE i > 0 and A[i] > key
            DO A[i +1] ← A[i]
                i ← i − 1
        A[i + 1] ← key
```

- The algorithm maintains a sorted array in each iteration (each time the for loop is executed).

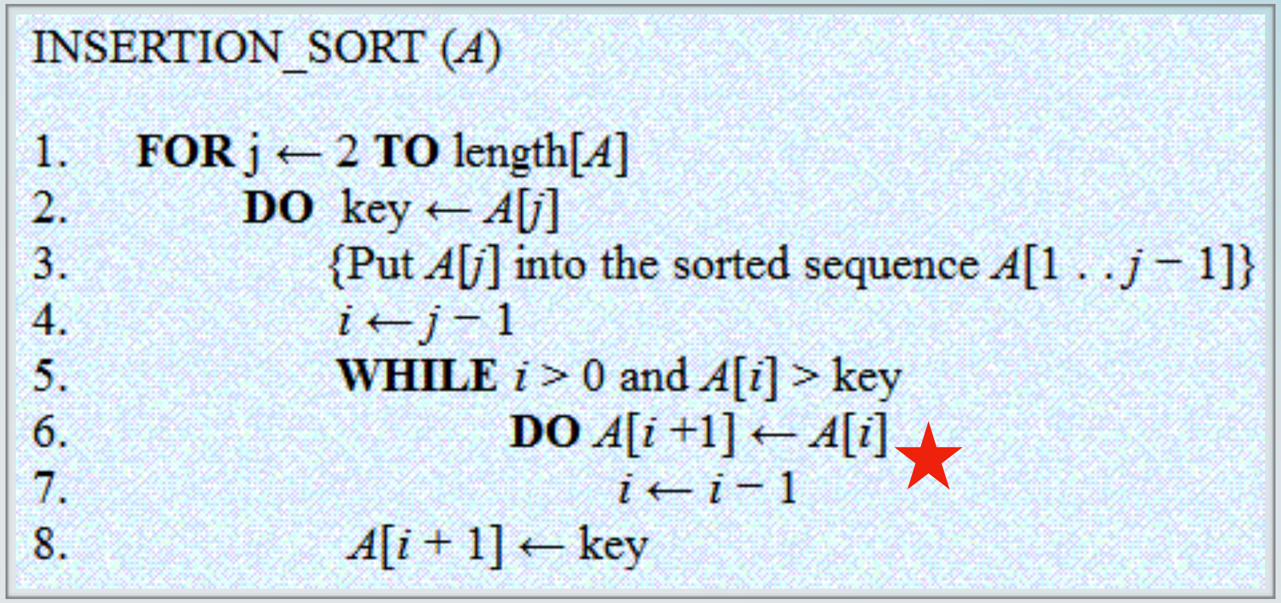


| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 6 | 19 | 4 | 10 | 1 | 17 | 14 | 21 | 24 |

j=4 i=3 A[3] = 19 > key = 4

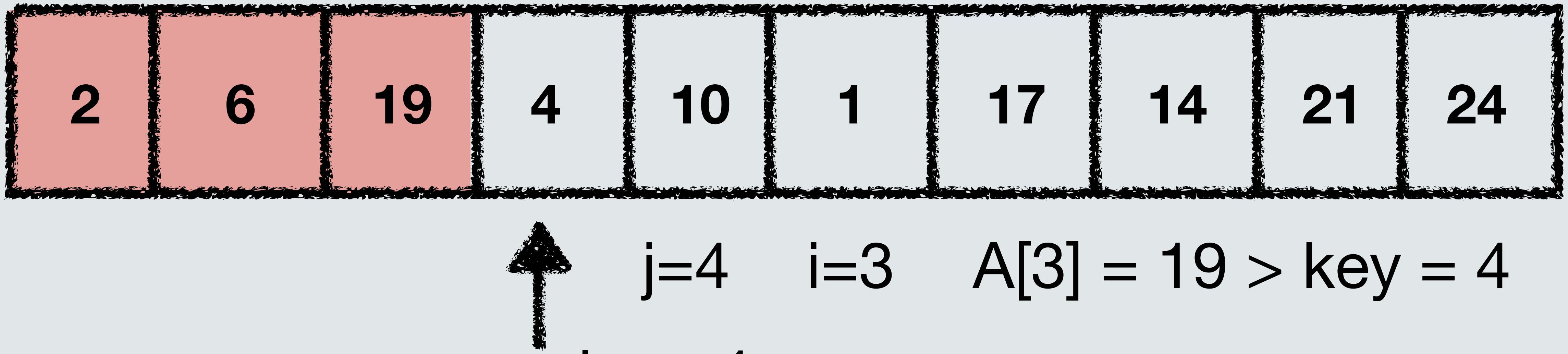
key=4

# Describing algorithms: Pseudocode



```
INSERTION_SORT (A)

FOR j ← 2 TO length[A]
    DO  key ← A[j]
        {Put A[j] into the sorted sequence A[1 . . j − 1]}
        i ← j − 1
        WHILE i > 0 and A[i] > key
            DO A[i +1] ← A[i] ★
               i ← i − 1
        A[i + 1] ← key
```

- The algorithm maintains a sorted array in each iteration (each time the for loop is executed).

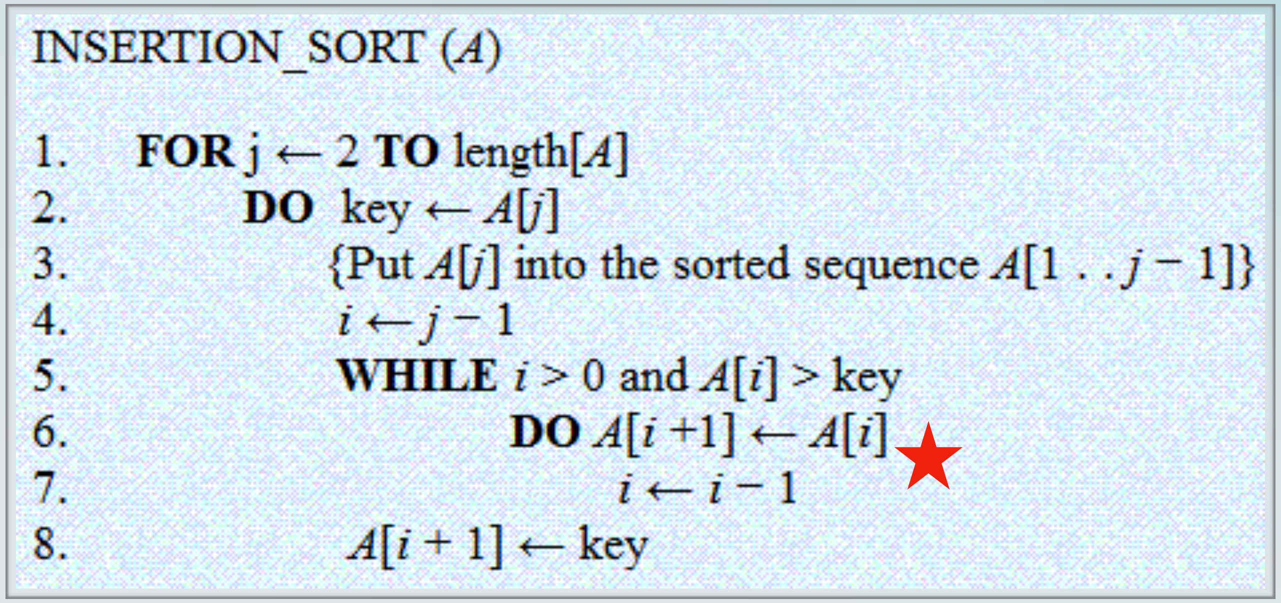


| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 6 | 19 | 4 | 10 | 1 | 17 | 14 | 21 | 24 |

j=4 i=3 A[3] = 19 > key = 4

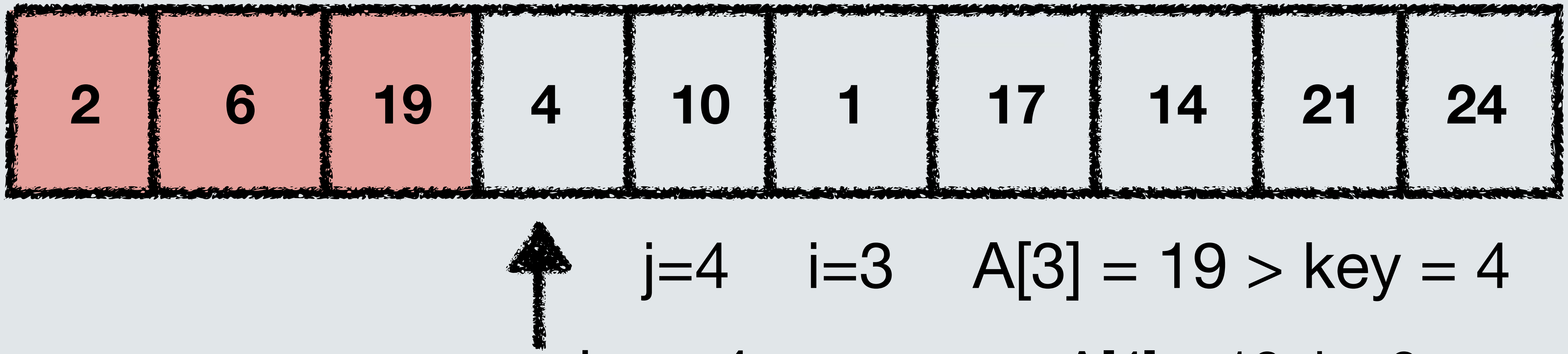
key=4

# Describing algorithms: Pseudocode



```
INSERTION_SORT (A)

FOR j ← 2 TO length[A]
        DO  key ← A[j]
            {Put A[j] into the sorted sequence A[1 . . j − 1]}
            i ← j − 1
            WHILE i > 0 and A[i] > key
                    DO A[i +1] ← A[i] ★
                        i ← i − 1
            A[i + 1] ← key
```

- The algorithm maintains a sorted array in each iteration (each time the for loop is executed).



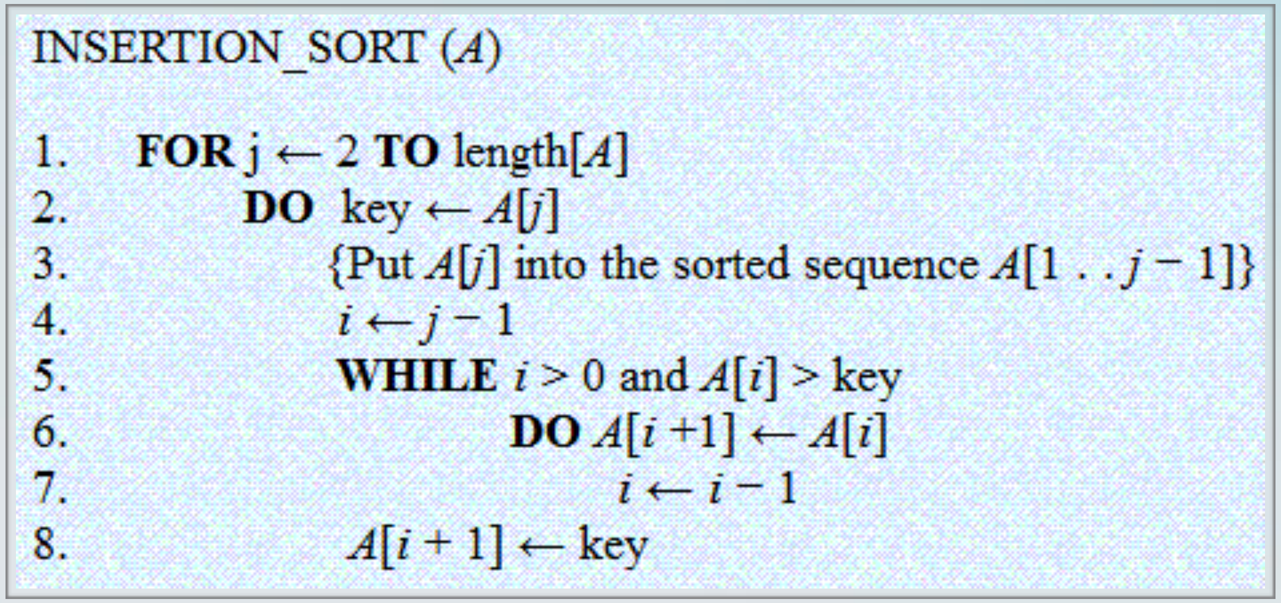
| 2 | 6 | 19 | 4 | 10 | 1 | 17 | 14 | 21 | 24 |
|---|---|---|---|---|---|---|---|---|---|

j=4 i=3 A[3] = 19 > key = 4

key=4

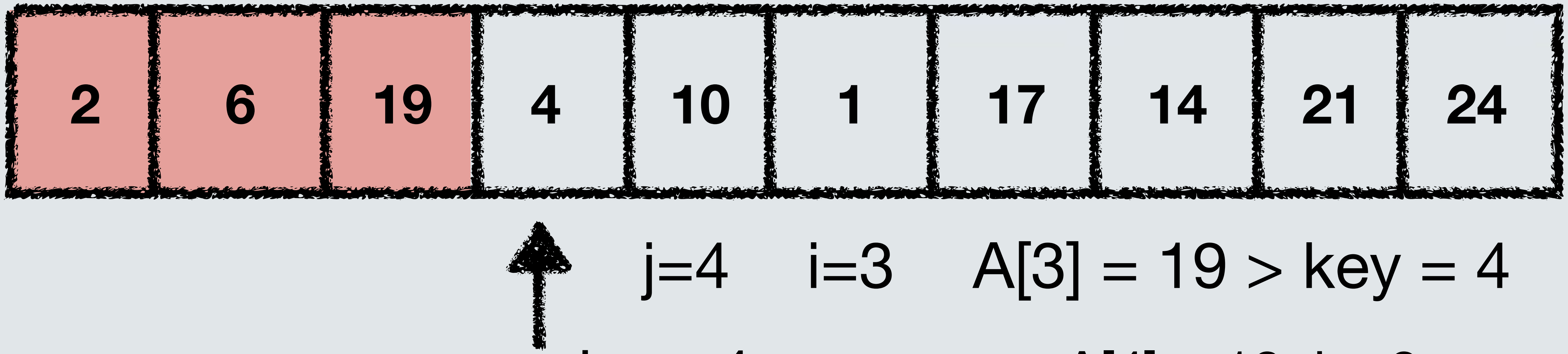
A[4] = 19, i = 2

# Describing algorithms: Pseudocode



```
INSERTION_SORT ($A$)

FOR j $\leftarrow$ 2 TO length[$A$]
    DO key $\leftarrow$ $A[j]$
        {Put $A[j]$ into the sorted sequence $A[1 \,..\, j-1]$}
        $i \leftarrow j - 1$
        WHILE $i > 0$ and $A[i] >$ key
            DO $A[i+1] \leftarrow A[i]$
                $i \leftarrow i - 1$
        $A[i+1] \leftarrow$ key
```

- The algorithm maintains a sorted array in each iteration (each time the for loop is executed).



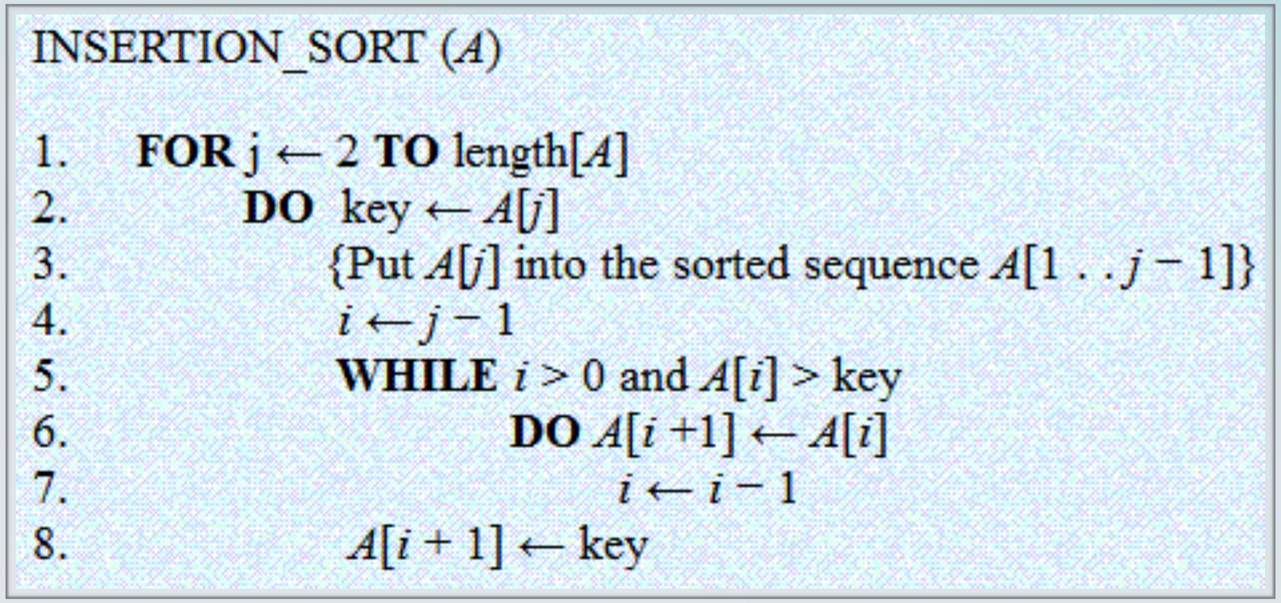
|  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 6 | 19 | 4 | 10 | 1 | 17 | 14 | 21 | 24 |

j=4 i=3 A[3] = 19 > key = 4

key=4

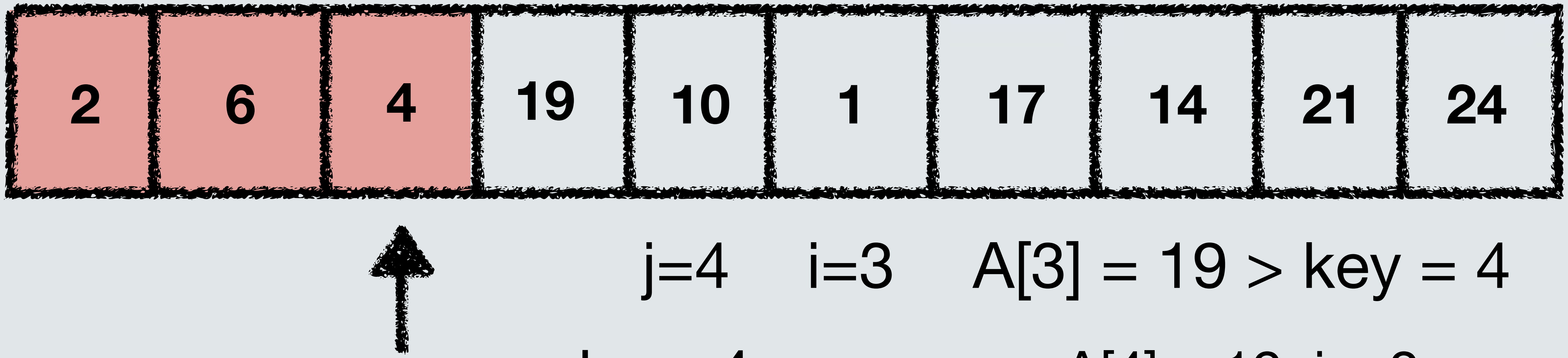
A[4] = 19, i = 2

# Describing algorithms: Pseudocode



```
INSERTION_SORT (A)

FOR j ← 2 TO length[A]
        DO  key ← A[j]
              {Put A[j] into the sorted sequence A[1 . . j − 1]}
              i ← j − 1
              WHILE i > 0 and A[i] > key
                        DO A[i +1] ← A[i]
                              i ← i − 1
               A[i + 1] ← key
```

- The algorithm maintains a sorted array in each iteration (each time the for loop is executed).

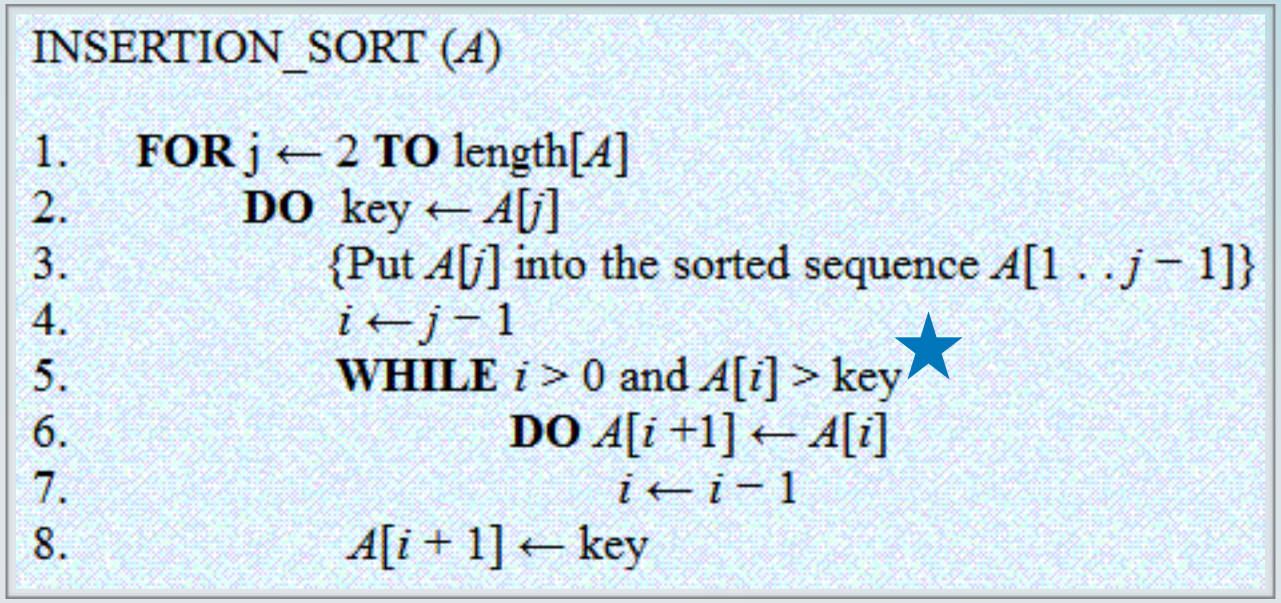


| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 6 | 4 | 19 | 10 | 1 | 17 | 14 | 21 | 24 |

j=4 i=3 A[3] = 19 > key = 4

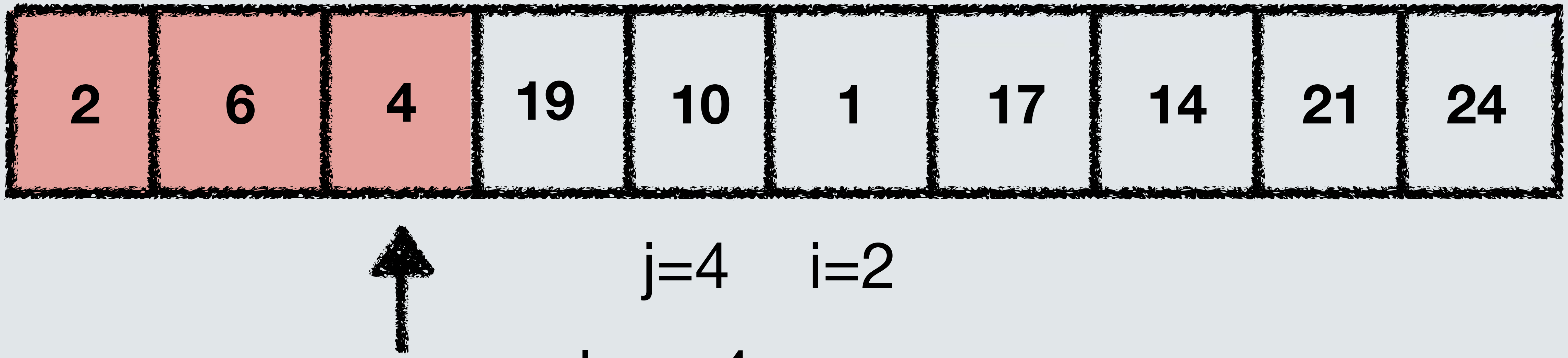
key=4 A[4] = 19, i = 2

# Describing algorithms: Pseudocode



