Algorithms and Data Structures

Content and Basic Notions

Algorithms and Data Structures

- A continuation of Introduction to Algorithms and Data Structures (INF2 - IADS).
- Mostly same techniques, more advanced applications.
  - Divide-and-Conquer, Greedy, Dynamic Programming
- More emphasis on “Algorithms” rather than “Data Structures”.
- More theorem proving.

The course content

The course content

- Review of algorithm design basics, including *time and space complexity*, and *asymptotic notation*.

The course content

- Review of algorithm design basics, including *time and space complexity*, and *asymptotic notation*.
- The *divide-and-conquer* paradigm:

The course content

- Review of algorithm design basics, including *time and space complexity*, and *asymptotic notation*.
- The *divide-and-conquer* paradigm:
  - Sorting, matrix multiplication, Fourier transform.

The course content

- Review of algorithm design basics, including *time and space complexity*, and *asymptotic notation*.
- The *divide-and-conquer* paradigm:
  - Sorting, matrix multiplication, Fourier transform.
  - Upper and lower bound proofs.

The course content

- Review of algorithm design basics, including *time and space complexity*, and *asymptotic notation*.
- The *divide-and-conquer* paradigm:
  - Sorting, matrix multiplication, Fourier transform.
  - Upper and lower bound proofs.
  - Solving recurrence relations.

The course content (cont)

The course content (cont)

- The *greedy* paradigm:

The course content (cont)

- The *greedy* paradigm:
    - Minimum spanning trees.

The course content (cont)

- The *greedy* paradigm:
  - Minimum spanning trees.
  - Network flows.

The course content (cont)

- The *greedy* paradigm:
  - Minimum spanning trees.
  - Network flows.
  - Linear programming.

The course content (cont)

- The *greedy* paradigm:
  - Minimum spanning trees.
  - Network flows.
  - Linear programming.
- The *dynamic programming* paradigm:

The course content (cont)

- The *greedy* paradigm:
  - Minimum spanning trees.
  - Network flows.
  - Linear programming.
- The *dynamic programming* paradigm:
  - Matrix-chain multiplication and other examples.

What is an 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.

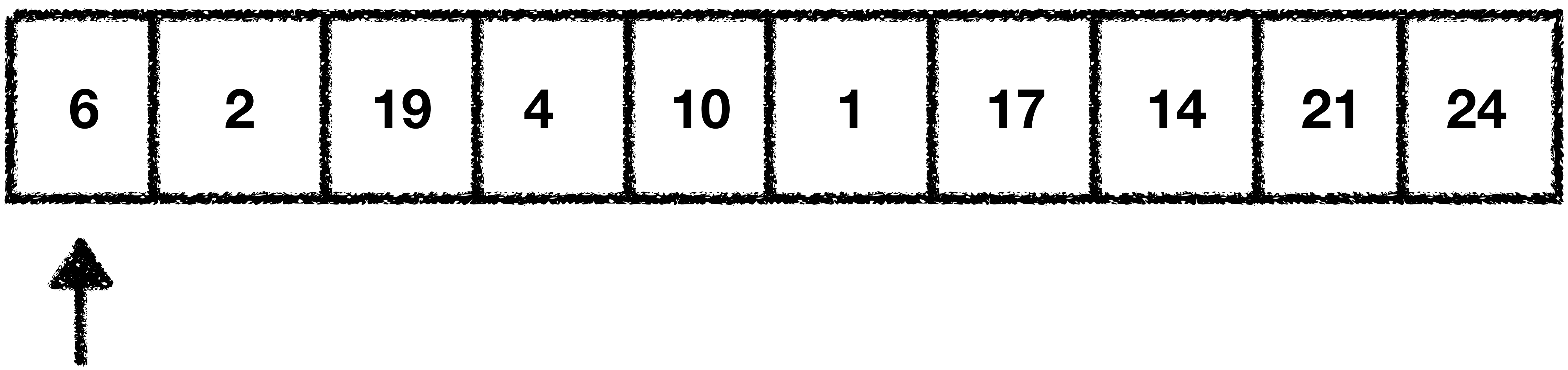
Example: Sorting

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

6	2	19	4	10	1	17	14	21	24

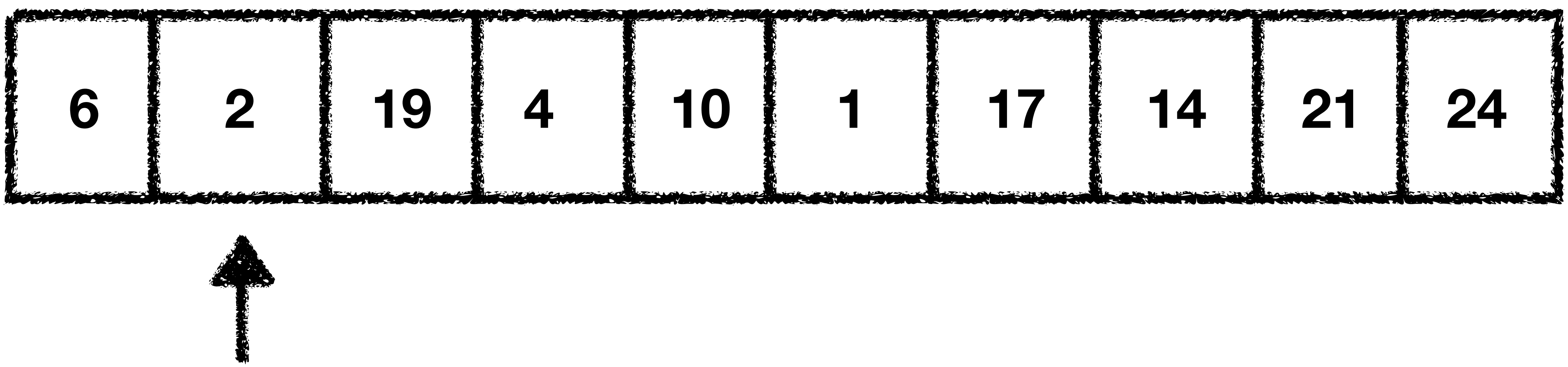
Example: Sorting

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



Example: Sorting

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



Example: Sorting

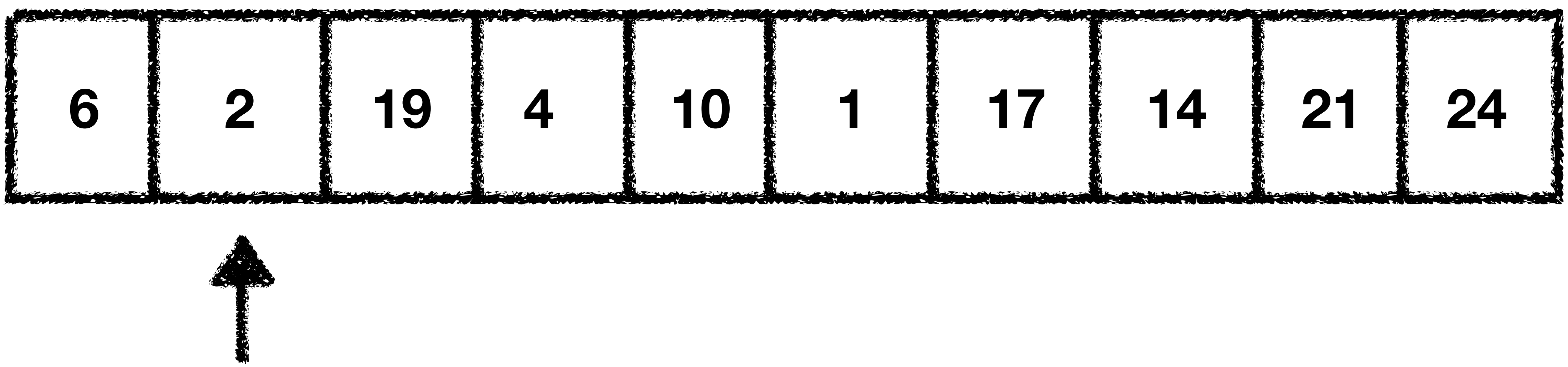
- Given a sequence of numbers, put them in increasing order.

6	2	19	4	10	1	17	14	21	24

Is 2 < 6?