```
INSERTION_SORT ($A$)

FOR j $\leftarrow$ 2 TO length[$A$]
    DO key $\leftarrow$ $A[j]$
        {Put $A[j]$ into the sorted sequence $A[1..j-1]$}
        $i \leftarrow j-1$
        WHILE $i > 0$ and $A[i] >$ key ★
            DO $A[i+1] \leftarrow A[i]$
                $i \leftarrow i-1$
        $A[i+1] \leftarrow$ key
```

- The algorithm maintains a sorted array in each iteration (each time the for loop is executed).



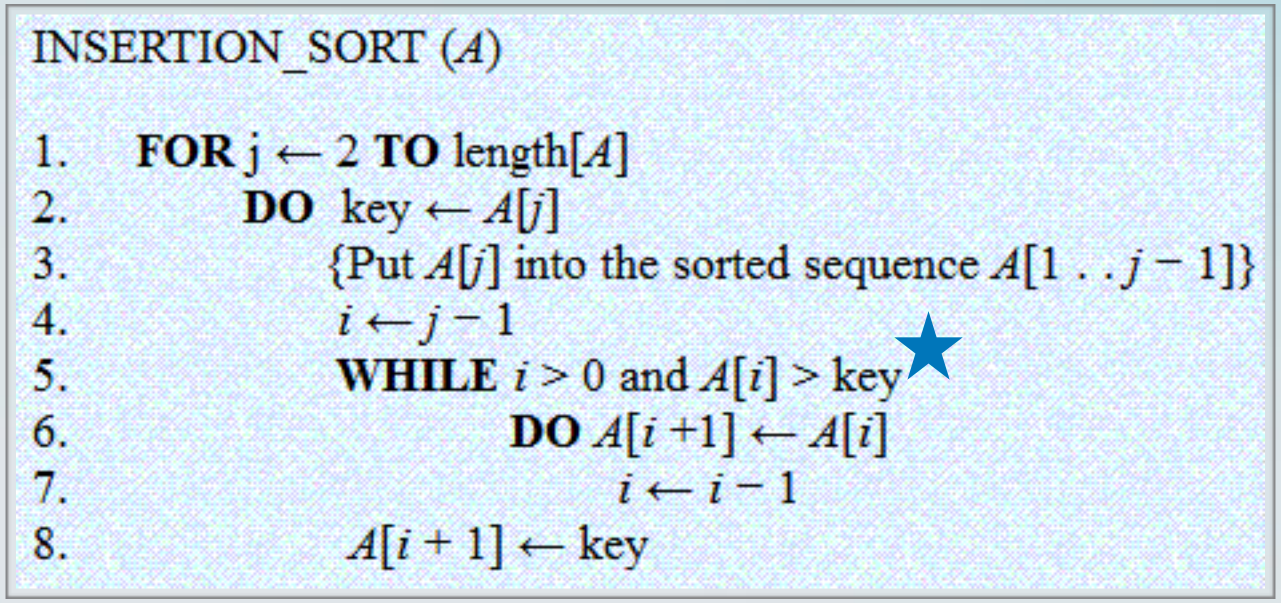
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 6 | 4 | 19 | 10 | 1 | 17 | 14 | 21 | 24 |

j=4 i=2

key=4

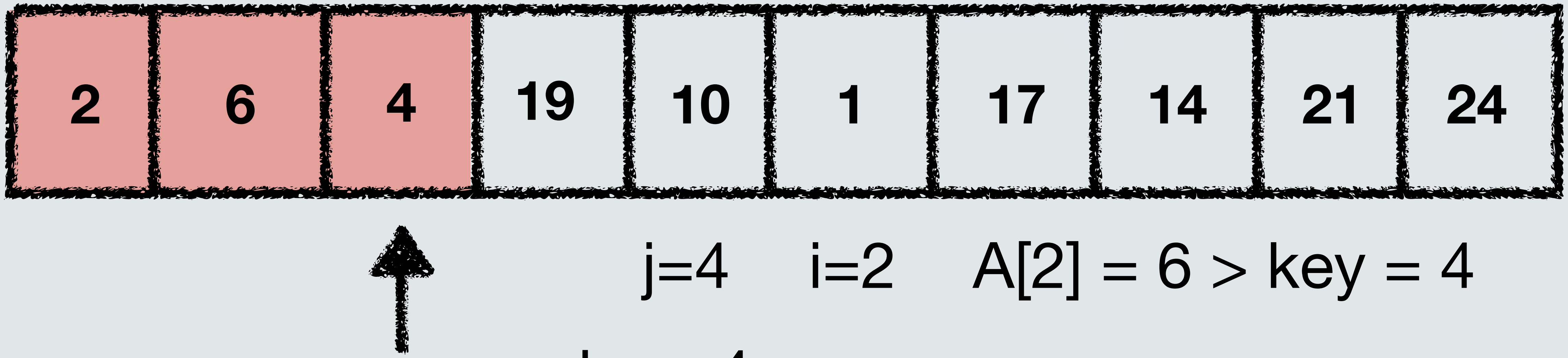
still in the while loop

# Describing algorithms: Pseudocode



```
INSERTION_SORT (A)

FOR j ← 2 TO length[A]
    DO  key ← A[j]
        {Put A[j] into the sorted sequence A[1 . . j − 1]}
        i ← j − 1
        WHILE i > 0 and A[i] > key ★
              DO A[i +1] ← A[i]
                 i ← i − 1
        A[i + 1] ← key
```

- The algorithm maintains a sorted array in each iteration (each time the for loop is executed).



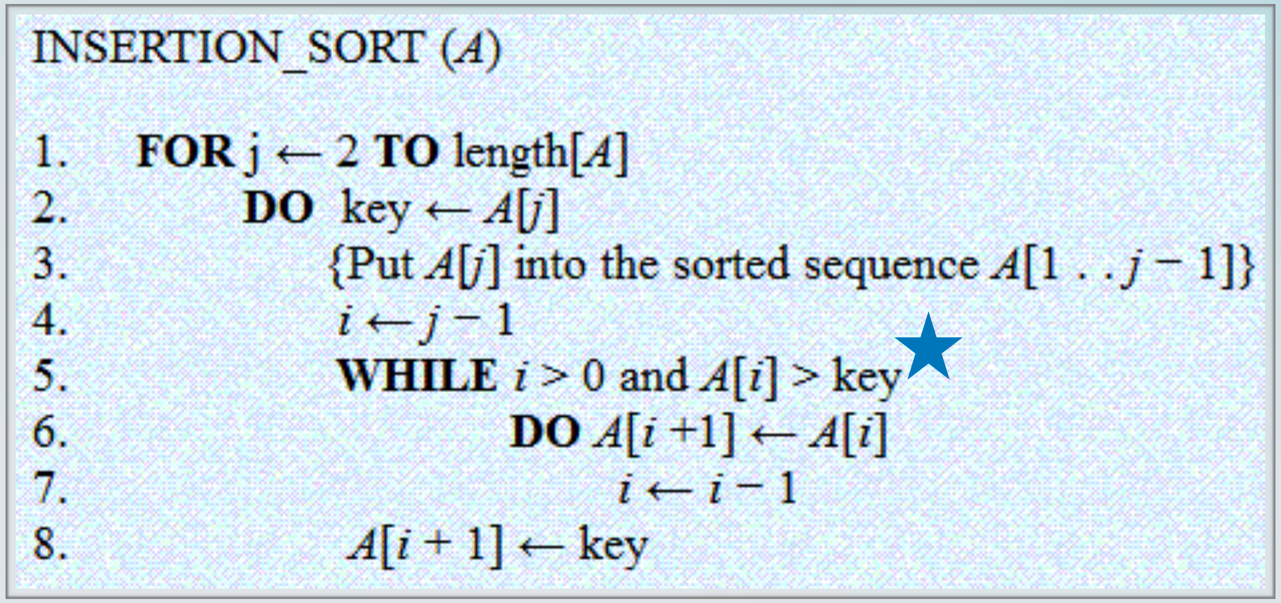
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 6 | 4 | 19 | 10 | 1 | 17 | 14 | 21 | 24 |

j=4 i=2 A[2] = 6 > key = 4

key=4

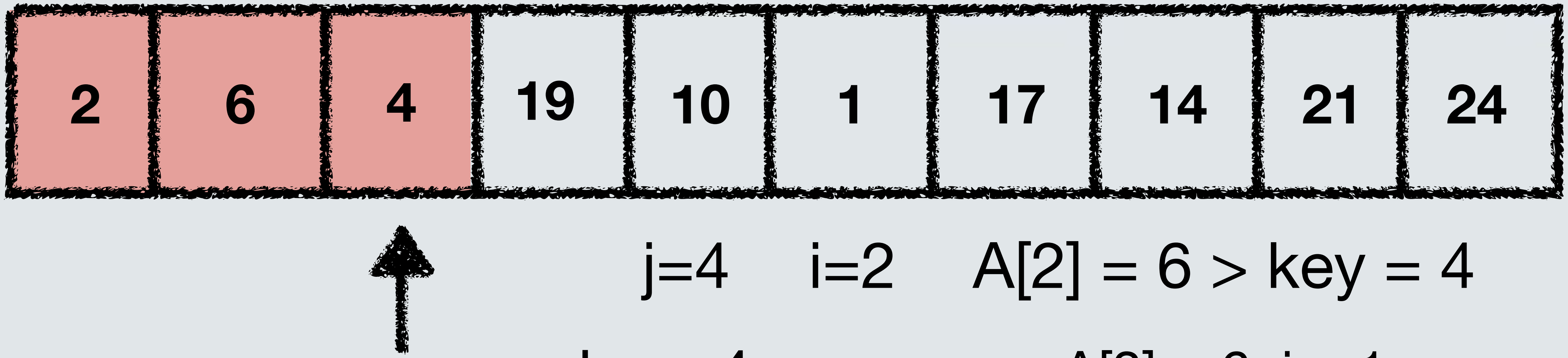
still in the while loop

# Describing algorithms: Pseudocode



```
INSERTION_SORT (A)

FOR j ← 2 TO length[A]
    DO  key ← A[j]
        {Put A[j] into the sorted sequence A[1 . . j − 1]}
        i ← j − 1
        WHILE i > 0 and A[i] > key ★
            DO A[i +1] ← A[i]
               i ← i − 1
        A[i + 1] ← key
```

- The algorithm maintains a sorted array in each iteration (each time the for loop is executed).



| 2 | 6 | 4 | 19 | 10 | 1 | 17 | 14 | 21 | 24 |
|---|---|---|---|---|---|---|---|---|---|

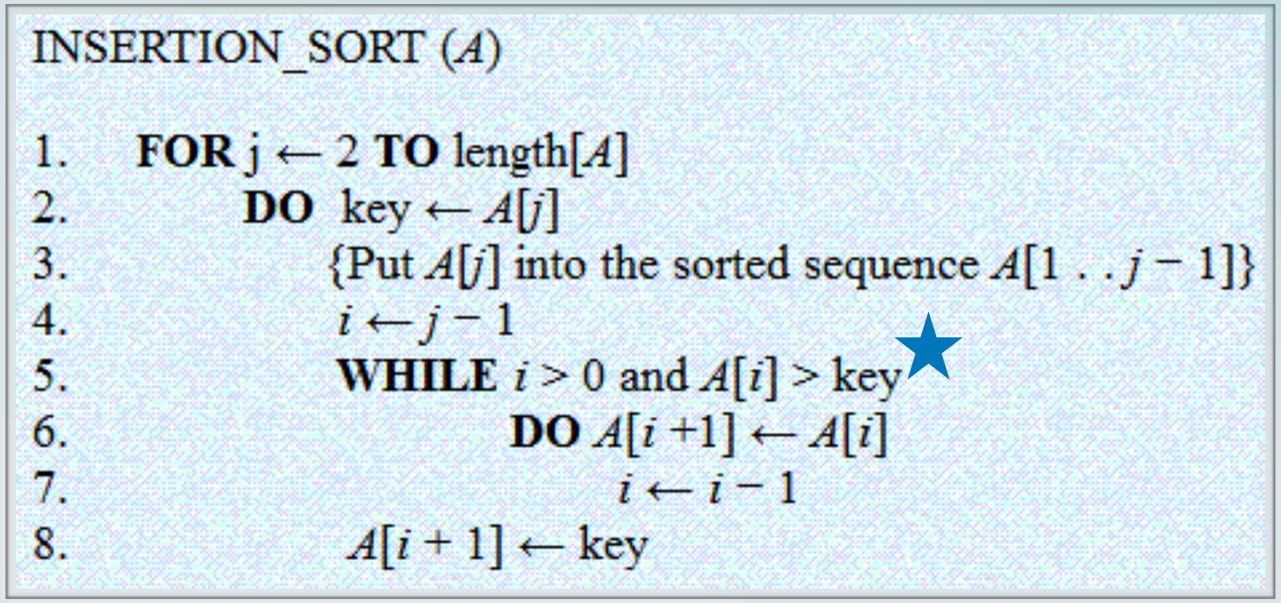
j=4 i=2 A[2] = 6 > key = 4

key=4

A[3] = 6, i = 1

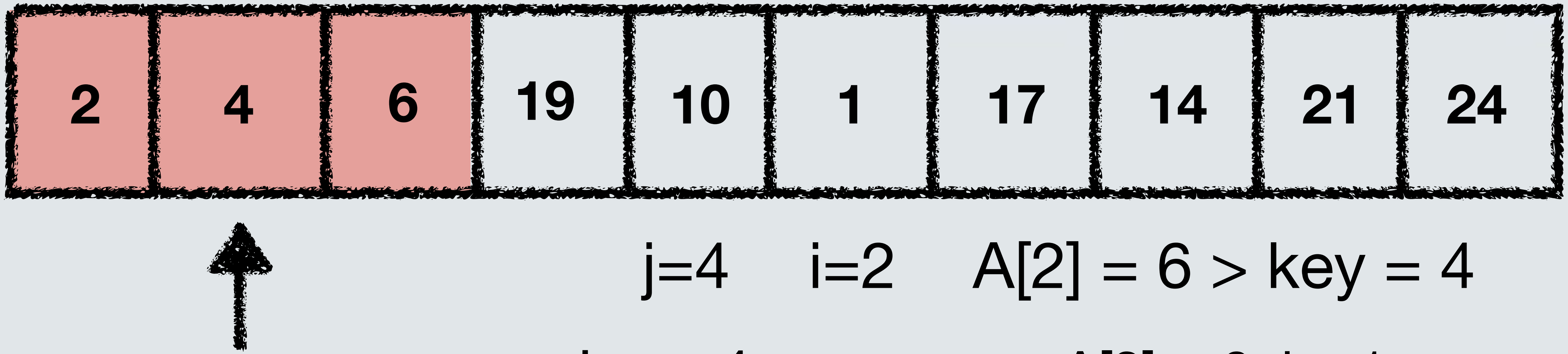
still in the while loop

# Describing algorithms: Pseudocode



```
INSERTION_SORT (A)

FOR j ← 2 TO length[A]
        DO  key ← A[j]
            {Put A[j] into the sorted sequence A[1 . . j − 1]}
            i ← j − 1
            WHILE i > 0 and A[i] > key ★
                    DO A[i +1] ← A[i]
                        i ← i − 1
            A[i + 1] ← key
```

- The algorithm maintains a sorted array in each iteration (each time the for loop is executed).



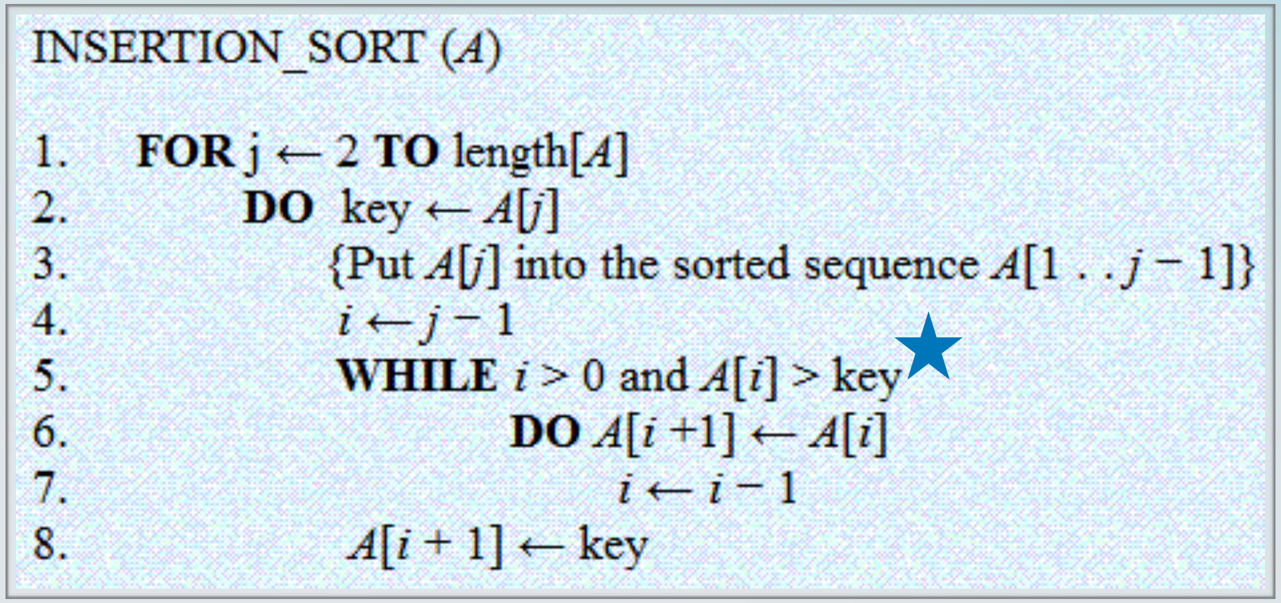
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 4 | 6 | 19 | 10 | 1 | 17 | 14 | 21 | 24 |

still in the while loop

j=4    i=2    A[2] = 6 > key = 4

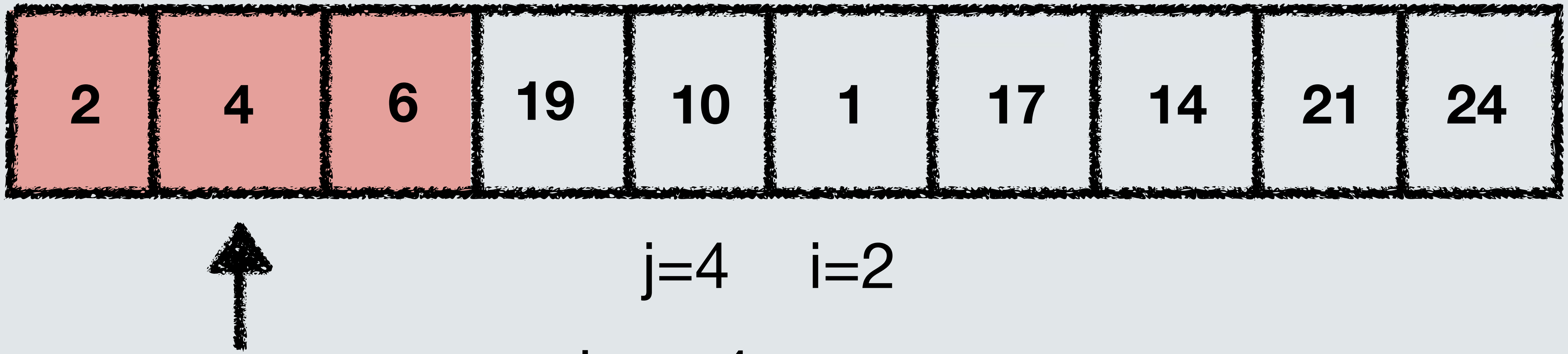
key=4    A[3] = 6, i = 1

# Describing algorithms: Pseudocode



```
INSERTION_SORT (A)

FOR j ← 2 TO length[A]
    DO key ← A[j]
        {Put A[j] into the sorted sequence A[1 . . j − 1]}
        i ← j − 1
        WHILE i > 0 and A[i] > key ★
            DO A[i +1] ← A[i]
                i ← i − 1
        A[i + 1] ← key
```

- The algorithm maintains a sorted array in each iteration (each time the for loop is executed).



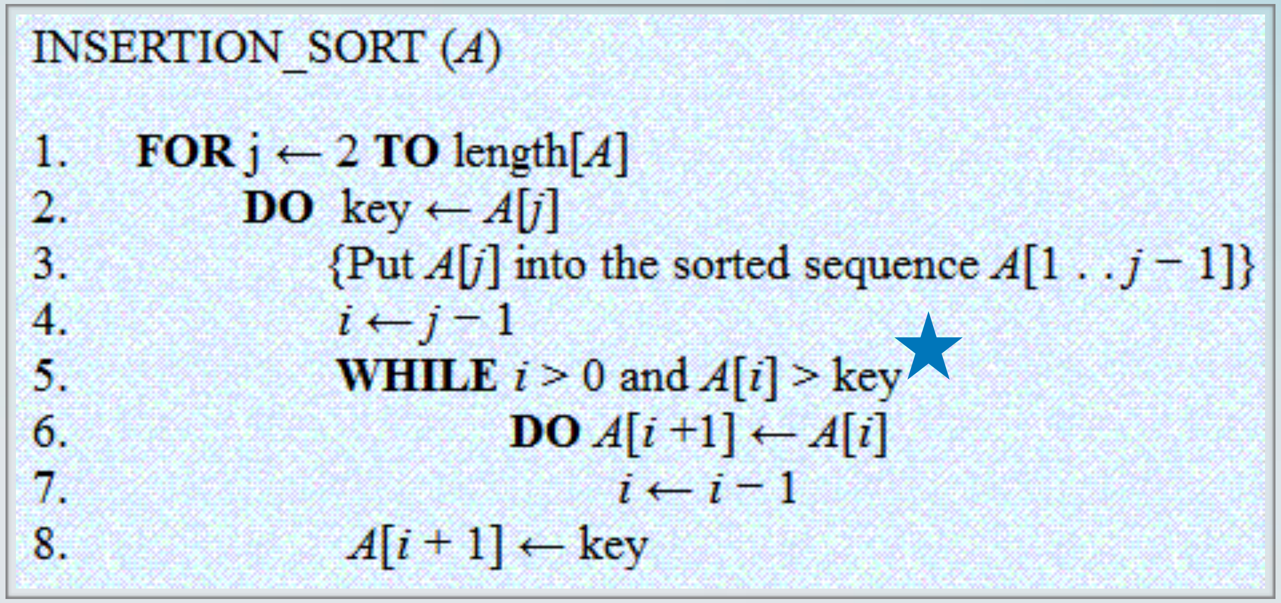
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 4 | 6 | 19 | 10 | 1 | 17 | 14 | 21 | 24 |

j=4 i=2

key=4

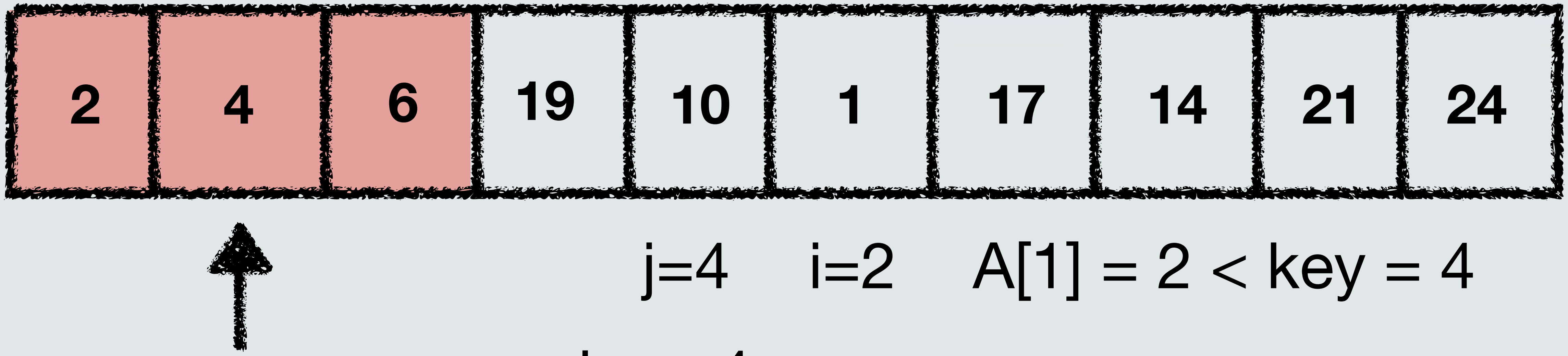
still in the while loop

# Describing algorithms: Pseudocode