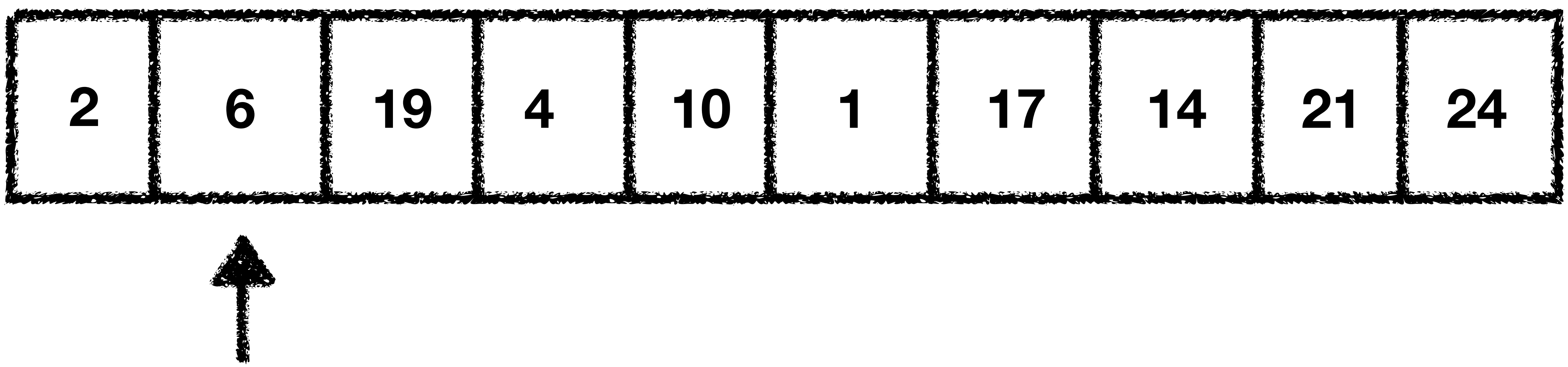
Example: Sorting

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



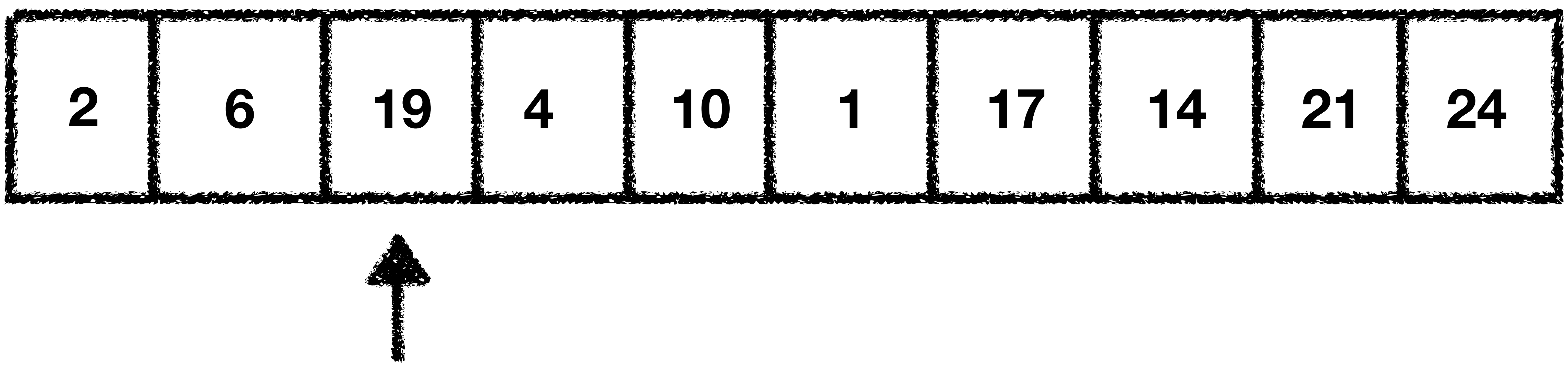
Example: Sorting

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



Example: Sorting

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



Example: Sorting

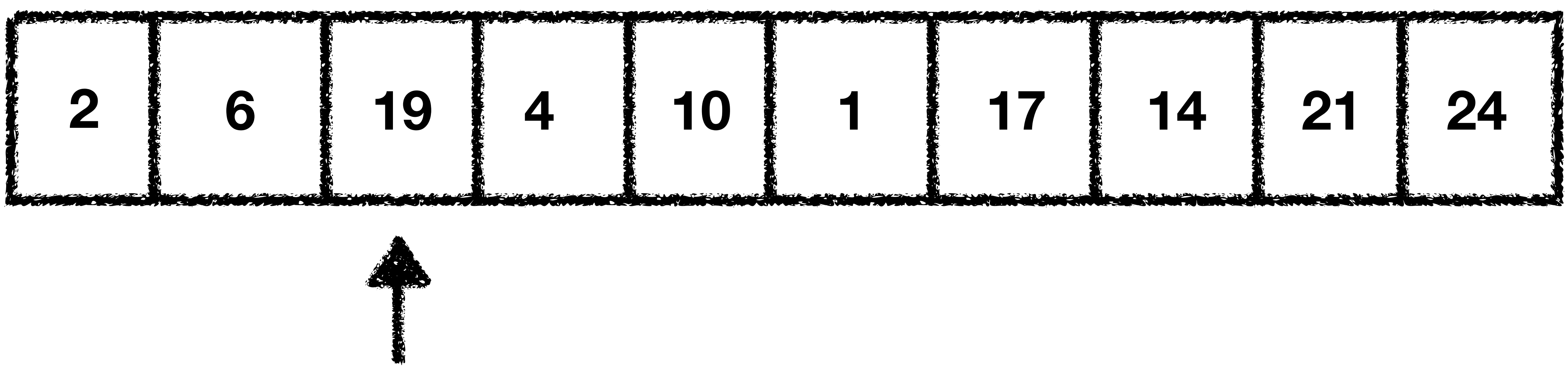
- Given a sequence of numbers, put them in increasing order.