```
INSERTION_SORT (A)

FOR j ← 2 TO length[A]
        DO key ← A[j]
                {Put A[j] into the sorted sequence A[1 . . j − 1]}
                i ← j − 1
                WHILE i > 0 and A[i] > key ★
                        DO A[i +1] ← A[i]
                                i ← i − 1
                A[i + 1] ← key
```

- The algorithm maintains a sorted array in each iteration (each time the for loop is executed).



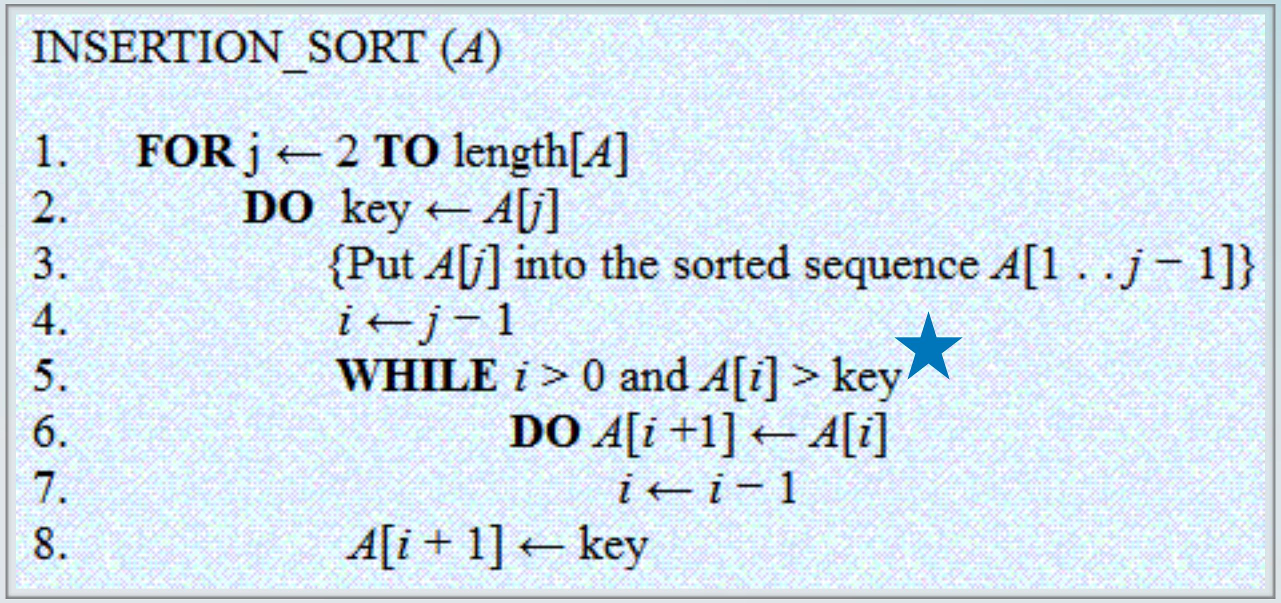
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 4 | 6 | 19 | 10 | 1 | 17 | 14 | 21 | 24 |

j=4 i=2 A[1] = 2 < key = 4

still in the while loop

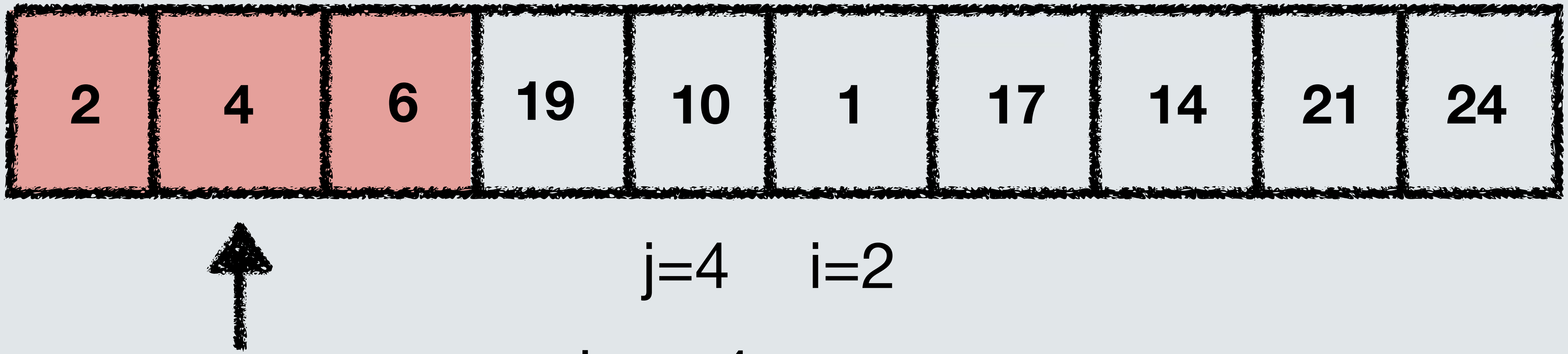
key=4

# Describing algorithms: Pseudocode



```
INSERTION_SORT (A)

FOR j ← 2 TO length[A]
        DO  key ← A[j]
              {Put A[j] into the sorted sequence A[1 . . j − 1]}
              i ← j − 1
              WHILE i > 0 and A[i] > key ★
                          DO A[i +1] ← A[i]
                                 i ← i − 1
               A[i + 1] ← key
```

- The algorithm maintains a sorted array in each iteration (each time the for loop is executed).



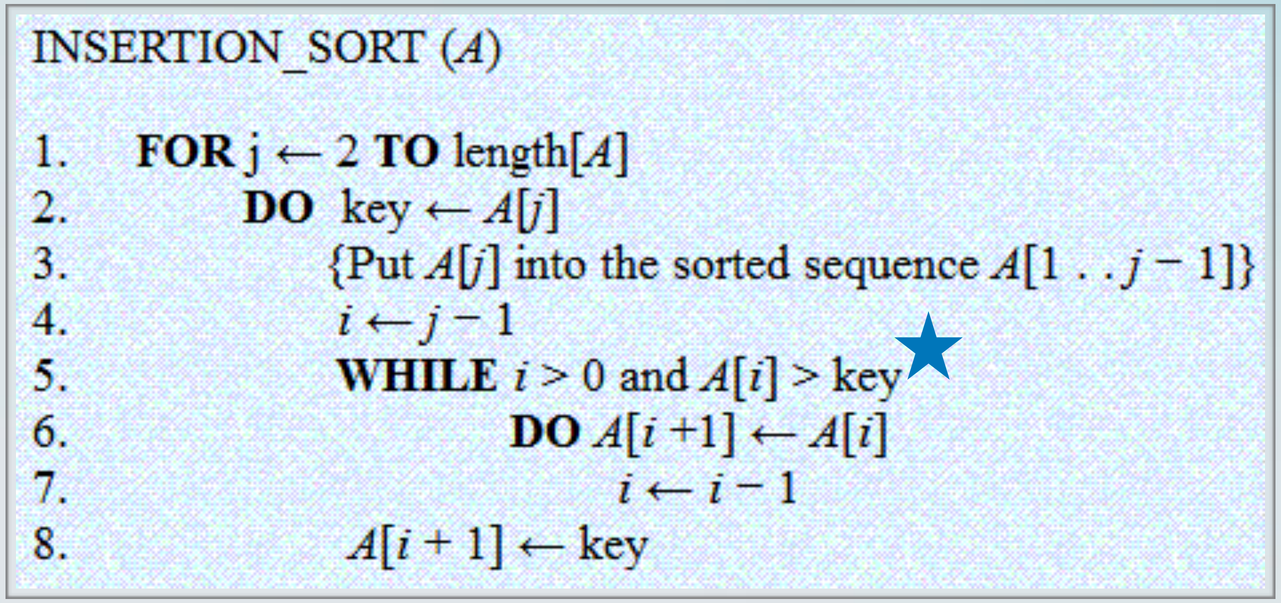
| 2 | 4 | 6 | 19 | 10 | 1 | 17 | 14 | 21 | 24 |
|---|---|---|---|---|---|---|---|---|---|

j=4 i=2

key=4

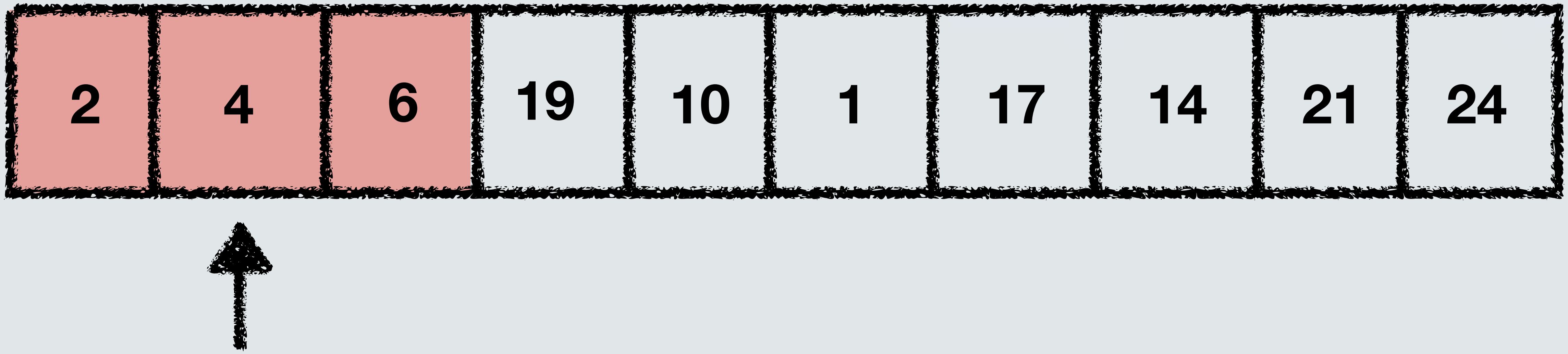
still in the while loop

# Describing algorithms: Pseudocode



```
INSERTION_SORT ($A$)

FOR j ← 2 TO length[$A$]
    DO key ← $A[j]$
        {Put $A[j]$ into the sorted sequence $A[1\ ..\ j-1]$}
        $i \leftarrow j-1$
        WHILE $i > 0$ and $A[i] >$ key ★
            DO $A[i+1] \leftarrow A[i]$
                $i \leftarrow i-1$
        $A[i+1] \leftarrow$ key
```

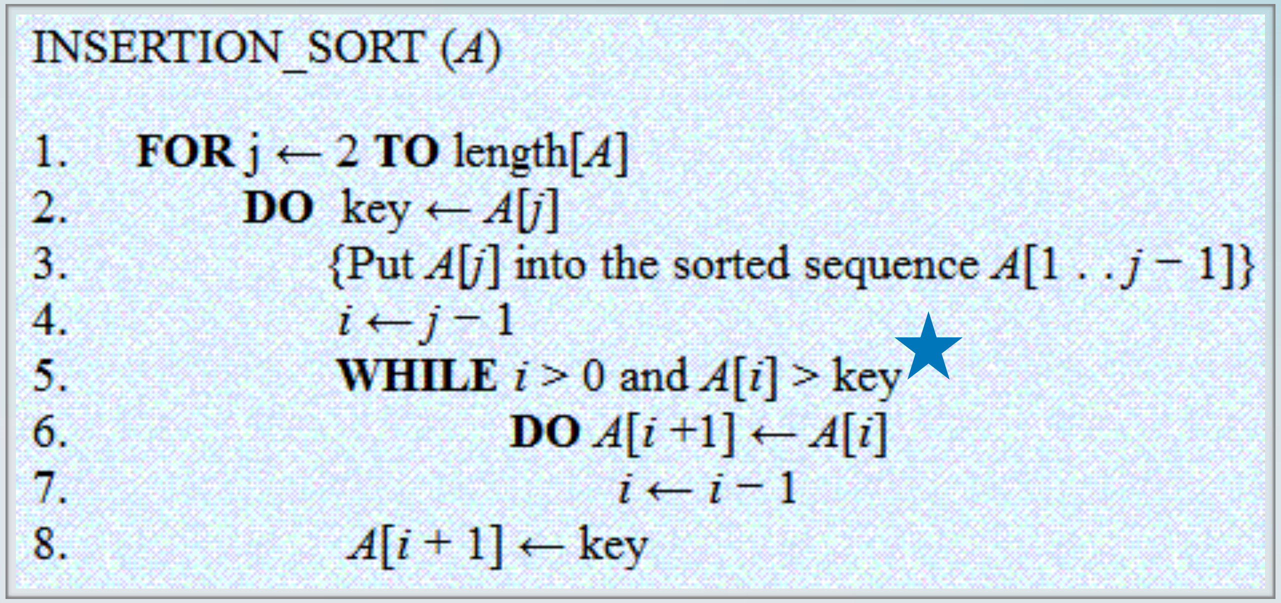
- The algorithm maintains a sorted array in each iteration (each time the for loop is executed).



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 4 | 6 | 19 | 10 | 1 | 17 | 14 | 21 | 24 |

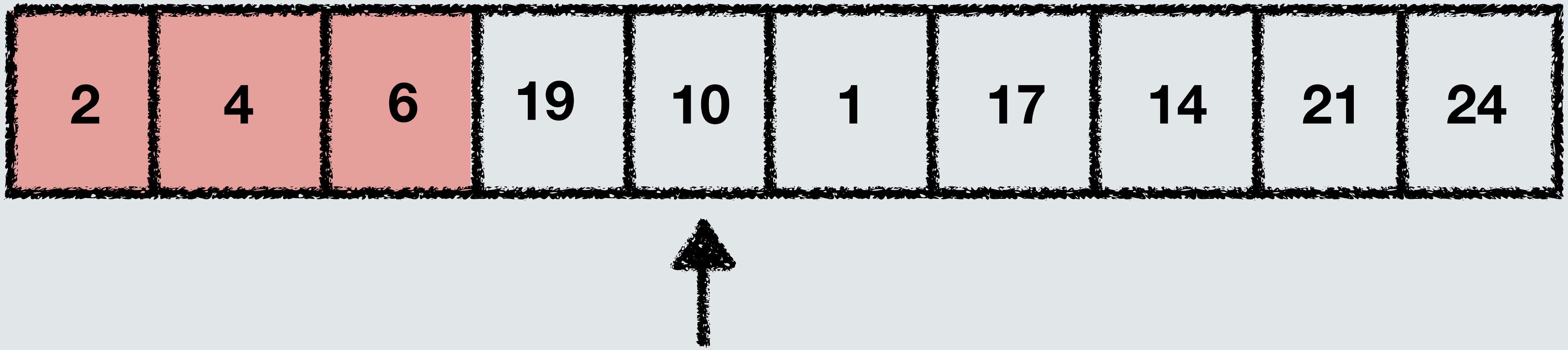
still in the while loop

# Describing algorithms: Pseudocode



```
INSERTION_SORT (A)

FOR j ← 2 TO length[A]
    DO key ← A[j]
        {Put A[j] into the sorted sequence A[1 . . j − 1]}
        i ← j − 1
        WHILE i > 0 and A[i] > key ★
            DO A[i +1] ← A[i]
                i ← i − 1
        A[i + 1] ← key
```

- The algorithm maintains a sorted array in each iteration (each time the for loop is executed).



still in the while loop

j=5

# Algorithmic techniques

- Brute force.
- Divide and Conquer.
- Greedy.
- Dynamic Programming.
- Integer linear program relaxation and rounding.
- Competitive analysis.
- Branch and Bound.

# Types of algorithms

- Searching algorithms.
- Sorting algorithms.
- Graph algorithms.
- Approximation algorithms.
- Online algorithms.
- Randomised algorithms.
- Exponential-time algorithms.

# What should we expect from algorithms?

- **Correctness:** It computes the desired output.
- **Termination:** Eventually terminates (or with high probability).
- **Efficiency:**
  - The algorithm runs *fast* and/or uses *limited memory.*
  - The algorithm produces a “good enough” outcome.

# Correctness

- Let’s look at the **InsertionSort** algorithm for sorting n numbers.
- Is it correct? Does it always produce a sorted sequence?
- Certainly seems to be the case, *intuitively*.
- How do we prove it *formally?*

# Loop invariance

# Loop invariance

- A loop invariant is a property that holds with respect to the loops executed by the algorithm.

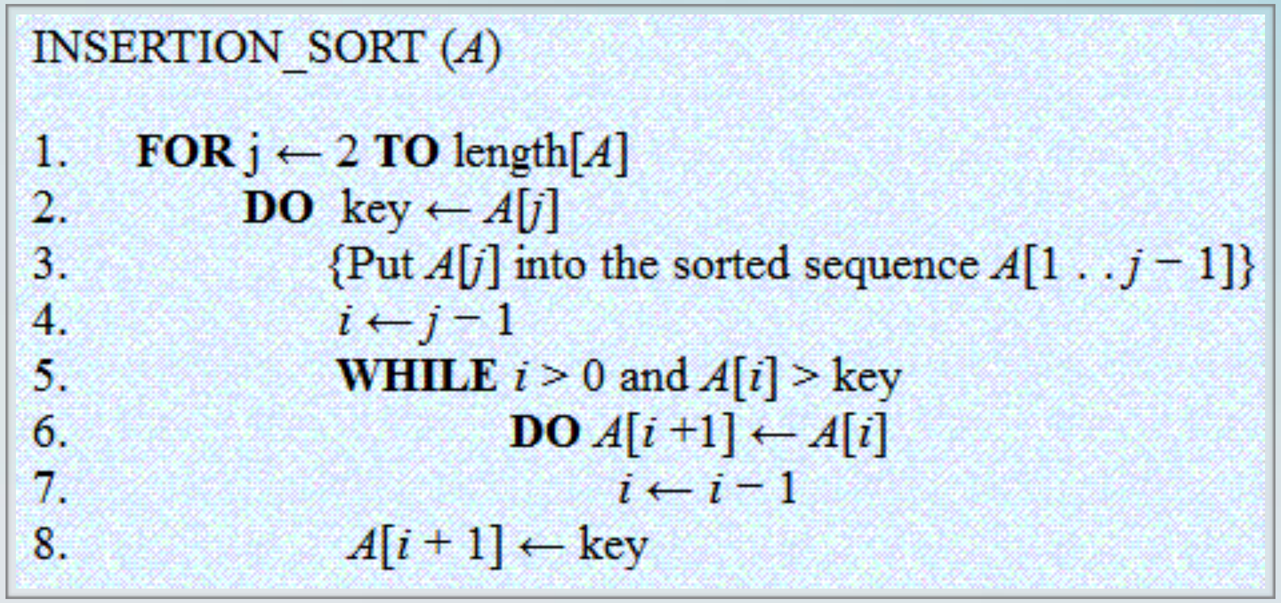
# Loop invariance

- A loop invariant is a property that holds with respect to the loops executed by the algorithm.
- For a loop invariant, we must show:
  - **Initialisation:** 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 for correctness.

# Loop invariance

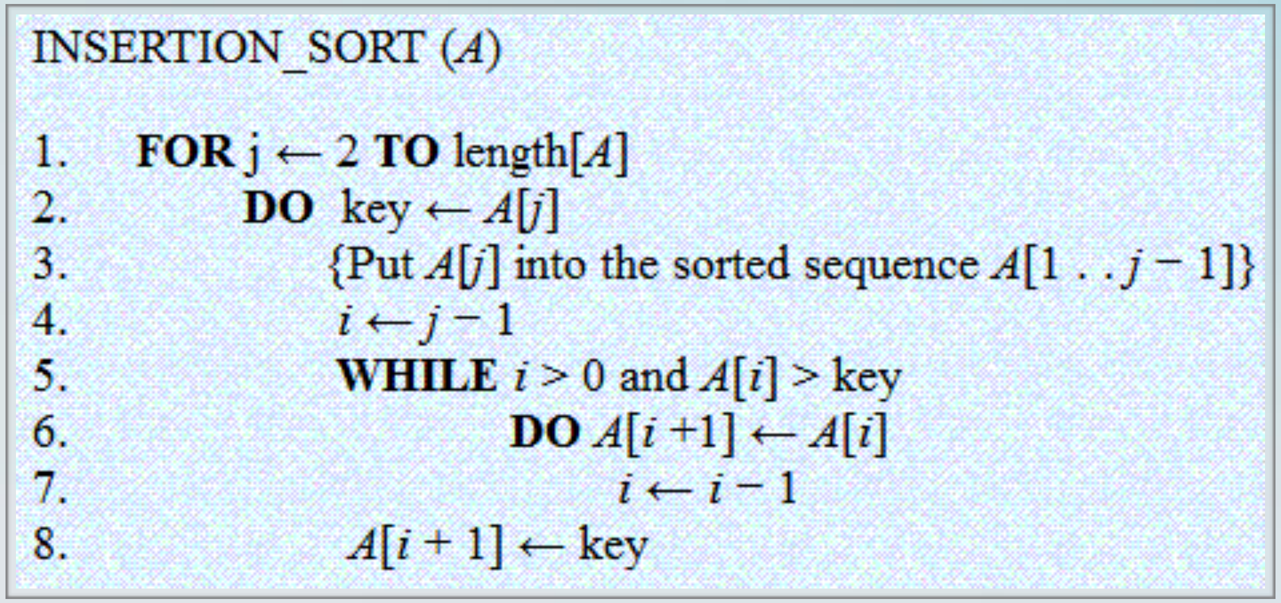
- A loop invariant is a property that holds with respect to the loops executed by the algorithm.
- For a loop invariant, we must show:
  - **Initialisation:** It is true prior to the first iteration of the loop.
  - **Maintenance:** If it is true before an iteration of the loop, it remains true before the next iteration.
  - **Termination:** When the loop terminates, the invariant gives us a useful property for correctness.
- Quite reminiscent of mathematical induction.

# Loop invariance for InsertionSort