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

Is 19 < 6?

Example: Sorting

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



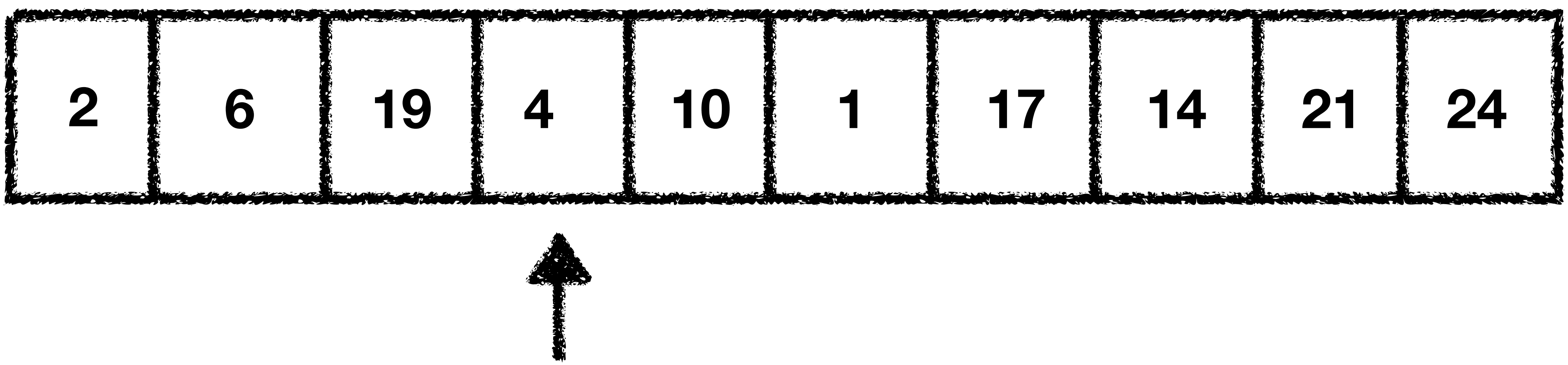
Example: Sorting

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

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

Example: Sorting

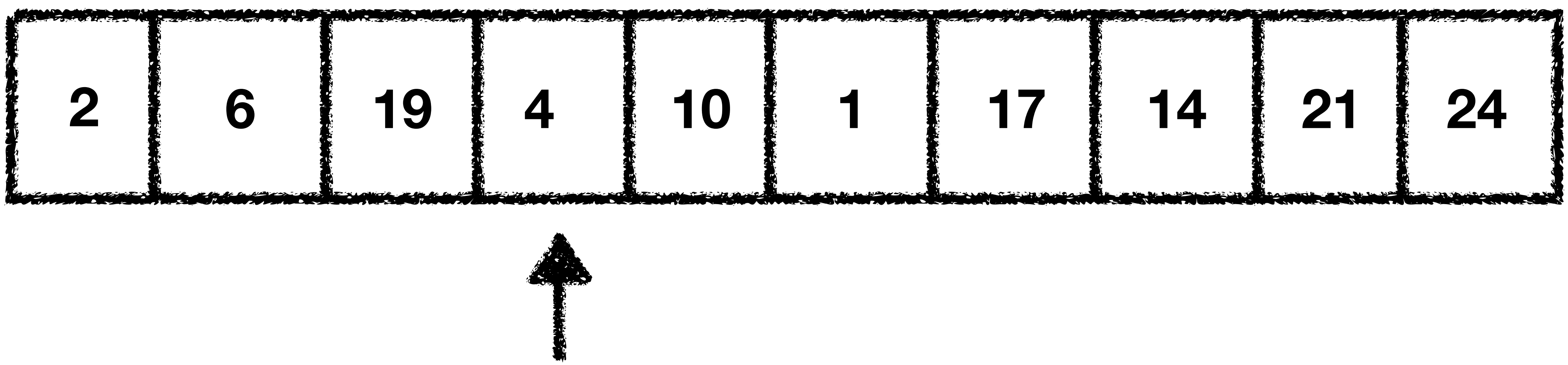
- Given a sequence of numbers, put them in increasing order.



Is 4 < 19?

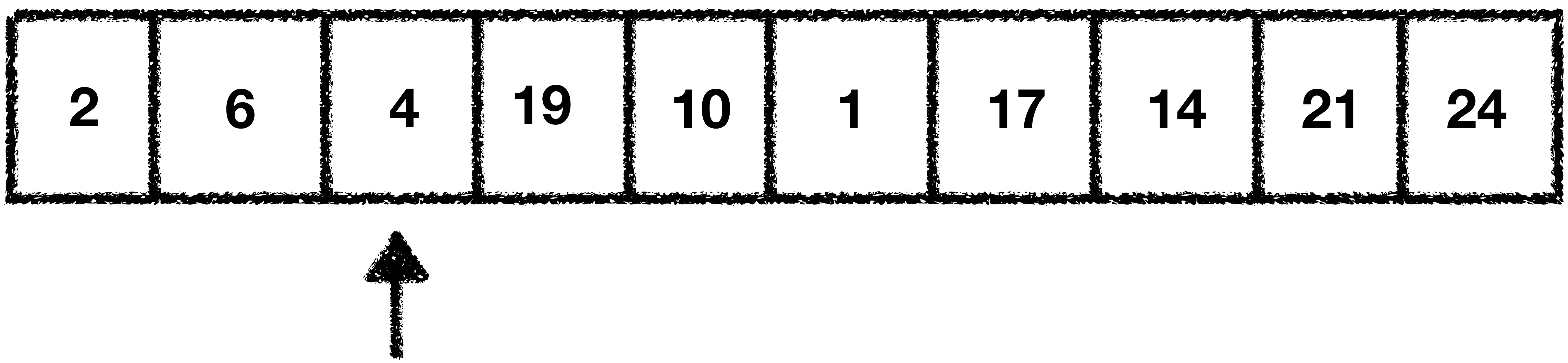
Example: Sorting

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



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	**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

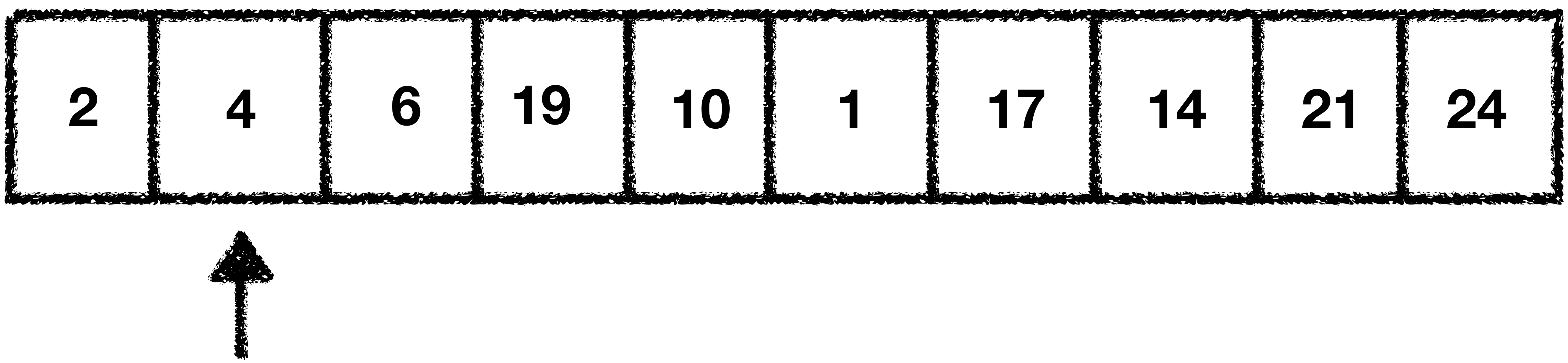
- Given a sequence of numbers, put them in increasing order.

2	4	6	19	10	1	17	14	21	24

Is 4 < 2?