```
INSERTION_SORT ($A$)

FOR j ← 2 TO length[$A$]
    DO key ← $A[j]$
        {Put $A[j]$ into the sorted sequence $A[1 .. j - 1]$}
        $i \leftarrow j - 1$
        WHILE $i > 0$ and $A[i] >$ key
            DO $A[i+1] \leftarrow A[i]$
                $i \leftarrow i - 1$
        $A[i + 1] \leftarrow$ key
```

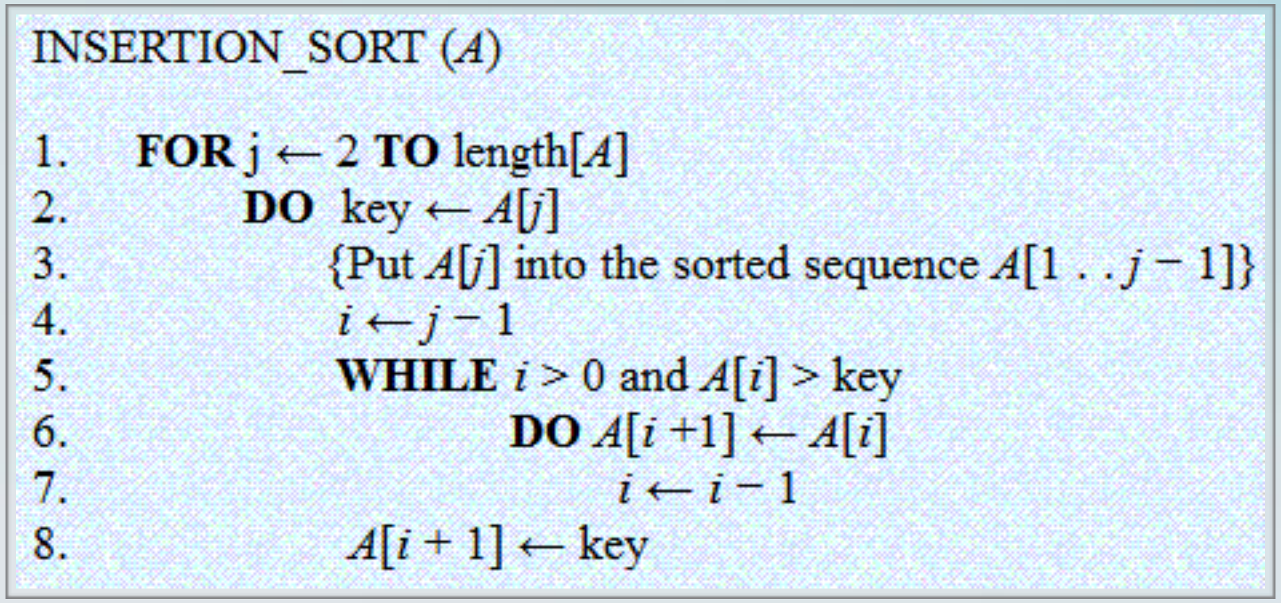
# Loop invariance for InsertionSort



```
INSERTION_SORT ($A$)

FOR j ← 2 TO length[$A$]
    DO key ← $A[j]$
        {Put $A[j]$ into the sorted sequence $A[1 . . j − 1]$}
        $i ← j − 1$
        WHILE $i > 0$ and $A[i]$ > key
            DO $A[i +1] ← A[i]$
                $i ← i − 1$
        $A[i + 1]$ ← key
```

- **Loop invariant:** The subarray A[1,…,j-1] consists of the elements originally in A[1,…,j-1] but in shorted order.

# Loop invariance for InsertionSort

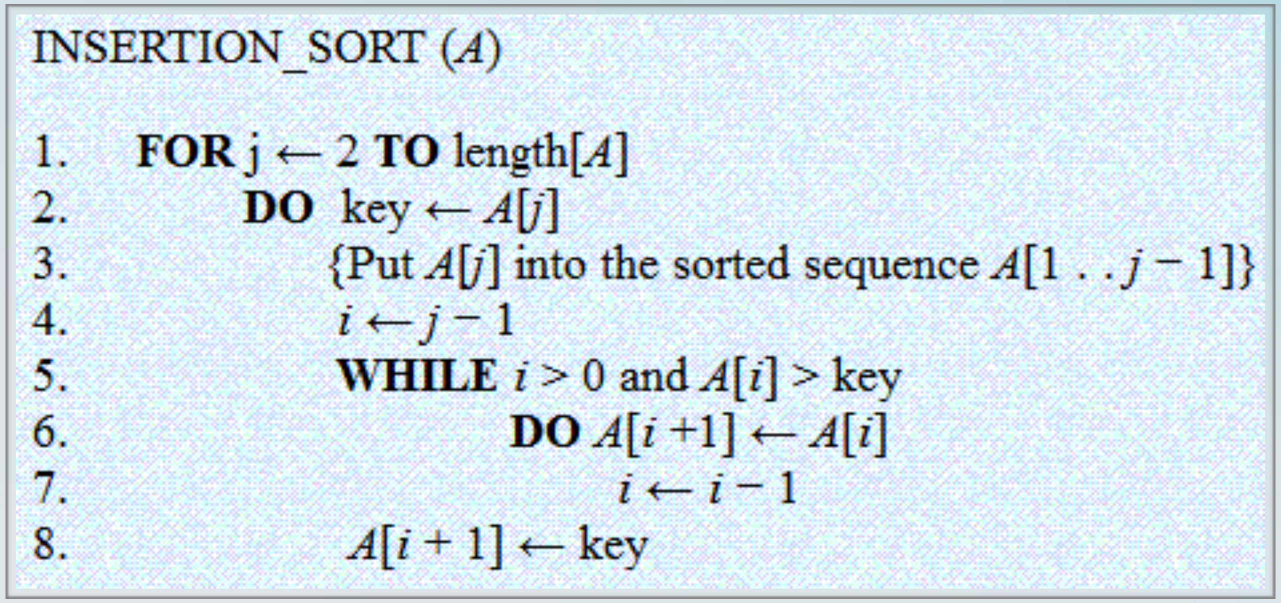


```
INSERTION_SORT ($A$)

FOR j $\leftarrow$ 2 TO length[$A$]
        DO  key $\leftarrow A[j]$
              {Put $A[j]$ into the sorted sequence $A[1\,.\,.\,j-1]$}
              $i \leftarrow j-1$
              WHILE $i > 0$ and $A[i] >$ key
                        DO $A[i+1] \leftarrow A[i]$
                              $i \leftarrow i-1$
              $A[i+1] \leftarrow$ key
```

- **Loop invariant:** The subarray A[1,…,j-1] consists of the elements originally in A[1,…,j-1] but in shorted order.

- **Initialisation:** Before the first iteration, the subarray is A[1], which contains the first element and is trivially sorted.

# Loop invariance for InsertionSort



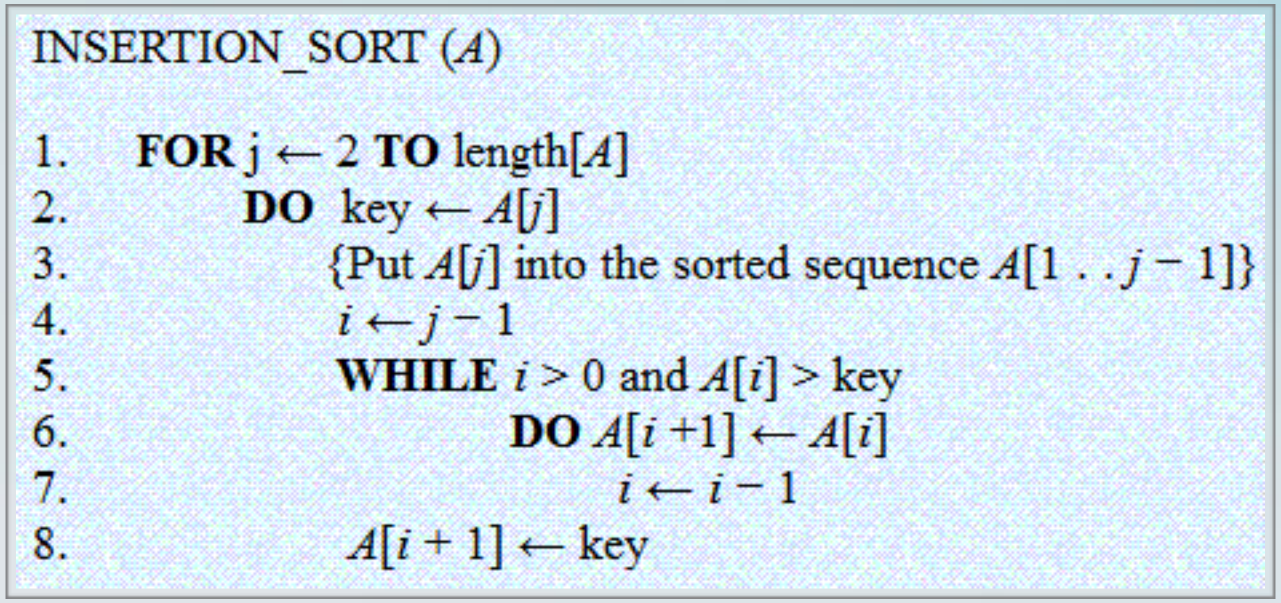
```
INSERTION_SORT ($A$)

FOR j ← 2 TO length[$A$]
    DO key ← $A[j]$
        {Put $A[j]$ into the sorted sequence $A[1 . . j - 1]$}
        $i ← j - 1$
        WHILE $i > 0$ and $A[i] >$ key
            DO $A[i + 1] ← A[i]$
                $i ← i - 1$
        $A[i + 1] ←$ key
```

- **Loop invariant:** The subarray A[1,…,j-1] consists of the elements originally in A[1,…,j-1] but in shorted order.

- **Initialisation:** Before the first iteration, the subarray is A[1], which contains the first element and is trivially sorted.

- **Maintenance:** We move A[j-1], A[j-2], A[j-3], … by one position to the right, until we find the proper position for A[j]. The subarray A[1,…,j] contains the original elements and it is sorted.

# Loop invariance for InsertionSort



```
INSERTION_SORT ($A$)

FOR j $\leftarrow$ 2 TO length[$A$]
        DO key $\leftarrow$ $A[j]$
            {Put $A[j]$ into the sorted sequence $A[1\,..\,j-1]$}
            $i \leftarrow j - 1$
            WHILE $i > 0$ and $A[i] >$ key
                    DO $A[i+1] \leftarrow A[i]$
                        $i \leftarrow i - 1$
            $A[i+1] \leftarrow$ key
```

- **Loop invariant:** The subarray A[1,…,j-1] consists of the elements originally in A[1,…,j-1] but in shorted order.

- **Initialisation:** Before the first iteration, the subarray is A[1], which contains the first element and is trivially sorted.

- **Maintenance:** We move A[j-1], A[j-2], A[j-3], … by one position to the right, until we find the proper position for A[j]. The subarray A[1,…,j] contains the original elements and it is sorted.

- **Termination:** Termination happens when length[A] is reached, so the counter is j = n+1. The loop invariant for j = n+1 is the sorted sequence of the n numbers.

# Running Time

- Different computers have different speeds.
- **Random Access Machine (RAM) model.**
- Instructions:
  - Arithmetic (add, subtract, multiply, etc).
  - Data movement (load, store, copy, etc).
  - Control (branch, subroutine call, return, etc).
- Each instruction is carried out in constant time.
- We can count the number of instructions, or the number of steps.

# Example: Running Time of LinearSearch

- Find if a number **x** exists in an **array** of **sorted numbers**.

**10**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 4 | 6 | 10 | 14 | 17 | 19 | 21 | 24 |

# Example: Running Time of LinearSearch

- Find if a number **x** exists in an **array** of **sorted numbers**.

**10**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 4 | 6 | 10 | 14 | 17 | 19 | 21 | 24 |

- We read through the array until we find the number.

# Example: Running Time of LinearSearch

- Find if a number **x** exists in an **array** of **sorted numbers**.

**10**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 4 | 6 | 10 | 14 | 17 | 19 | 21 | 24 |

- We read through the array until we find the number.
- For each element, we make a comparison.

# Example: Running Time of LinearSearch

- Find if a number **x** exists in an **array** of **sorted numbers**.

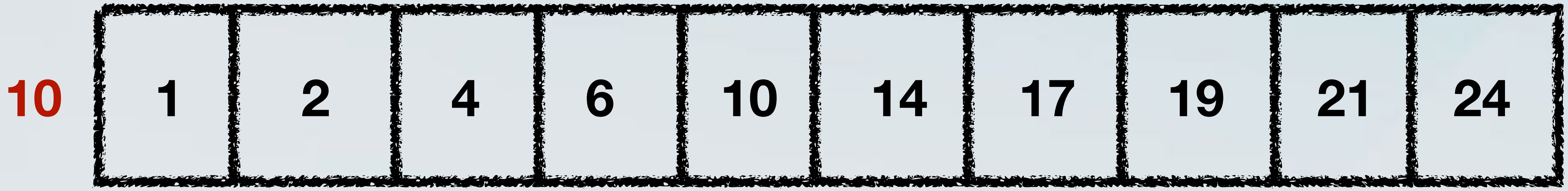
**10**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 4 | 6 | 10 | 14 | 17 | 19 | 21 | 24 |

- We read through the array until we find the number.
- For each element, we make a comparison.
- We need to initialise counters and write a for loop.

# Example: Running Time of LinearSearch

- Find if a number **x** exists in an **array** of **sorted numbers**.

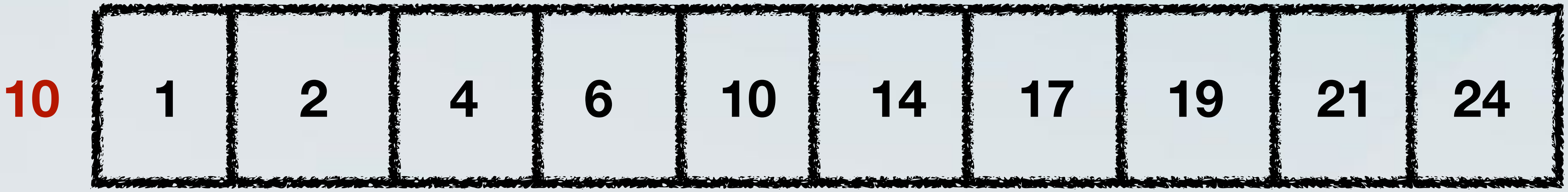


|  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|
| **10** | **1** | **2** | **4** | **6** | **10** | **14** | **17** | **19** | **21** | **24** |

- We read through the array until we find the number.
- For each element, we make a comparison.
- We need to initialise counters and write a for loop.
- Will certainly finish within c * n steps, where c is some large enough constant.

# Example: Running Time of LinearSearch

- Find if a number **x** exists in an **array** of **sorted numbers**.

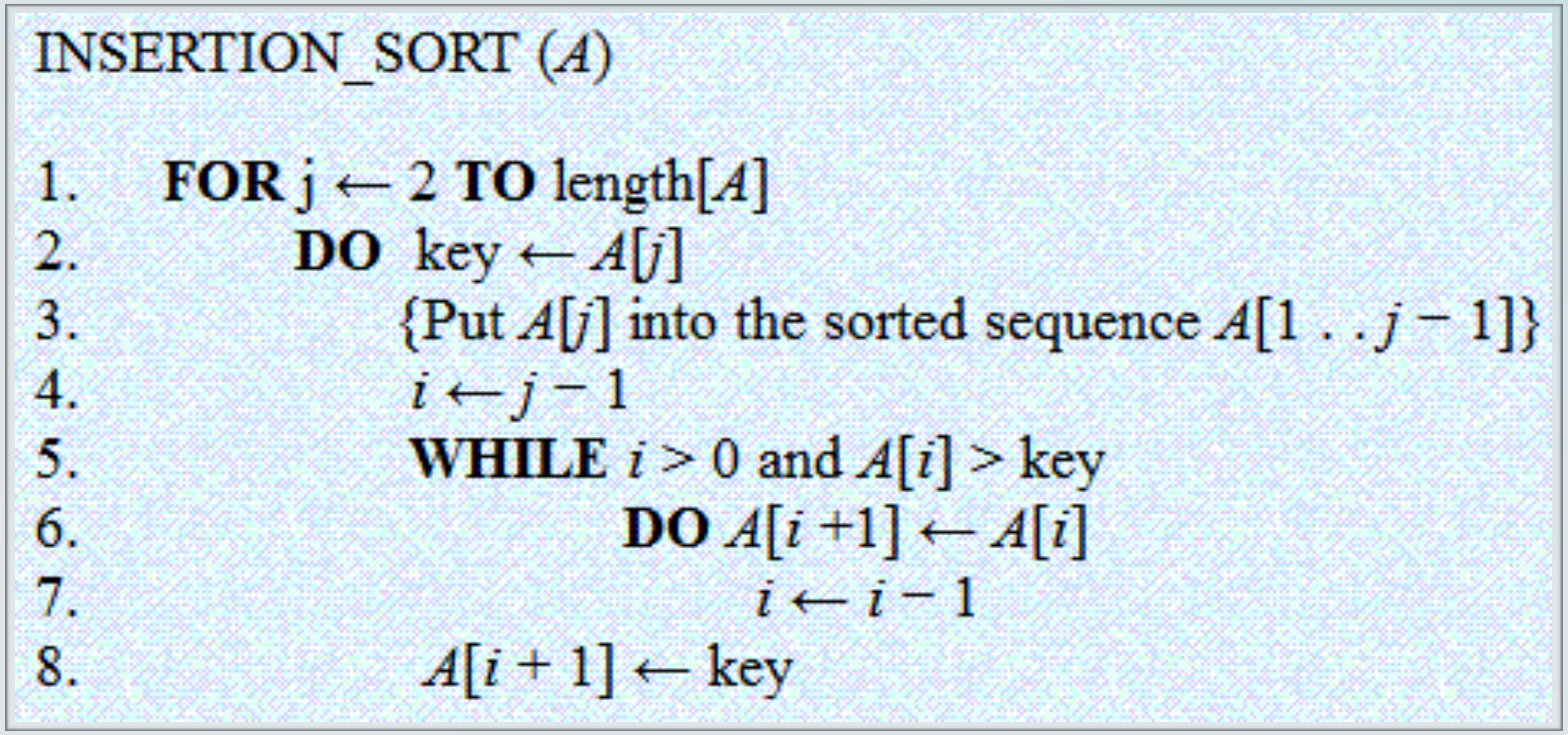


10

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 4 | 6 | 10 | 14 | 17 | 19 | 21 | 24 |

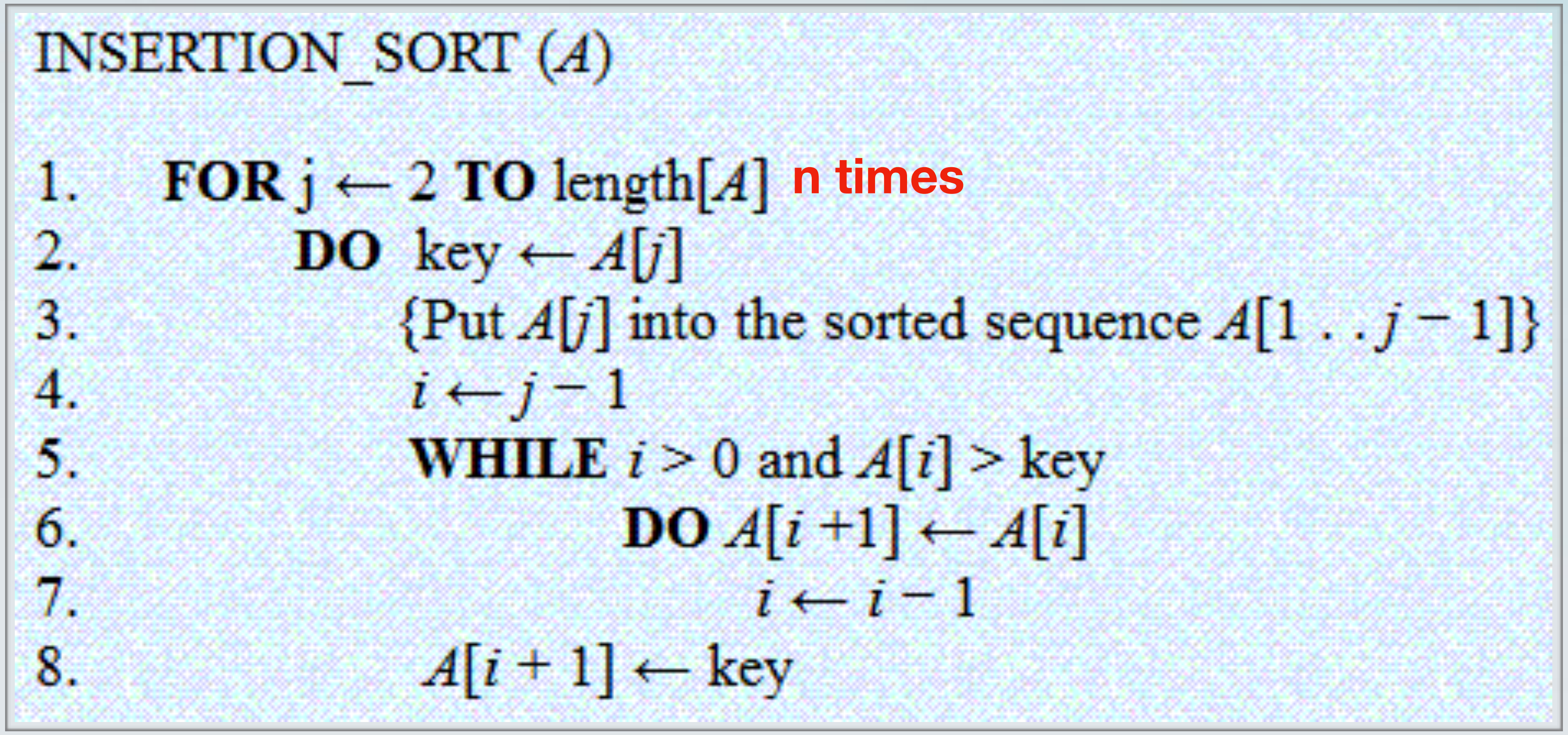
- We read through the array until we find the number.
- For each element, we make a comparison.
- We need to initialise counters and write a for loop.
- Will certainly finish within c * n steps, where c is some large enough constant.
- Does it require at least n steps in the worst case?

# Example: Running Time of InsertionSort



```
INSERTION_SORT ($A$)