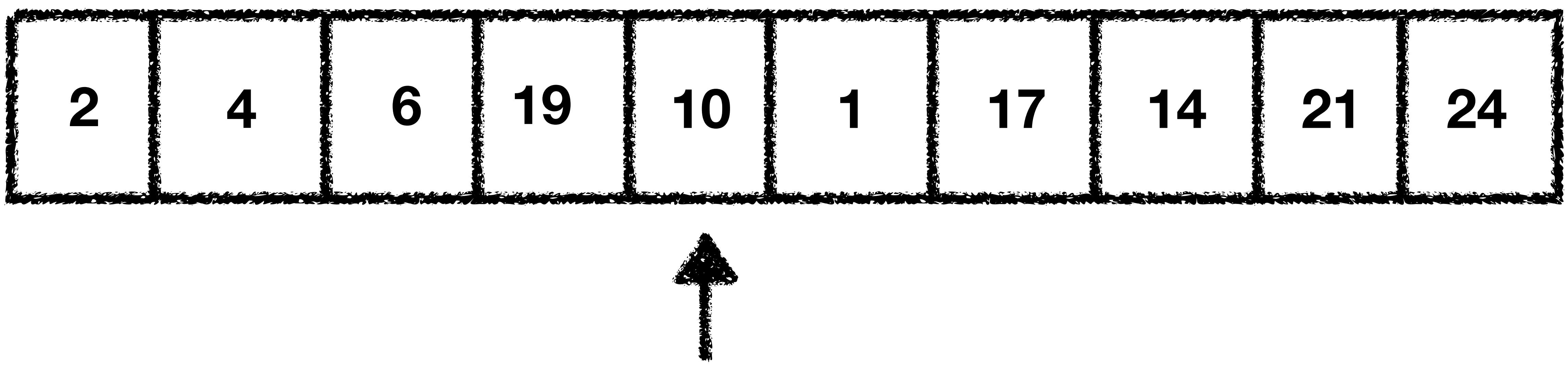
Example: Sorting

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



Example: Sorting

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



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

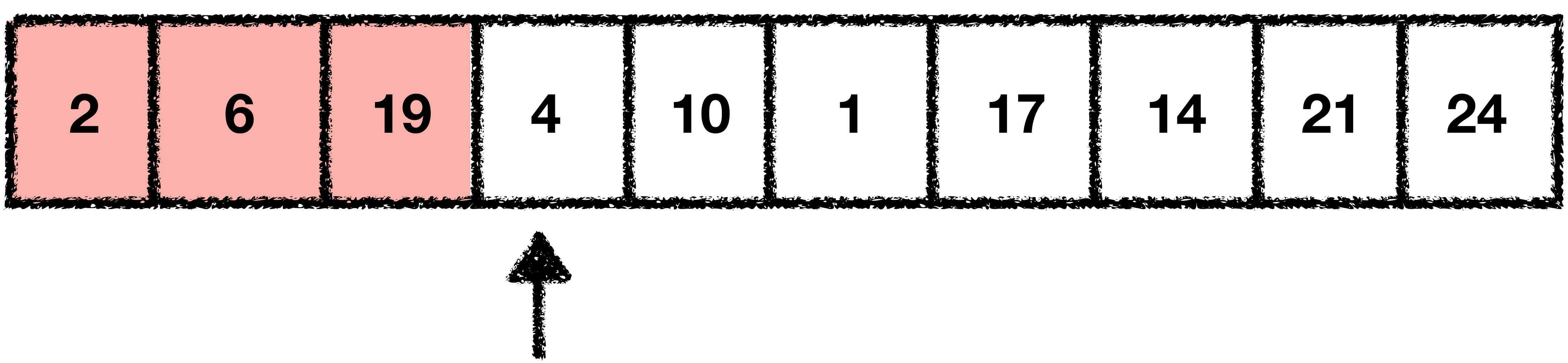
continues the same way…

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 ← 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 ← 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).