FOR j ← 2 TO length[$A$]
    DO key ← $A[j]$
        {Put $A[j]$ into the sorted sequence $A[1 \,..\, j-1]$}
        $i \leftarrow j - 1$
        WHILE $i > 0$ and $A[i]$ > key
            DO $A[i+1] \leftarrow A[i]$
                $i \leftarrow i - 1$
        $A[i+1]$ ← key
```

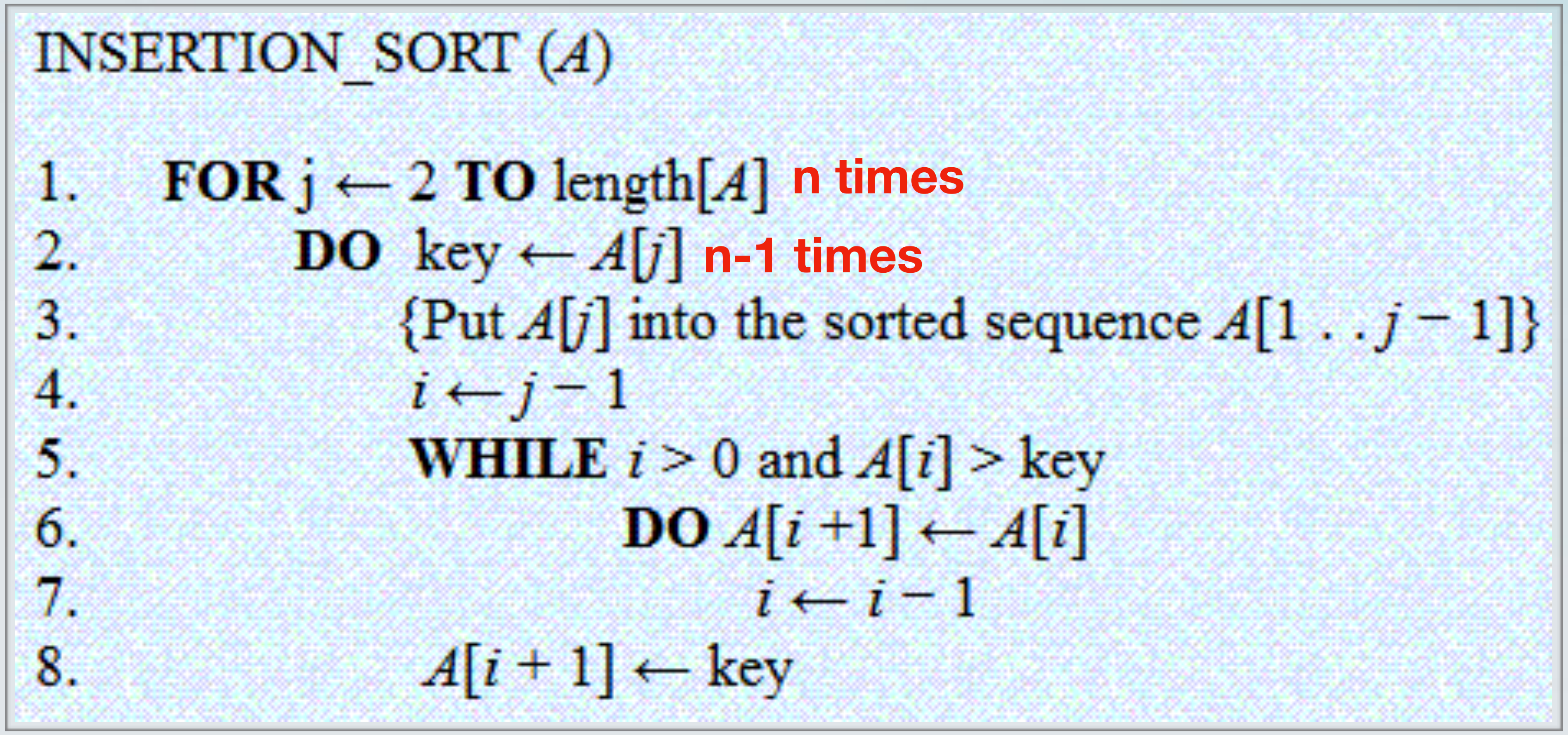
# Example: Running Time of InsertionSort



```
INSERTION_SORT (A)

FOR j ← 2 TO length[A]   n times
    DO key ← A[j]
       {Put A[j] into the sorted sequence A[1 . . j − 1]}
       i ← j − 1
       WHILE i > 0 and A[i] > key
             DO A[i +1] ← A[i]
                i ← i − 1
       A[i + 1] ← key
```

for loops, the tests are executed
one more time than the loop body

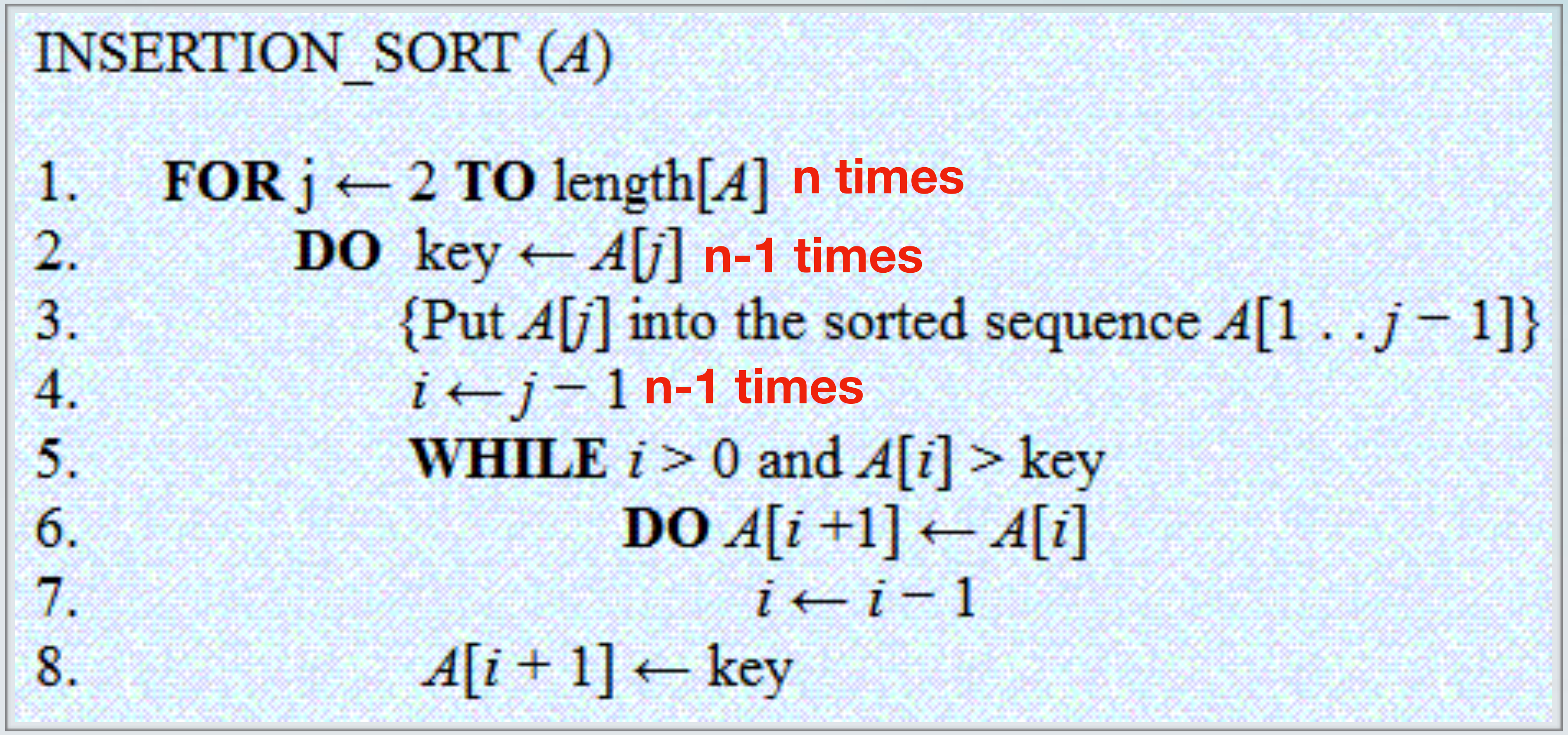
# Example: Running Time of InsertionSort



```
INSERTION_SORT (A)

FOR j ← 2 TO length[A]   n times
    DO key ← A[j]   n-1 times
        {Put A[j] into the sorted sequence A[1 . . j − 1]}
        i ← j − 1
        WHILE i > 0 and A[i] > key
            DO A[i +1] ← A[i]
                i ← i − 1
        A[i + 1] ← key
```

for loops, the tests are executed
one more time than the loop body

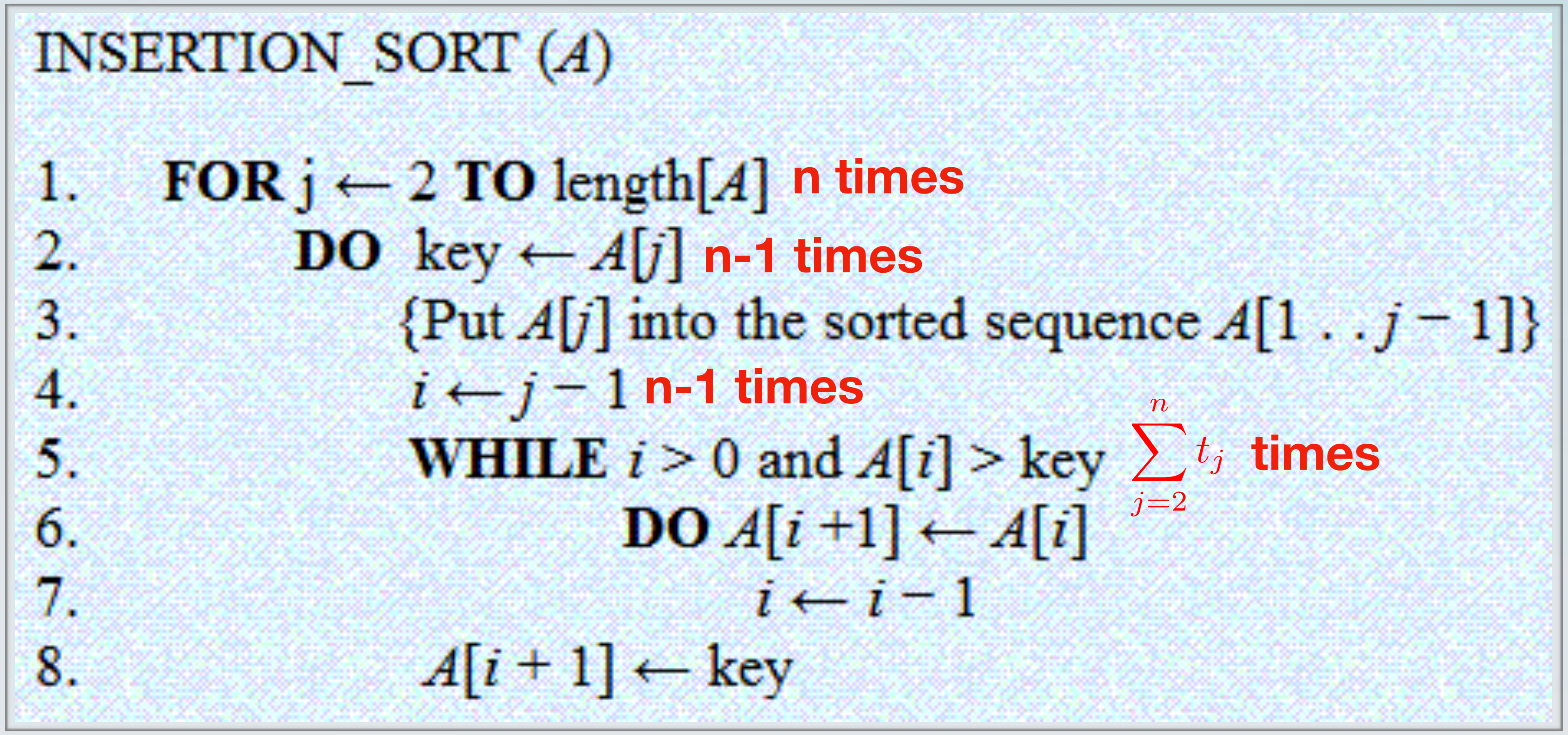
# Example: Running Time of InsertionSort



```
INSERTION_SORT ($A$)

FOR j $\leftarrow$ 2 TO length[$A$]  n times
    DO key $\leftarrow A[j]$  n-1 times
        {Put $A[j]$ into the sorted sequence $A[1 \,..\, j-1]$}
        $i \leftarrow j - 1$  n-1 times
        WHILE $i > 0$ and $A[i] >$ key
            DO $A[i+1] \leftarrow A[i]$
                $i \leftarrow i - 1$
        $A[i+1] \leftarrow$ key
```

for loops, the tests are executed
one more time than the loop body

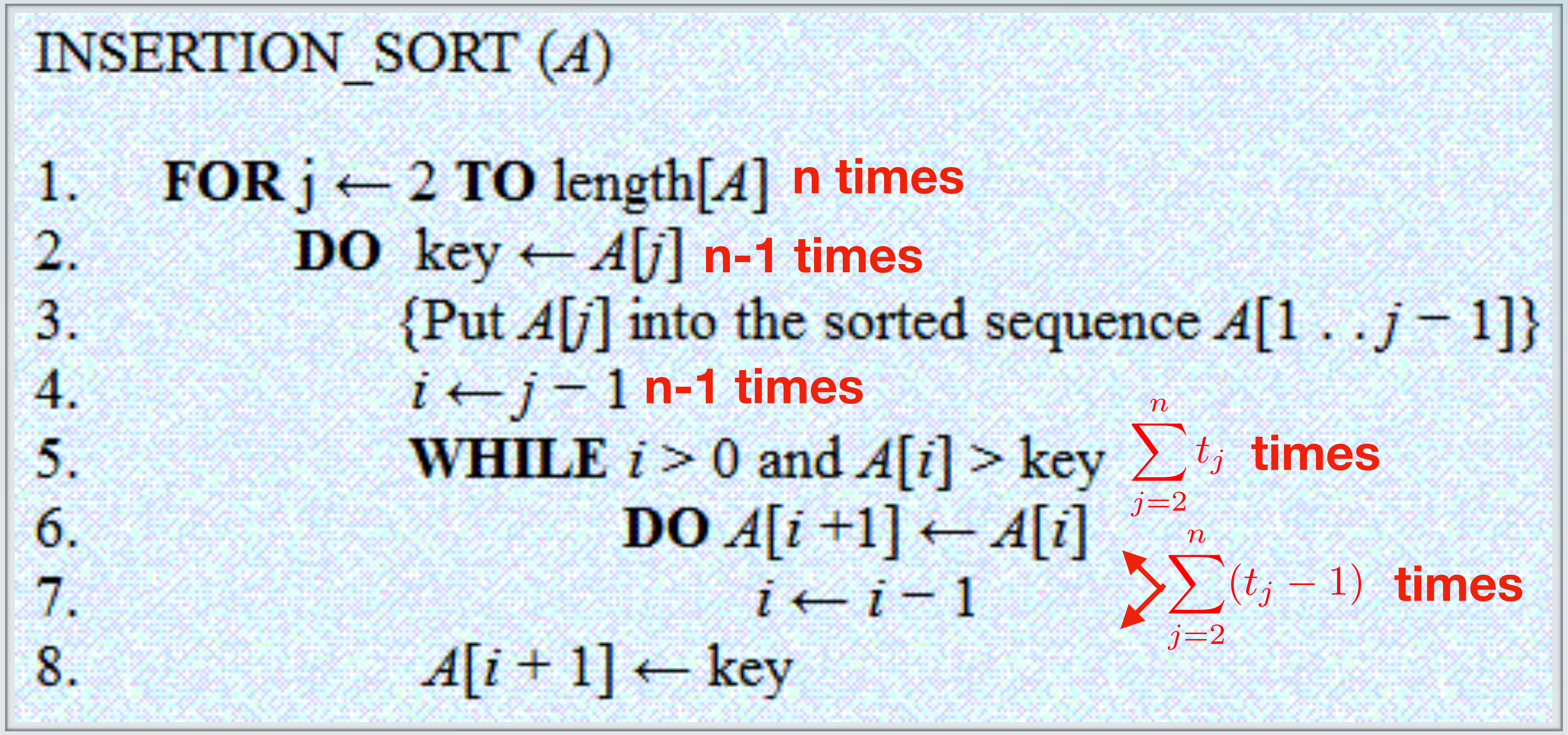
# Example: Running Time of InsertionSort



```
INSERTION_SORT (A)

FOR j ← 2 TO length[A]  n times
    DO  key ← A[j]  n-1 times
        {Put A[j] into the sorted sequence A[1 . . j − 1]}
        i ← j − 1  n-1 times
        WHILE i > 0 and A[i] > key  $\sum_{j=2}^{n} t_j$  times
            DO A[i +1] ← A[i]
               i ← i − 1
        A[i + 1] ← key
```

for loops, the tests are executed
one more time than the loop body

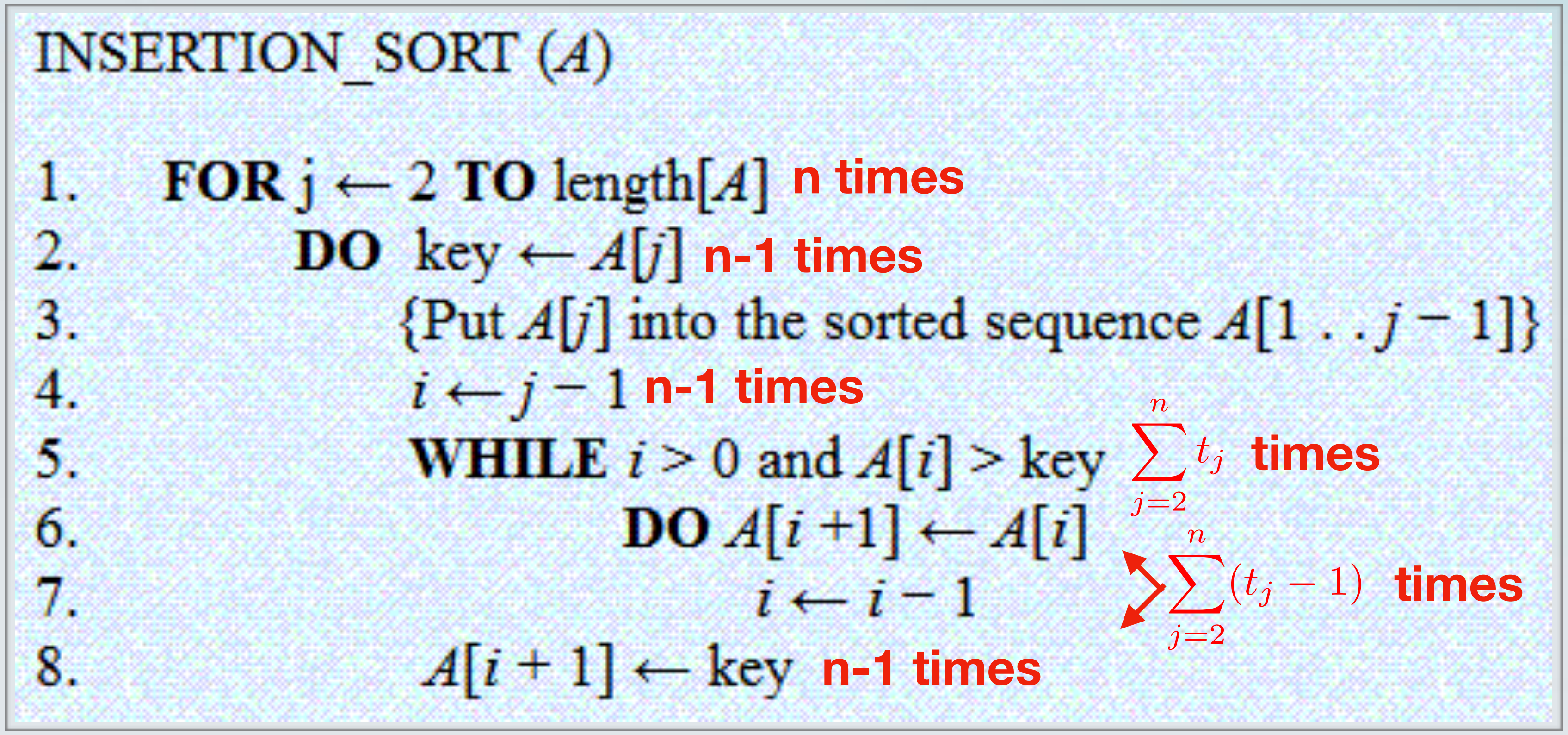
# Example: Running Time of InsertionSort



```
INSERTION_SORT (A)

FOR j ← 2 TO length[A]  n times
    DO key ← A[j]  n-1 times
       {Put A[j] into the sorted sequence A[1 . . j − 1]}
       i ← j − 1  n-1 times
       WHILE i > 0 and A[i] > key  $\sum_{j=2}^{n} t_j$  times
             DO A[i +1] ← A[i]
                i ← i − 1          $\sum_{j=2}^{n} (t_j - 1)$  times
       A[i + 1] ← key
```

for loops, the tests are executed
one more time than the loop body

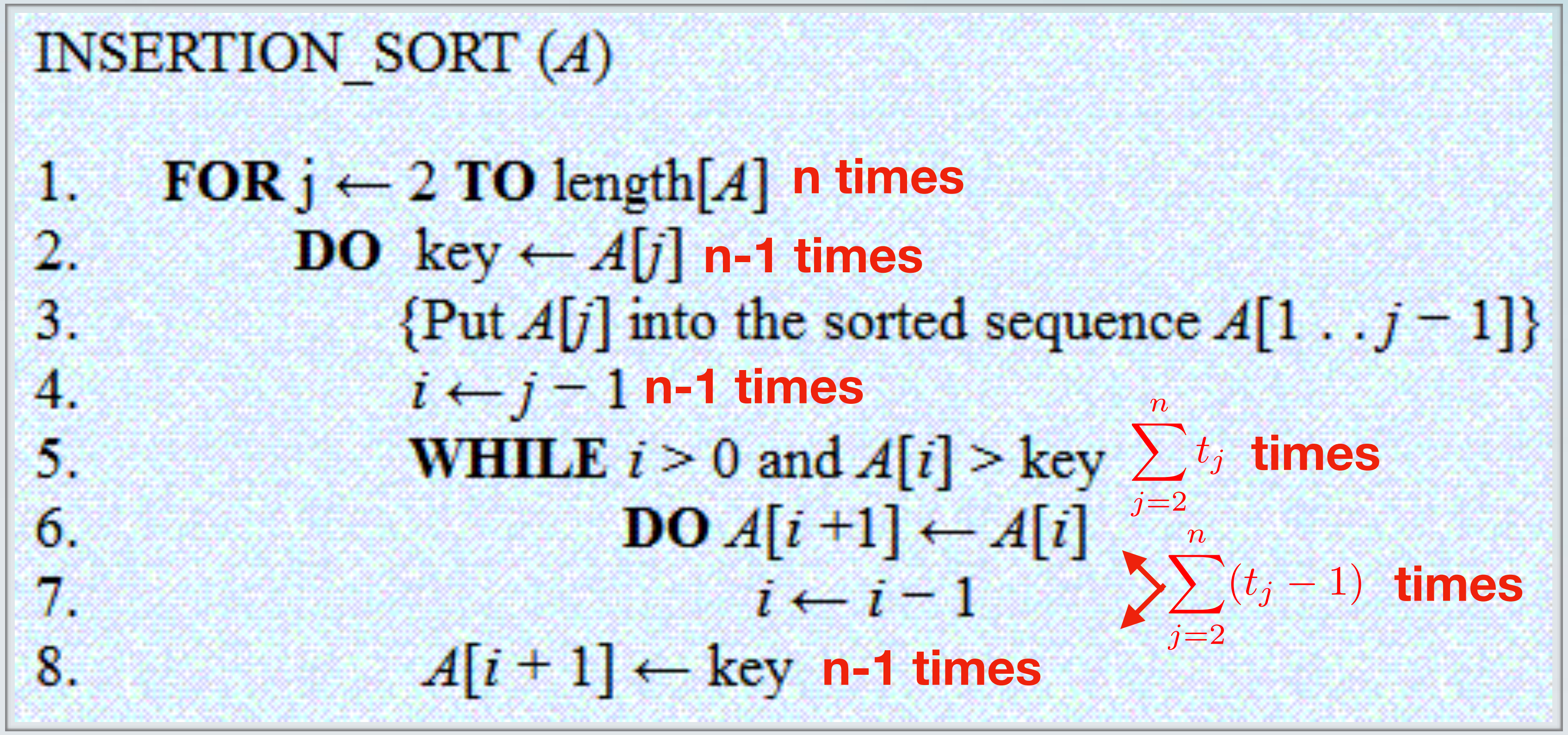
# Example: Running Time of InsertionSort