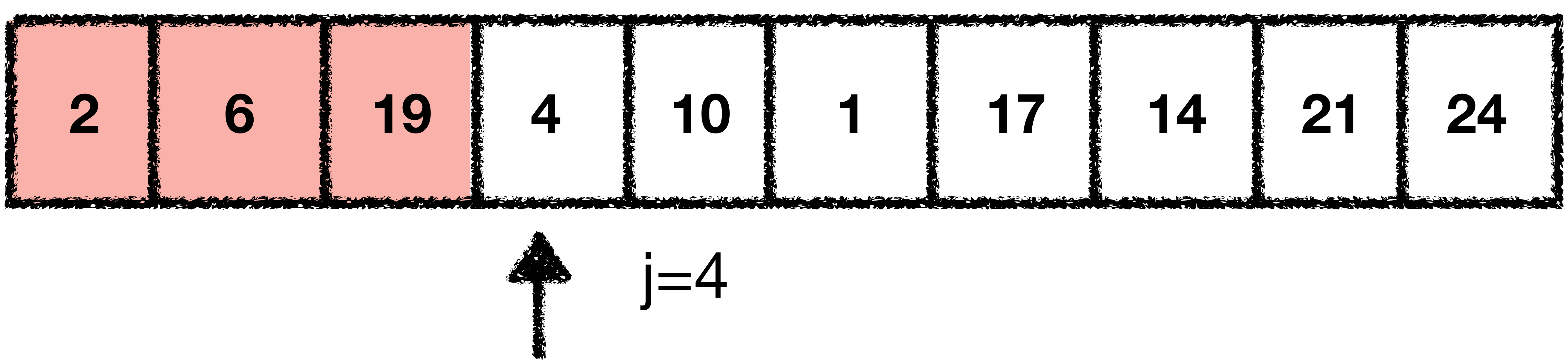
2	6	19	4	10	1	17	14	21	24

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 ← 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).

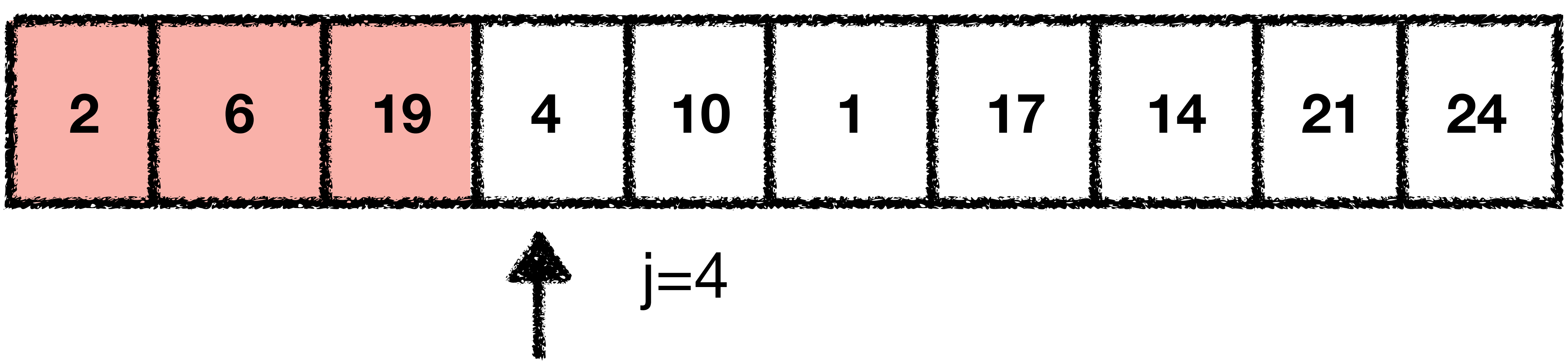


Describing algorithms: Pseudocode

```
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 ← 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).

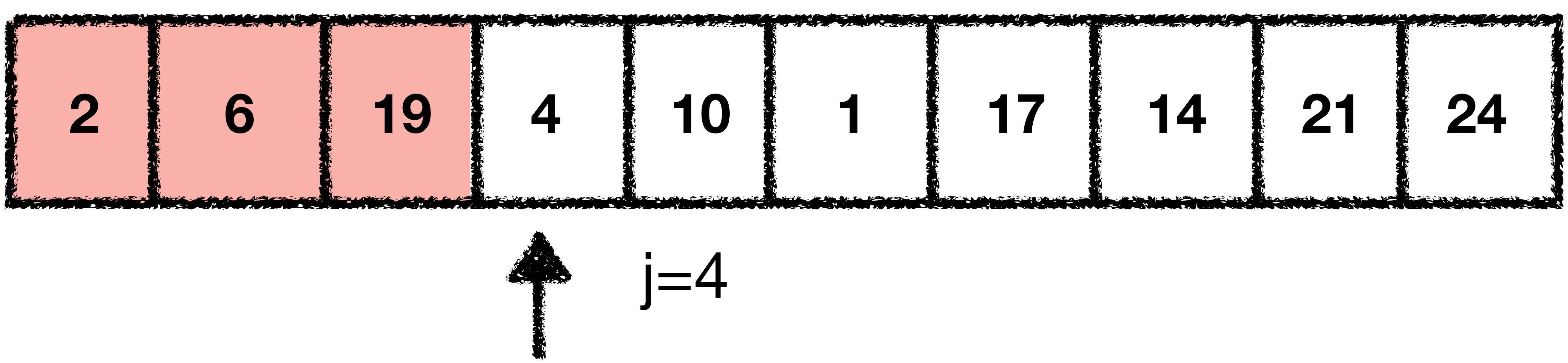


Describing algorithms: Pseudocode