```
INSERTION_SORT ($A$)

FOR j $\leftarrow$ 2 TO length[$A$]    n times
    DO key $\leftarrow A[j]$    n-1 times
       {Put $A[j]$ into the sorted sequence $A[1\,..\,j-1]$}
       $i \leftarrow j - 1$    n-1 times
       WHILE $i > 0$ and $A[i] >$ key    $\sum_{j=2}^{n} t_j$ times
             DO $A[i+1] \leftarrow A[i]$    $\sum_{j=2}^{n} (t_j - 1)$ times
                $i \leftarrow i - 1$
       $A[i+1] \leftarrow$ key    n-1 times
```

for loops, the tests are executed
one more time than the loop body

# Example: Running Time of InsertionSort

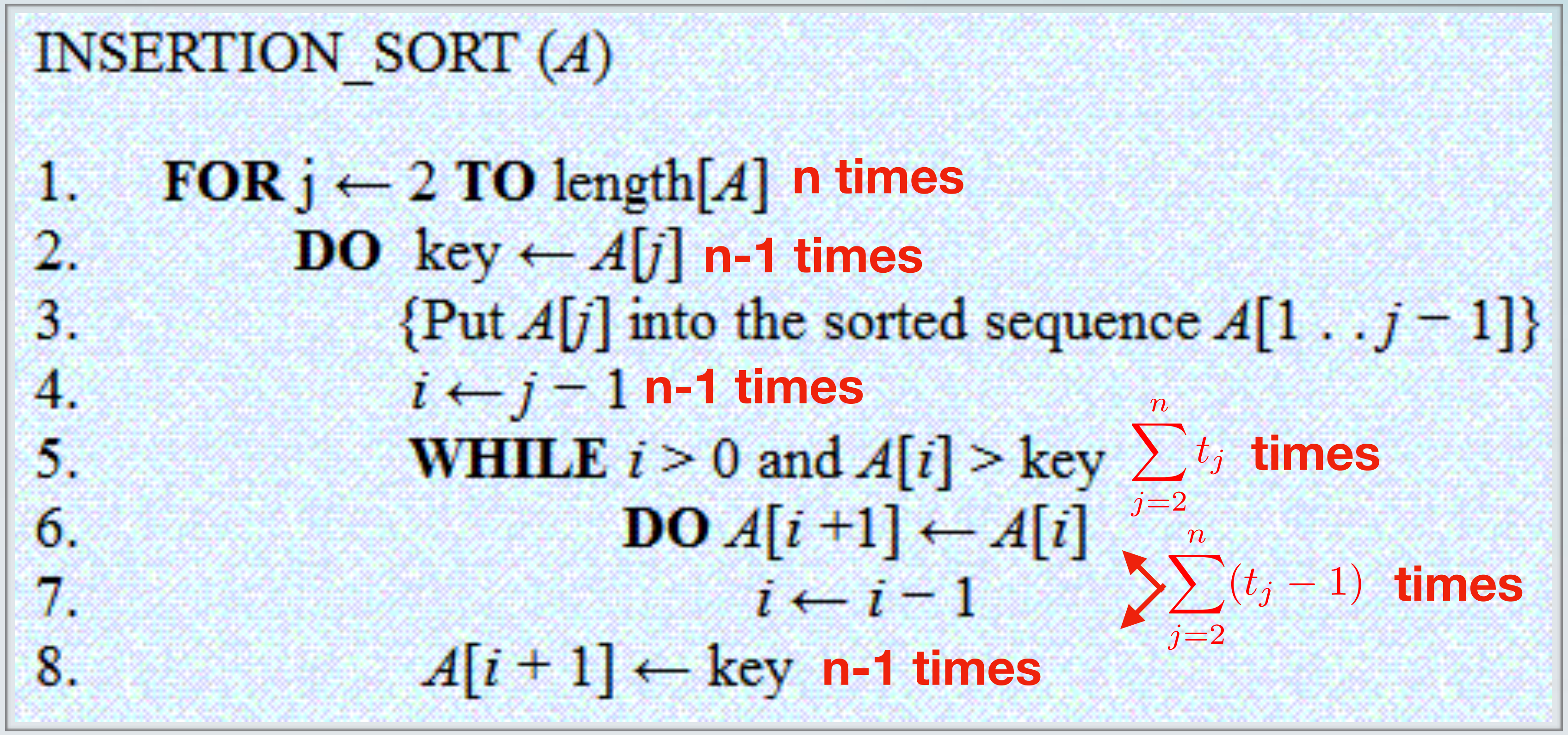


```
INSERTION_SORT (A)

FOR j ← 2 TO length[A] n times
    DO key ← A[j] n-1 times
       {Put A[j] into the sorted sequence A[1 . . j − 1]}
       i ← j − 1 n-1 times
       WHILE i > 0 and A[i] > key $\sum_{j=2}^{n} t_j$ times
             DO A[i +1] ← A[i]        } $\sum_{j=2}^{n} (t_j - 1)$ times
                i ← i − 1             }
       A[i + 1] ← key n-1 times
```

for loops, the tests are executed one more time than the loop body

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

# Example: Running Time of InsertionSort



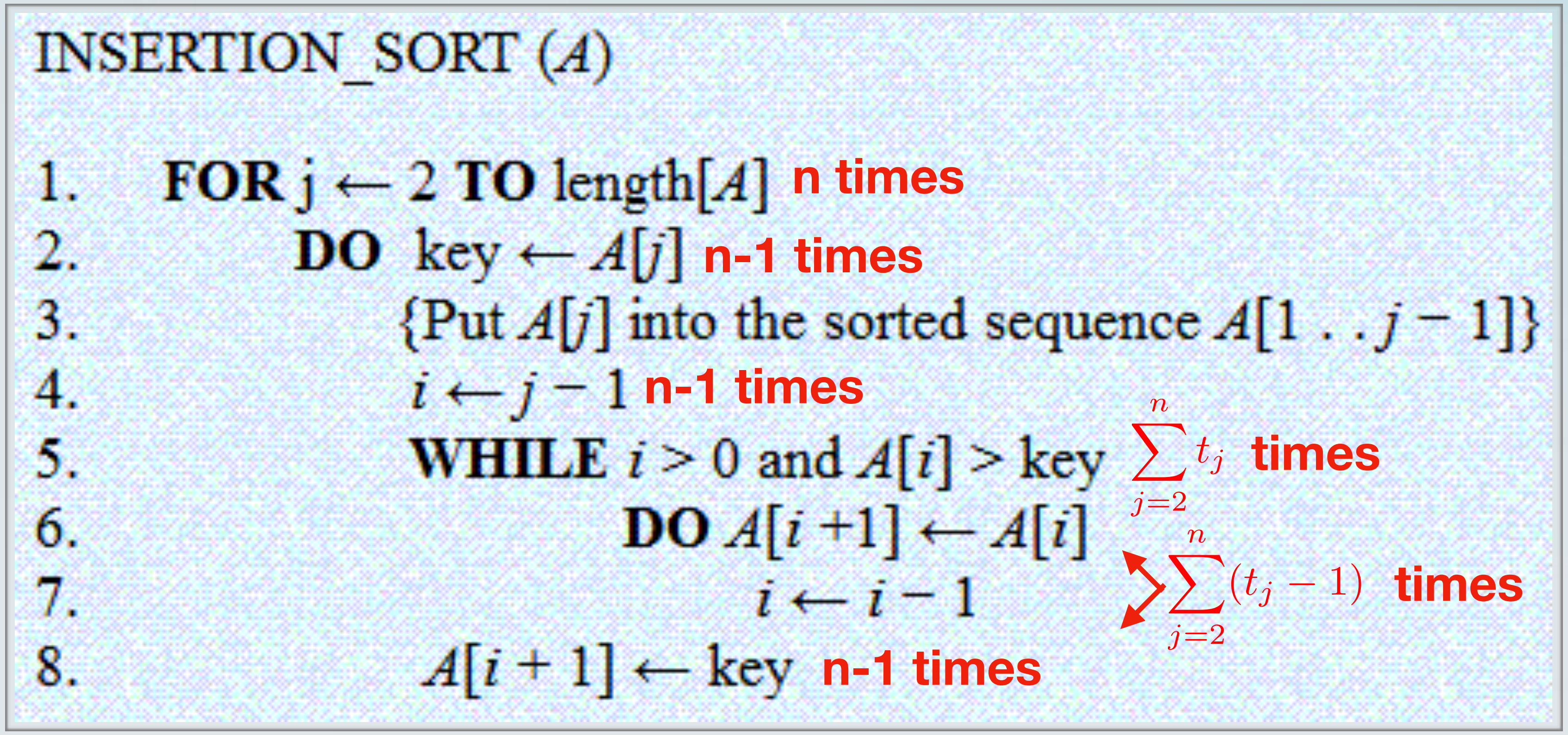
```
INSERTION_SORT ($A$)

FOR j ← 2 TO length[$A$]  n times
    DO key ← $A[j]$  n-1 times
        {Put $A[j]$ into the sorted sequence $A[1 \,..\, j-1]$}
        $i \leftarrow j-1$  n-1 times
        WHILE $i > 0$ and $A[i] >$ key  $\sum_{j=2}^{n} t_j$  times
            DO $A[i+1] \leftarrow A[i]$
                $i \leftarrow i-1$      $\sum_{j=2}^{n} (t_j - 1)$  times
        $A[i+1]$ ← key  n-1 times
```

for loops, the tests are executed
one more time than the loop body

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

Best case?

# Example: Running Time of InsertionSort



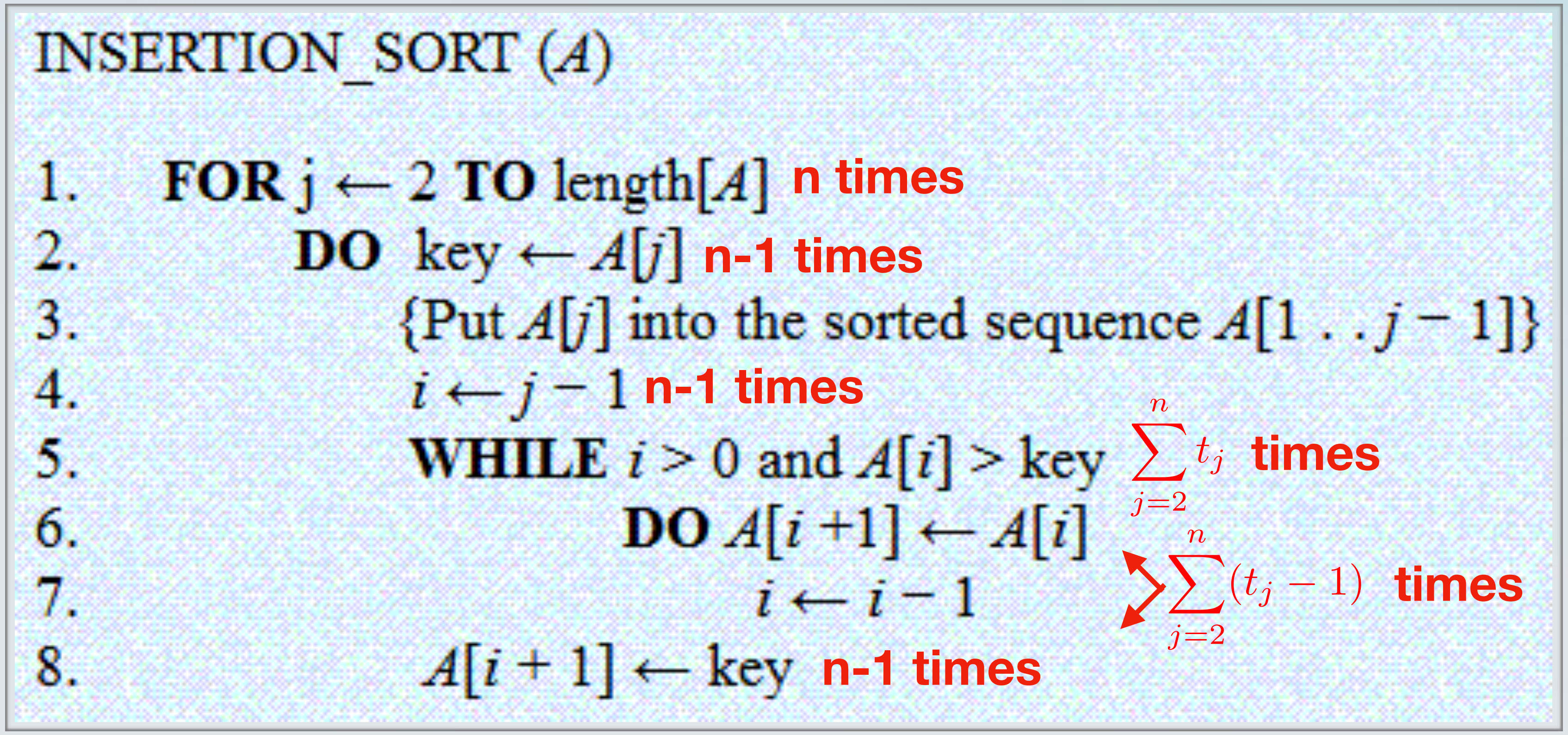
```
INSERTION_SORT ($A$)

FOR j ← 2 TO length[$A$]   n times
    DO key ← $A[j]$   n-1 times
        {Put $A[j]$ into the sorted sequence $A[1\,..\,j-1]$}
        $i \leftarrow j-1$   n-1 times
        WHILE $i > 0$ and $A[i] >$ key   $\sum_{j=2}^{n} t_j$ times
            DO $A[i+1] \leftarrow A[i]$
                $i \leftarrow i-1$   $\sum_{j=2}^{n} (t_j - 1)$ times
        $A[i+1] \leftarrow$ key   n-1 times
```

for loops, the tests are executed one more time than the loop body

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

Best case? **Sorted array,** $t_j = 1$

# Example: Running Time of InsertionSort



```
INSERTION_SORT ($A$)

FOR j $\leftarrow$ 2 TO length[$A$]   n times
    DO key $\leftarrow A[j]$   n-1 times
        {Put $A[j]$ into the sorted sequence $A[1 \, . \, . \, j-1]$}
        $i \leftarrow j-1$   n-1 times
        WHILE $i > 0$ and $A[i] >$ key   $\sum_{j=2}^{n} t_j$ times
            DO $A[i+1] \leftarrow A[i]$
                $i \leftarrow i-1$   $\sum_{j=2}^{n} (t_j - 1)$ times
        $A[i+1] \leftarrow$ key   n-1 times
```

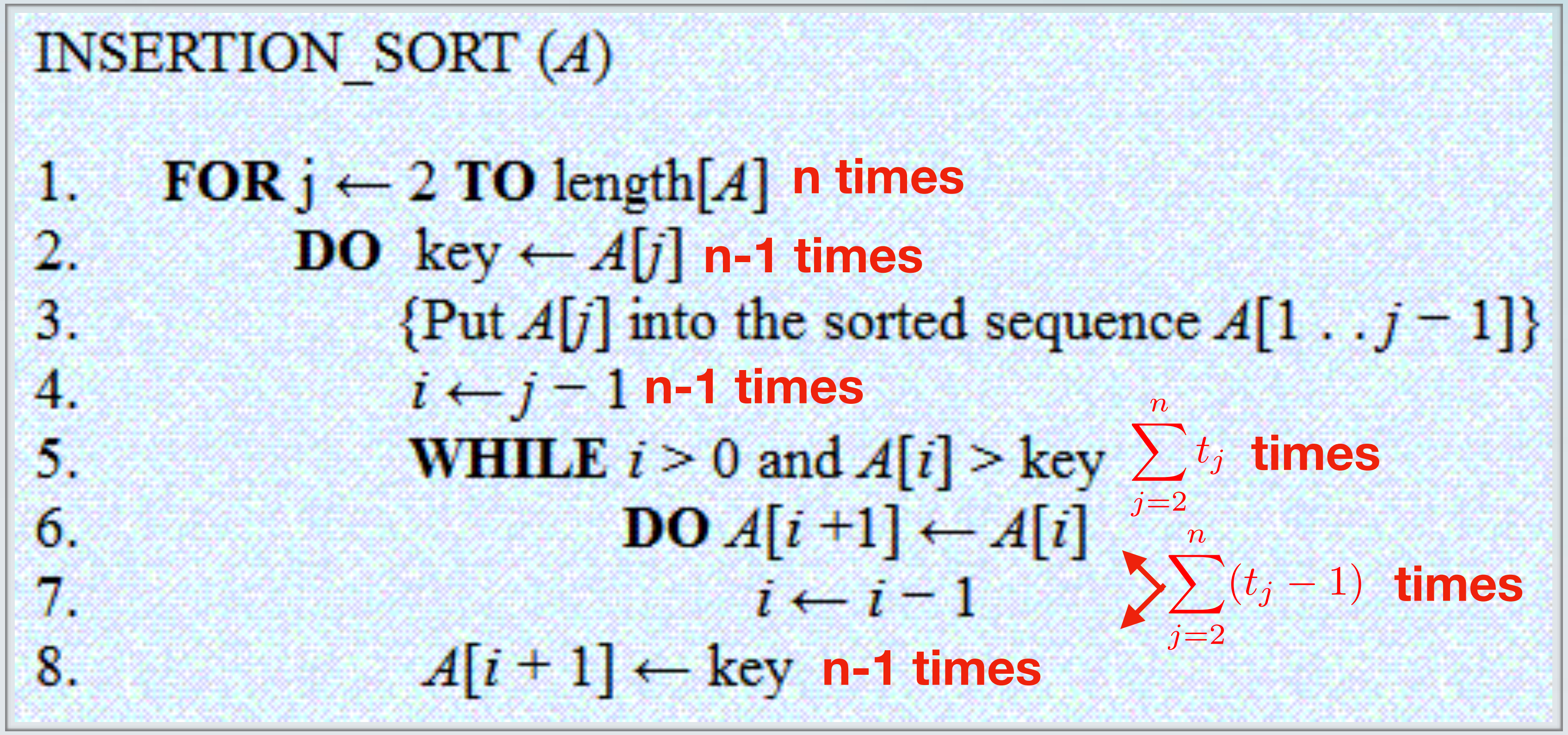
for loops, the tests are executed one more time than the loop body

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

Best case? **Sorted array,** $t_j = 1$

Worst case?

# Example: Running Time of InsertionSort



```
INSERTION_SORT ($A$)

FOR j ← 2 TO length[$A$] n times
    DO key ← $A[j]$ n-1 times
        {Put $A[j]$ into the sorted sequence $A[1..j-1]$}
        $i \leftarrow j-1$ n-1 times
        WHILE $i > 0$ and $A[i] >$ key $\sum_{j=2}^{n} t_j$ times
            DO $A[i+1] \leftarrow A[i]$
                $i \leftarrow i-1$ $\sum_{j=2}^{n} (t_j - 1)$ times
        $A[i+1]$ ← key n-1 times
```

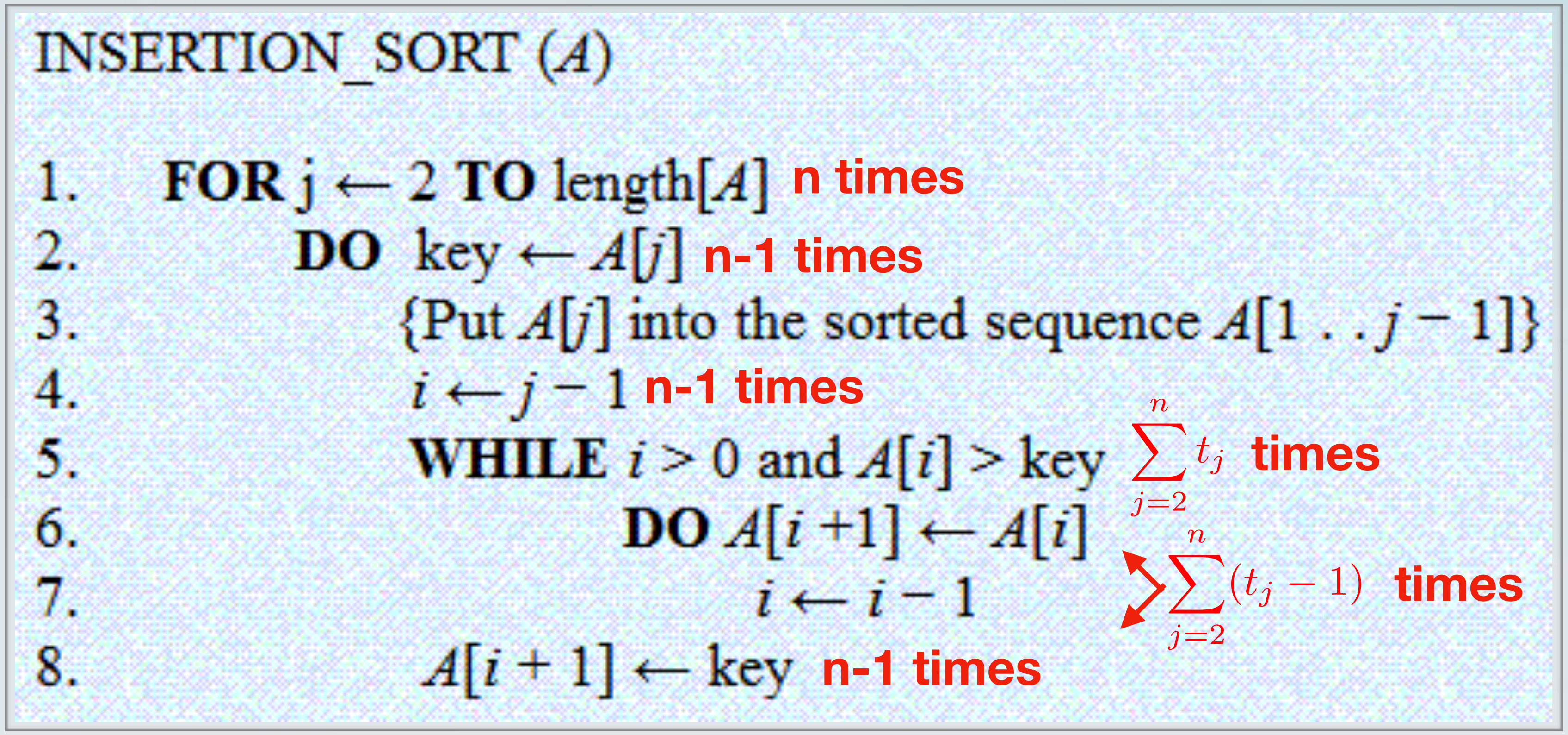
for loops, the tests are executed one more time than the loop body

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

Best case? **Sorted array,** $t_j = 1$

Worst case? **Reverse sorted array,** $t_j = j$

# Example: Running Time of InsertionSort



```
INSERTION_SORT (A)

FOR j ← 2 TO length[A]  n times
    DO key ← A[j]  n-1 times
       {Put A[j] into the sorted sequence A[1 . . j − 1]}
       i ← j − 1  n-1 times
       WHILE i > 0 and A[i] > key  $\sum_{j=2}^{n} t_j$  times
             DO A[i +1] ← A[i]      $\sum_{j=2}^{n} (t_j - 1)$  times
                i ← i − 1
       A[i + 1] ← key  n-1 times
```

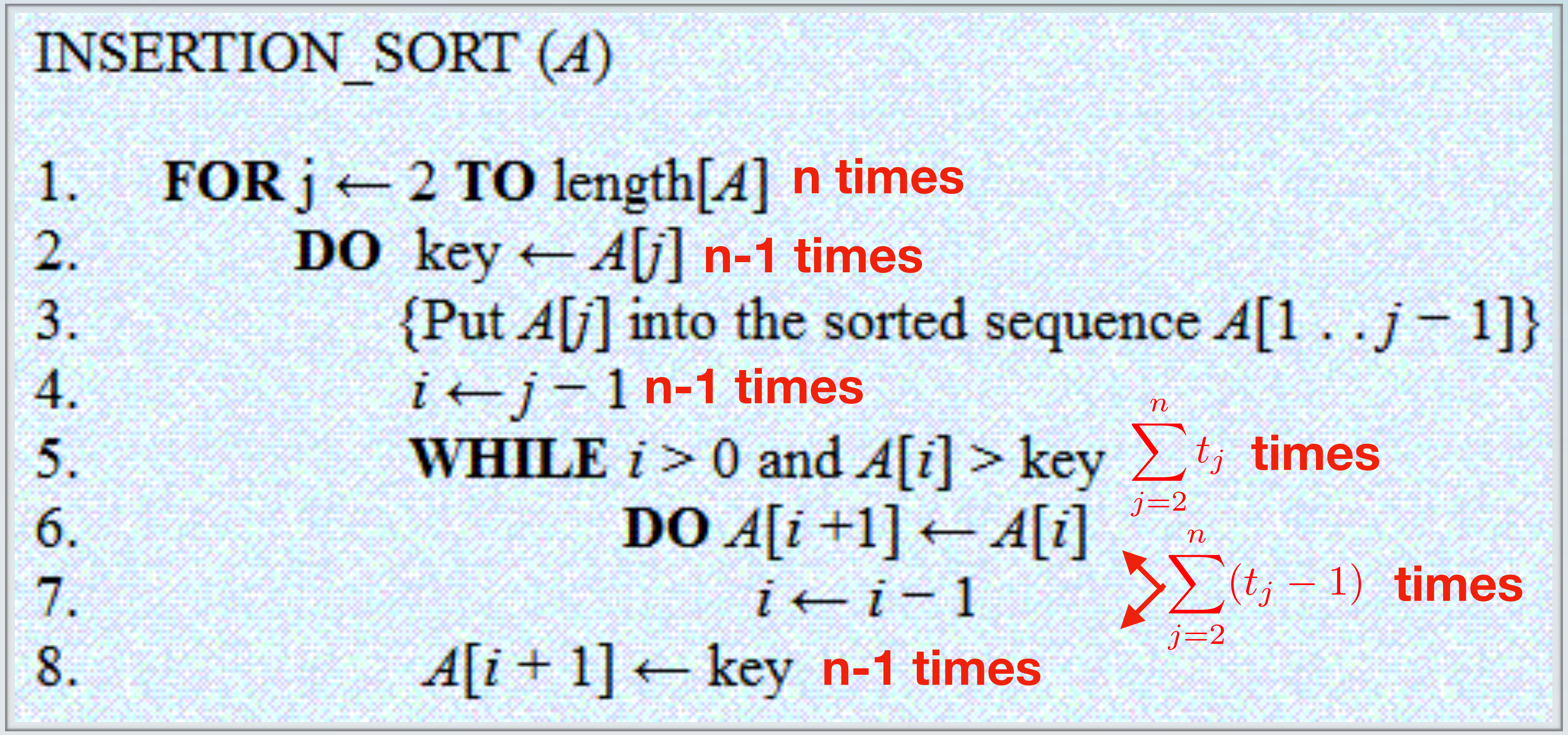
for loops, the tests are executed one more time than the loop body

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

Best case? **Sorted array,** $t_j = 1$ Bounded by some $cn$ for some constant c

Worst case? **Reverse sorted array,** $t_j = j$

# Example: Running Time of InsertionSort



```
INSERTION_SORT ($A$)

FOR j $\leftarrow$ 2 TO length[$A$]   n times
    DO key $\leftarrow A[j]$   n-1 times
        {Put $A[j]$ into the sorted sequence $A[1\,..\,j-1]$}
        $i \leftarrow j-1$   n-1 times
        WHILE $i > 0$ and $A[i] >$ key   $\sum_{j=2}^{n} t_j$ times
            DO $A[i+1] \leftarrow A[i]$
                $i \leftarrow i-1$   $\sum_{j=2}^{n} (t_j - 1)$ times
        $A[i+1] \leftarrow$ key   n-1 times
```

for loops, the tests are executed one more time than the loop body

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

Best case? **Sorted array,** $t_j = 1$ Bounded by some $cn$ for some constant c

Worst case? **Reverse sorted array,** $t_j = j$ Bounded by some $cn^2$ for some constant c

# Memory Usage

- Each memory cell can hold one element of the input.
- Total memory usage = Memory used to hold the input + extra memory used by the algorithm (auxiliary memory).
- What is the total and the auxiliary memory usage of **LinearSearch**?
- What is the total and the auxiliary memory usage of **InsertionSort**?

# Worst vs Best vs Average Case

# Worst vs Best vs Average Case

- **Convention:** When we say “the running time of Algorithm A”, we mean the worst-case running time, over all possible inputs to the algorithm.

# Worst vs Best vs Average Case

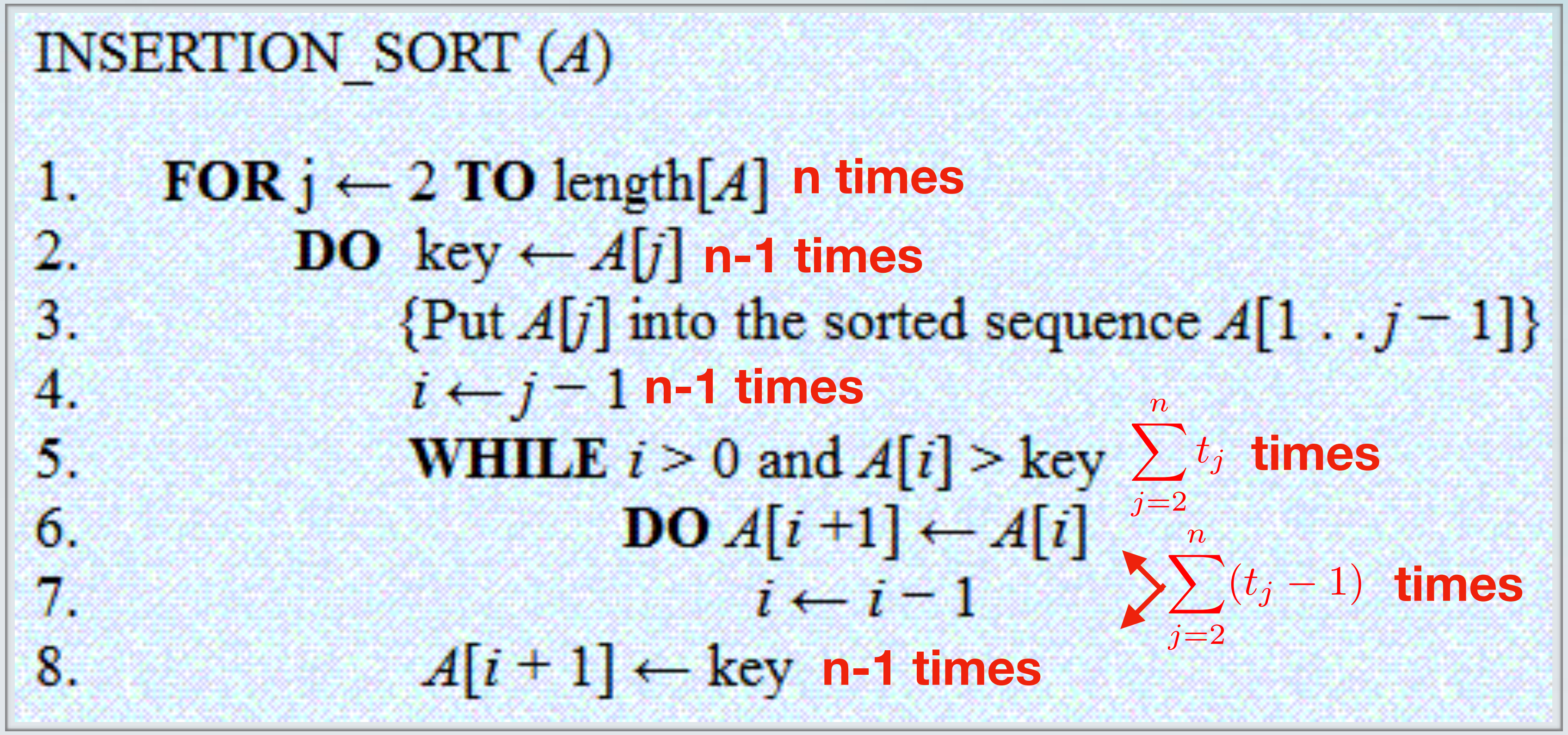
- **Convention:** When we say “the running time of Algorithm A”, we mean the worst-case running time, over all possible inputs to the algorithm.

- We can also measure the best-case running time, over all possible inputs to the problem.

# Worst vs Best vs Average Case

- **Convention:** When we say “the running time of Algorithm A”, we mean the worst-case running time, over all possible inputs to the algorithm.
- We can also measure the best-case running time, over all possible inputs to the problem.
- In between: average-case running time.
  - Running time of the algorithm on inputs which are chosen at random from some distribution.
  - The appropriate distribution depends on the application.
  - The analysis can be difficult.

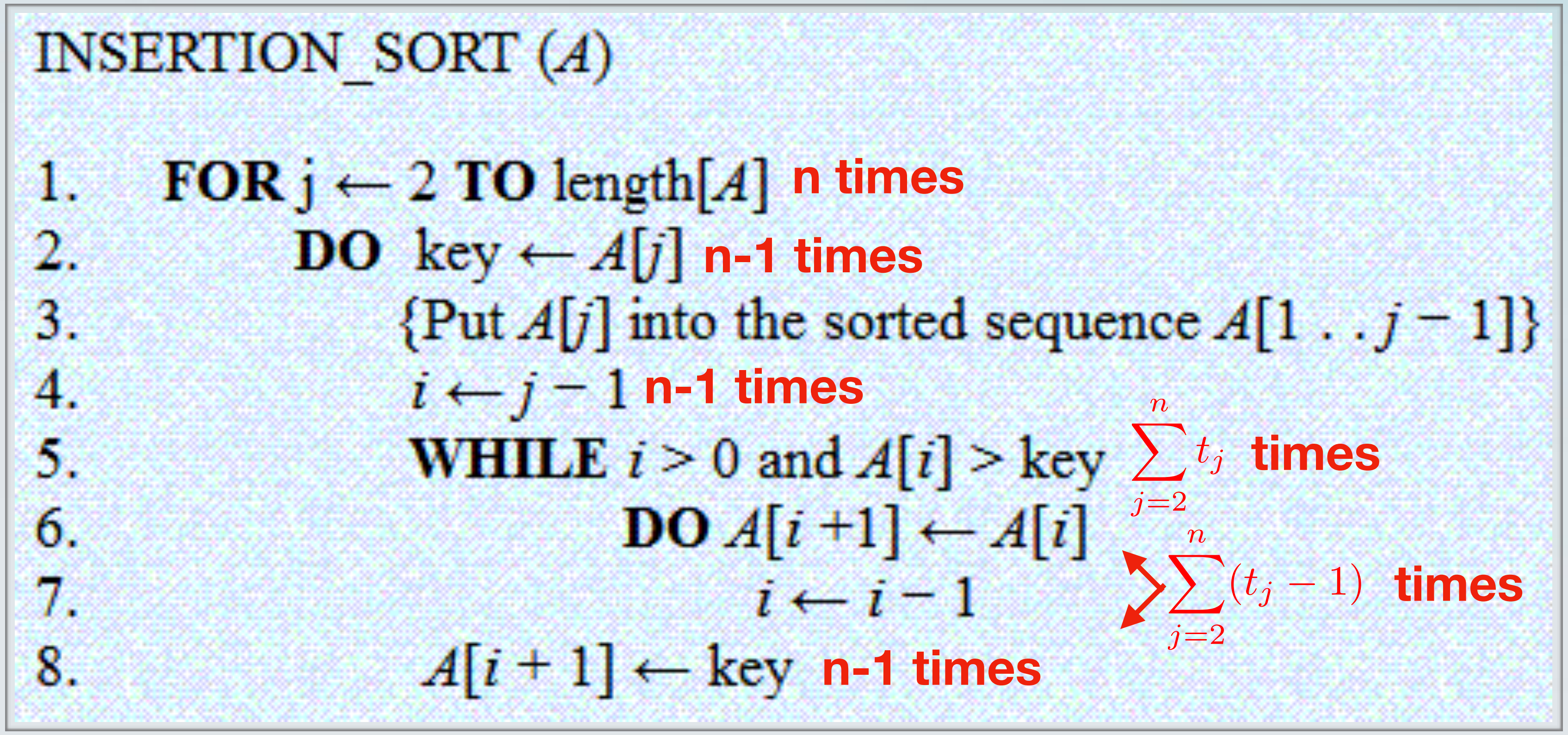
# Example: Average Running Time of InsertionSort



```
INSERTION_SORT ($A$)

FOR j ← 2 TO length[$A$]   n times
    DO key ← $A[j]$   n-1 times
        {Put $A[j]$ into the sorted sequence $A[1 \ldots j-1]$}
        $i \leftarrow j - 1$   n-1 times
        WHILE $i > 0$ and $A[i] >$ key   $\sum_{j=2}^{n} t_j$ times
            DO $A[i+1] \leftarrow A[i]$   $\sum_{j=2}^{n} (t_j - 1)$ times
                $i \leftarrow i - 1$
        $A[i+1] \leftarrow$ key   n-1 times
```

for loops, the tests are executed one more time than the loop body

# Example: Average Running Time of InsertionSort

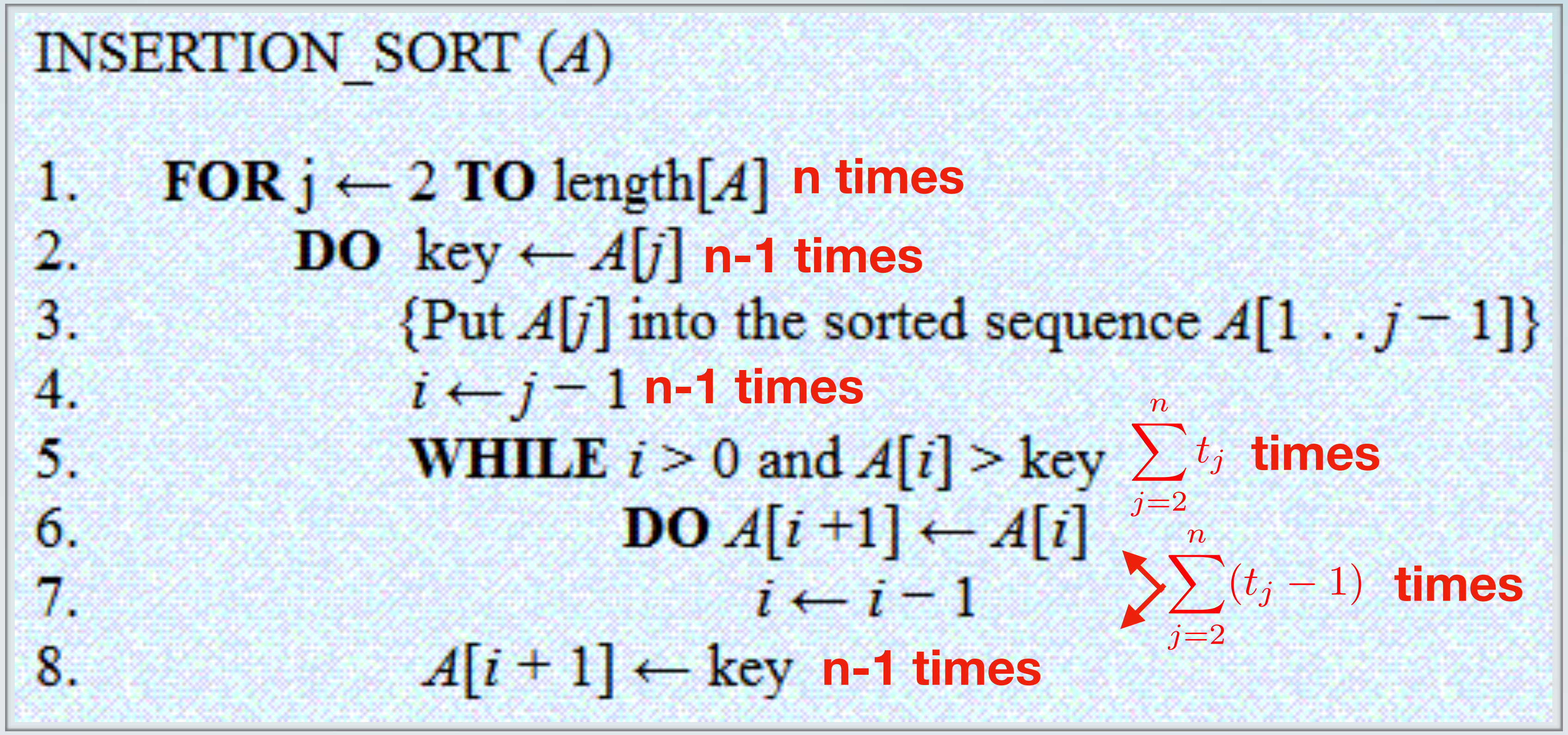


```
INSERTION_SORT (A)

FOR j ← 2 TO length[A]   n times
    DO key ← A[j]   n-1 times
        {Put A[j] into the sorted sequence A[1 . . j − 1]}
        i ← j − 1   n-1 times
        WHILE i > 0 and A[i] > key   $\sum_{j=2}^{n} t_j$   times
            DO A[i +1] ← A[i]
                i ← i − 1   $\sum_{j=2}^{n} (t_j - 1)$   times
        A[i + 1] ← key   n-1 times
```

for loops, the tests are executed
one more time than the loop body

- Select an input uniformly at random from all possible sequences with n numbers.

# Example: Average Running Time of InsertionSort

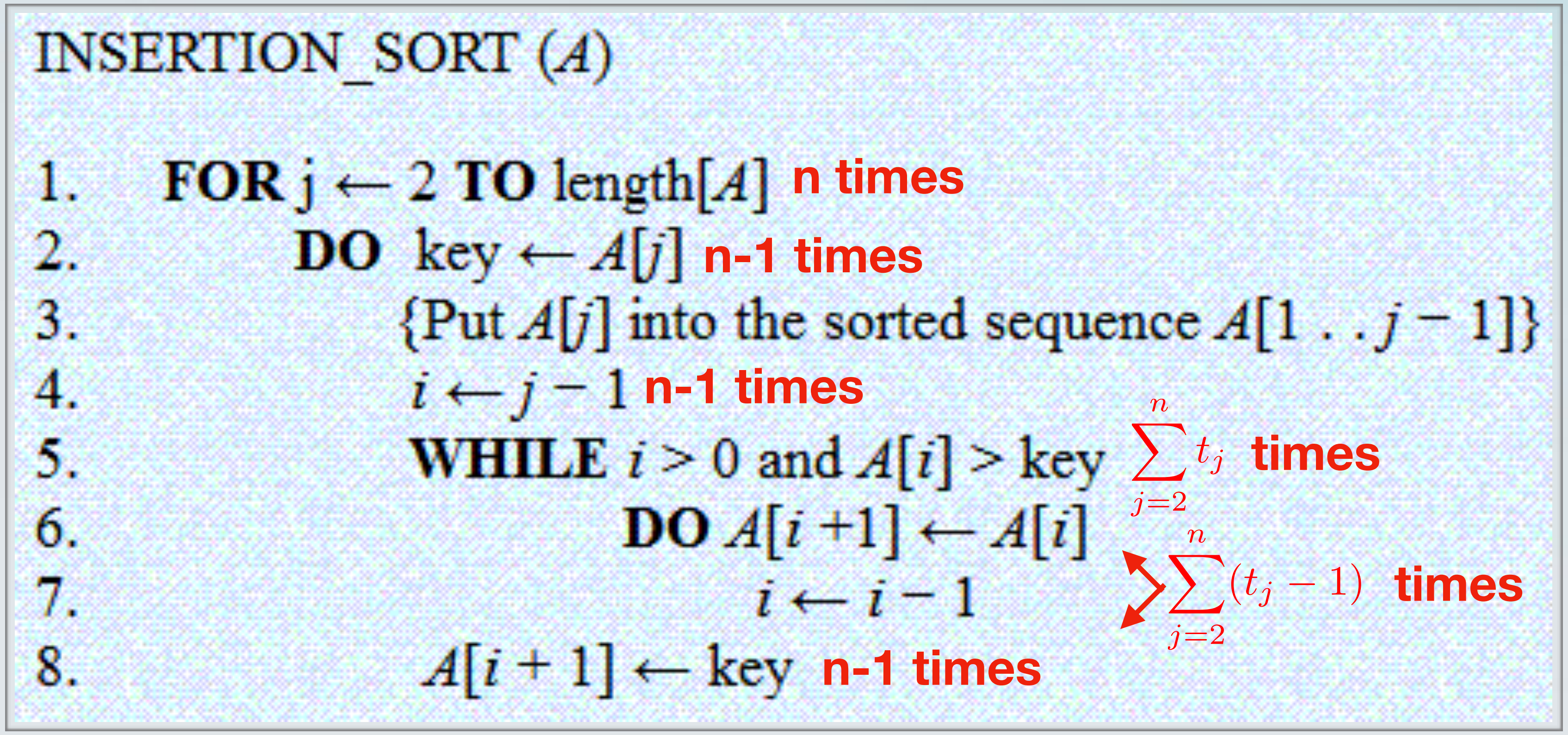


```
INSERTION_SORT (A)

FOR j ← 2 TO length[A]   n times
    DO key ← A[j]   n-1 times
        {Put A[j] into the sorted sequence A[1 . . j − 1]}
        i ← j − 1   n-1 times
        WHILE i > 0 and A[i] > key   $\sum_{j=2}^{n} t_j$ times
            DO A[i +1] ← A[i]   $\sum_{j=2}^{n} (t_j - 1)$ times
                i ← i − 1
        A[i + 1] ← key   n-1 times
```

for loops, the tests are executed
one more time than the loop body

- Select an input uniformly at random from all possible sequences with n numbers.
- On average, key will be smaller than half of the elements in A[1,…,j].

# Example: Average Running Time of InsertionSort

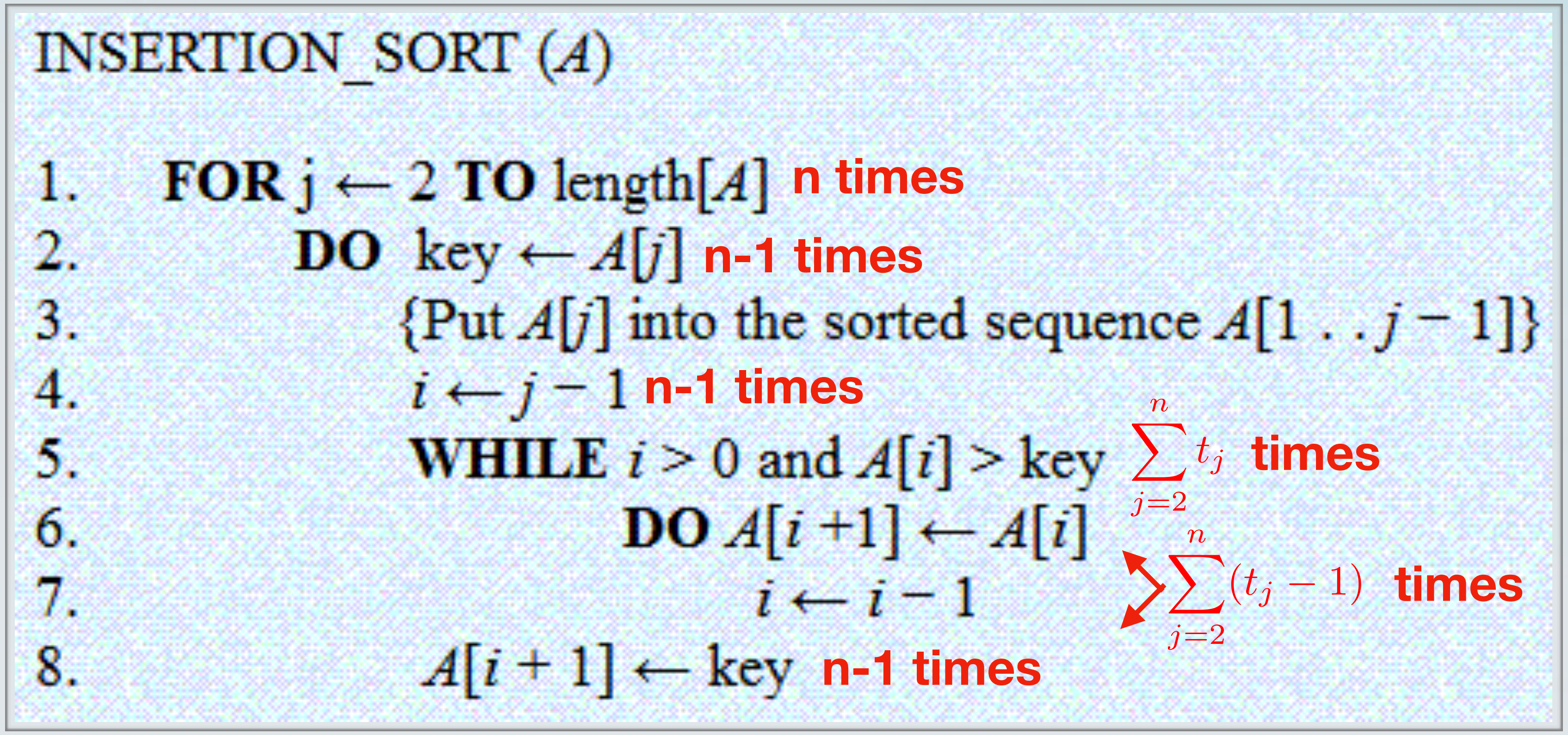


```
INSERTION_SORT (A)

FOR j ← 2 TO length[A]   n times
    DO key ← A[j]   n-1 times
        {Put A[j] into the sorted sequence A[1 . . j − 1]}
        i ← j − 1   n-1 times
        WHILE i > 0 and A[i] > key   $\sum_{j=2}^{n} t_j$ times
            DO A[i +1] ← A[i]
            i ← i − 1   $\sum_{j=2}^{n} (t_j - 1)$ times
        A[i + 1] ← key   n-1 times
```

for loops, the tests are executed
one more time than the loop body

- Select an input uniformly at random from all possible sequences with n numbers.
- On average, key will be smaller than half of the elements in A[1,…,j].
- The while loop will look “halfway” through the sorted subarray A[1,…,j].

# Example: Average Running Time of InsertionSort

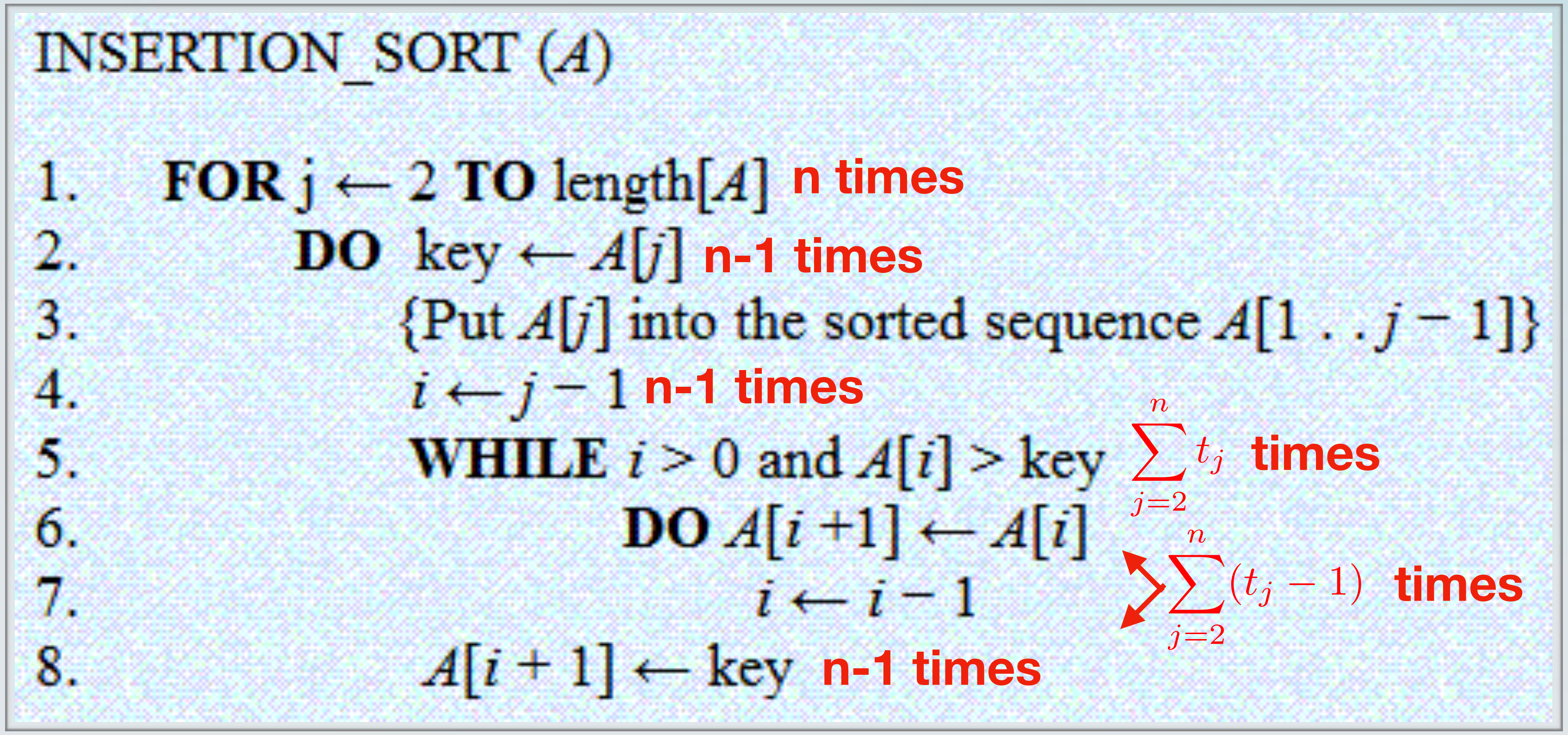


```
INSERTION_SORT ($A$)

FOR j $\leftarrow$ 2 TO length[$A$]   n times
    DO key $\leftarrow A[j]$   n-1 times
        {Put $A[j]$ into the sorted sequence $A[1..j-1]$}
        $i \leftarrow j - 1$   n-1 times
        WHILE $i > 0$ and $A[i] >$ key   $\sum_{j=2}^{n} t_j$ times
            DO $A[i+1] \leftarrow A[i]$
                $i \leftarrow i - 1$   $\sum_{j=2}^{n} (t_j - 1)$ times
        $A[i+1] \leftarrow$ key   n-1 times
```

for loops, the tests are executed one more time than the loop body

- Select an input uniformly at random from all possible sequences with n numbers.
- On average, key will be smaller than half of the elements in A[1,…,j].
- The while loop will look “halfway” through the sorted subarray A[1,…,j].
- This means that $t_j = \dfrac{j}{2}$

# Example: Average Running Time of InsertionSort



```
INSERTION_SORT (A)

FOR j ← 2 TO length[A]   n times
    DO key ← A[j]   n-1 times
       {Put A[j] into the sorted sequence A[1 . . j − 1]}
       i ← j − 1   n-1 times
       WHILE i > 0 and A[i] > key   $\sum_{j=2}^{n} t_j$ times
             DO A[i +1] ← A[i]   $\sum_{j=2}^{n} (t_j - 1)$ times
                i ← i − 1
       A[i + 1] ← key   n-1 times
```

for loops, the tests are executed
one more time than the loop body

- Select an input uniformly at random from all possible sequences with n numbers.
- On average, key will be smaller than half of the elements in A[1,…,j].
- The while loop will look “halfway” through the sorted subarray A[1,…,j].
- This means that $t_j = \frac{j}{2}$

Bounded by some $cn^2$ for some constant c

# Asymptotic Notation

- When n becomes large, it makes less of a difference if an algorithm takes 2n or 3n steps to finish.
- In particular, 3logn steps are fewer than 2n steps.
- We would like to avoid having to calculate the precise constants.
- We use asymptotic notation.

# Asymptotic Notation

$$\mathbf{O(g(n)) = f(n)} : \text{there exist positive constants } c \text{ and } n_0 \text{ such that}$$
$$0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0.$$

$$\mathbf{\Omega(g(n)) = f(n)} : \text{there exist positive constants } c \text{ and } n_0 \text{ such that}$$
$$0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0.$$

$$\mathbf{\Theta(g(n)) = f(n)} : \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that}$$
$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0.$$

$$\mathbf{o(g(n)) = f(n)} : \text{for any constant } c > 0, \text{ there exists a constant}$$
$$n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0.$$

$$\mathbf{o(g(n)) = f(n)} : \text{for any constant } c > 0, \text{ there exists a constant}$$
$$n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0.$$

# Asymptotic Notation

$$\mathbf{O(g(n)) = f(n)} : \text{there exist positive constants } c \text{ and } n_0 \text{ such that}$$
$$0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0.$$

- For sufficiently large inputs, there is a constant such that c g(n) is not smaller than f(n).
- For example, for sufficiently large inputs, 2n is larger than 3logn. Therefore, 3log n = O(n).
- Intuitively, g(n) grows “not slower” than f(n).
- **Use**: If we can upper bound the running time of an algorithm by _c*g(n)_, where c is some constant and _g(•)_ is a function of the input, then we can say that the running time is _O(g(n))._

# Asymptotic Notation

$$\mathbf{\Omega(g(n)) = f(n)} : \text{there exist positive constants } c \text{ and } n_0 \text{ such that}$$
$$0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 .$$

- For sufficiently large inputs, there is a constant such that c g(n) is not larger than f(n).

- For example, for sufficiently large inputs, 3logn is smaller than 2n. Therefore, 2n = Ω(logn).

- Intuitively, g(n) grows “not faster” than f(n).

- **Use**: If we can lower bound the running time of an algorithm by *c*g(n)*, where c is some constant and *g(•)* is a function of the input, then we can say that the running time is *Ω(g(n)).*

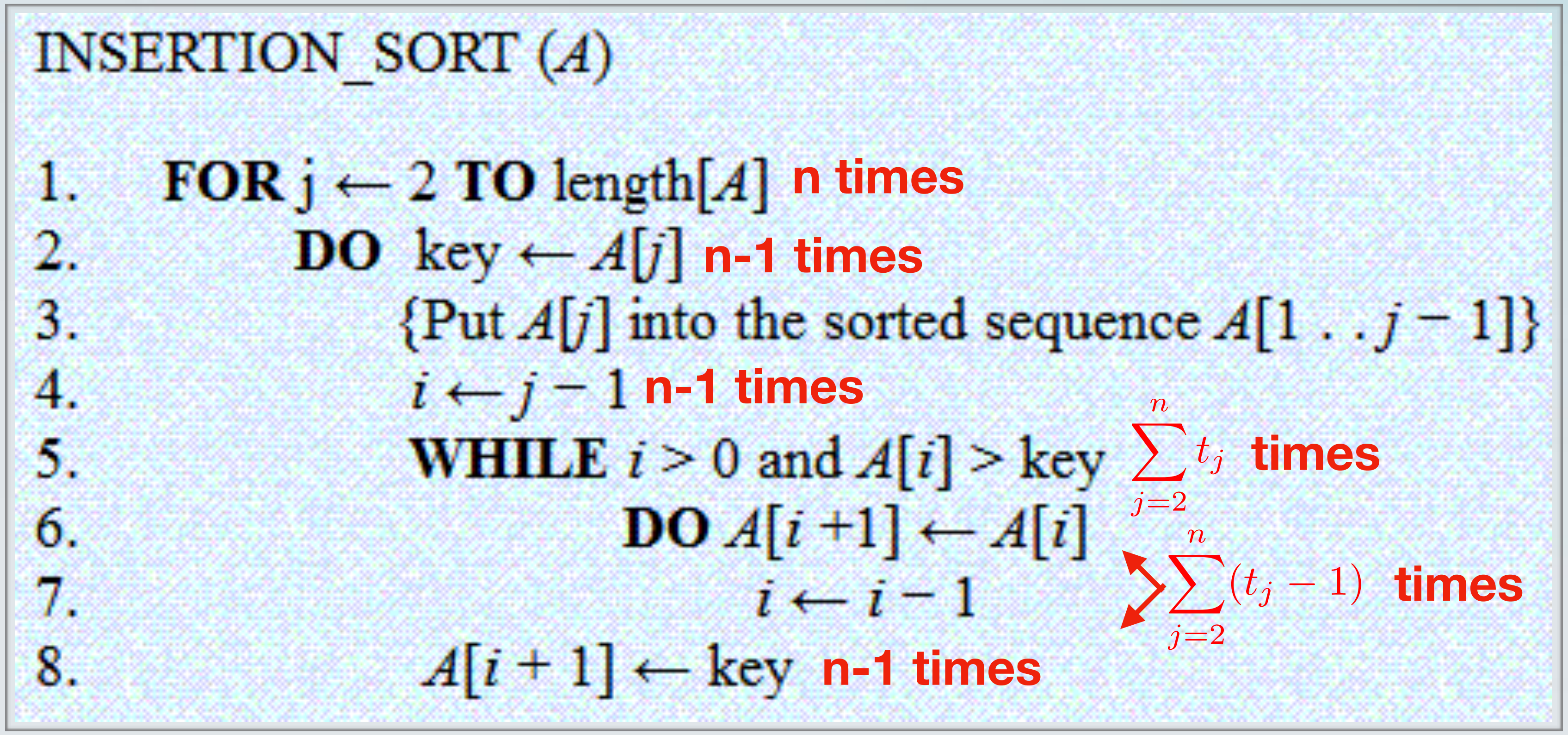
# Asymptotic Notation

$$\mathbf{\Theta(g(n)) = f(n)} : \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that}$$
$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0.$$

- If a function is both *O(g(n))* and *Ω(g(n)).*

- **Use:** If we can show that the running time of an algorithm is lower bounded by *c1*g(n)* and upper bounded by *c2*g(n)* for some constants c1 and c2 and some function *g(•)* of n, then we can say that the running time is *Θ(g(n)).*

# Example: Running Time of InsertionSort



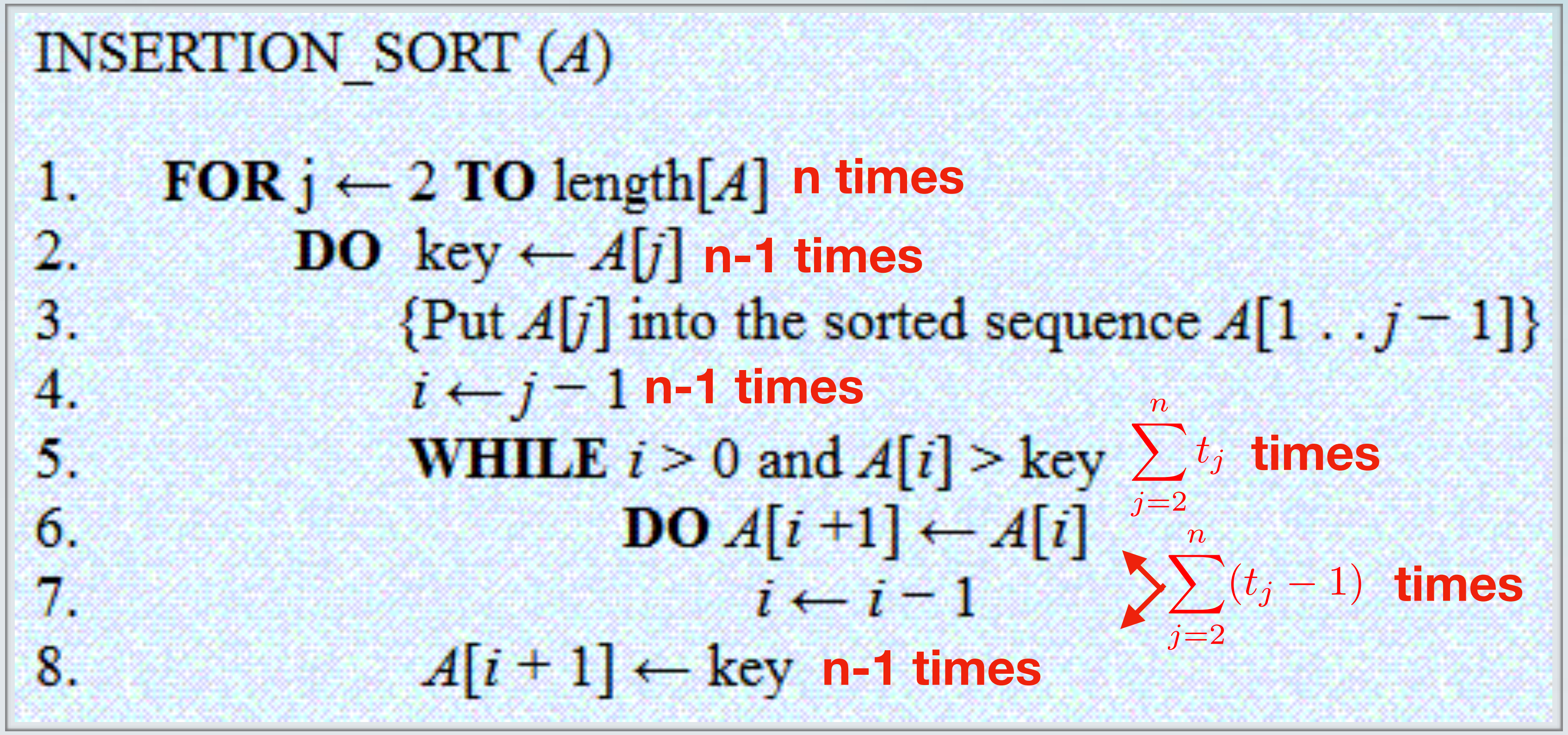
```
INSERTION_SORT (A)

FOR j ← 2 TO length[A]   n times
    DO key ← A[j]   n-1 times
       {Put A[j] into the sorted sequence A[1 . . j − 1]}
       i ← j − 1   n-1 times
       WHILE i > 0 and A[i] > key   $\sum_{j=2}^{n} t_j$ times
             DO A[i +1] ← A[i]      $\sum_{j=2}^{n} (t_j - 1)$ times
                i ← i − 1
       A[i + 1] ← key   n-1 times
```

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

Best case? **Sorted array,** $t_j = 1$ Bounded by some $cn$ for some constant c

Worst case? **Reverse sorted array,** $t_j = j$ Bounded by some $cn^2$ for some constant c

# Example: Running Time of InsertionSort



```
INSERTION_SORT ($A$)

FOR j ← 2 TO length[$A$]   n times
    DO  key ← $A[j]$   n-1 times
        {Put $A[j]$ into the sorted sequence $A[1 \,.\,.\, j-1]$}
        $i \leftarrow j-1$   n-1 times
        WHILE $i > 0$ and $A[i] >$ key   $\sum_{j=2}^{n} t_j$ times
              DO $A[i+1] \leftarrow A[i]$   $\sum_{j=2}^{n} (t_j - 1)$ times
                 $i \leftarrow i-1$
        $A[i+1] \leftarrow$ key   n-1 times
```

**What is the asymptotic complexity of InsertionSort?**

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

Best case? **Sorted array,** $t_j = 1$ Bounded by some $cn$ for some constant c

Worst case? **Reverse sorted array,** $t_j = j$ Bounded by some $cn^2$ for some constant c

# Asymptotic Notation

$$\mathbf{o(g(n)) = f(n)} : \text{for any constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0.$$

- The bound holds for sufficiently large inputs and for any constant c.
- Equivalent interpretation: $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$
- As n approaches infinity, *f(n)* becomes insignificant compared to *g(n)*.
- For example: $2n = o(n^2)$

# Asymptotic Notation

$$\mathbf{o(g(n)) = f(n)} : \text{for any constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \le cg(n) < f(n) \text{ for all } n \ge n_0.$$

- The bound holds for sufficiently large inputs and for any constant c.
- Equivalent interpretation: $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty.$
- As n approaches infinity, *g(n)* becomes insignificant compared to *f(n)*.
- For example: $4n^2 = \omega(n)$

# Examples

$$5n^3 + 100 = O(n^3)$$

$$5n^3 + 100 = \Omega(n^3)$$

$$\log n = o(n^5)$$

$$n^5 = o(2^n)$$

$$\log(4n) = \log n + \log 4 = O(\log n)$$

$$\log(n^4) = 4 \log n = O(\log n)$$

$$(4n)^3 = 64n^4 = O(n^3)$$

$$(n^4)^3 = n^12 = \omega(n^3)$$

$$3^{(4n)} = 81^n = \omega(3^n)$$

# Running time hierarchy

| $O(\log n)$ | $O(n)$ | $O(n \log n)$ | $O(n^2)$ | $O(n^\alpha)$ | $O(c^n)$ |
|---|---|---|---|---|---|
| logarithmic | linear |  | quadratic | polynomial | exponential |
| The algorithm does not even read the whole input. | The algorithm accesses the input only a constant number of times. | The algorithm splits the inputs into two pieces of similar size, solves each part and merges the solutions. | The algorithm considers pairs of elements. | The algorithm performs many nested loops. | The algorithm considers many subsets of the input elements. |

|  |  |  |  |
|---|---|---|---|
| constant | $O(1)$ | superlinear | $\omega(n)$ |
| superconstant | $\omega(1)$ | superpolynomial | $\omega(n^\alpha)$ |
| sublinear | $o(n)$ | subexponential | $o(c^n)$ |