```
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 ← 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 ← 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

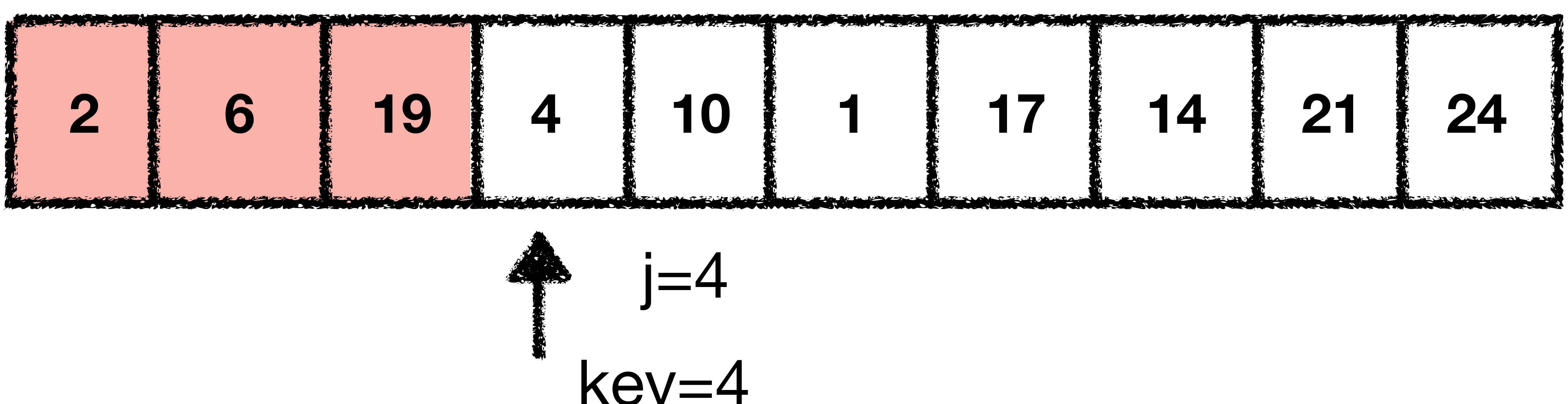
key=4

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 ← 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).

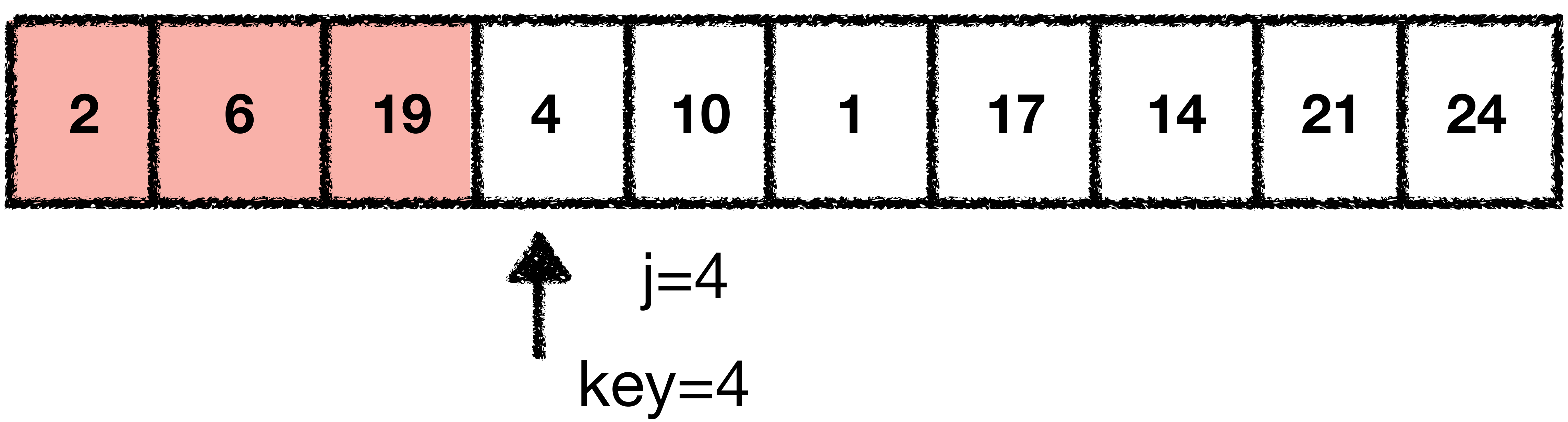


Describing algorithms: Pseudocode

```
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 ← 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).

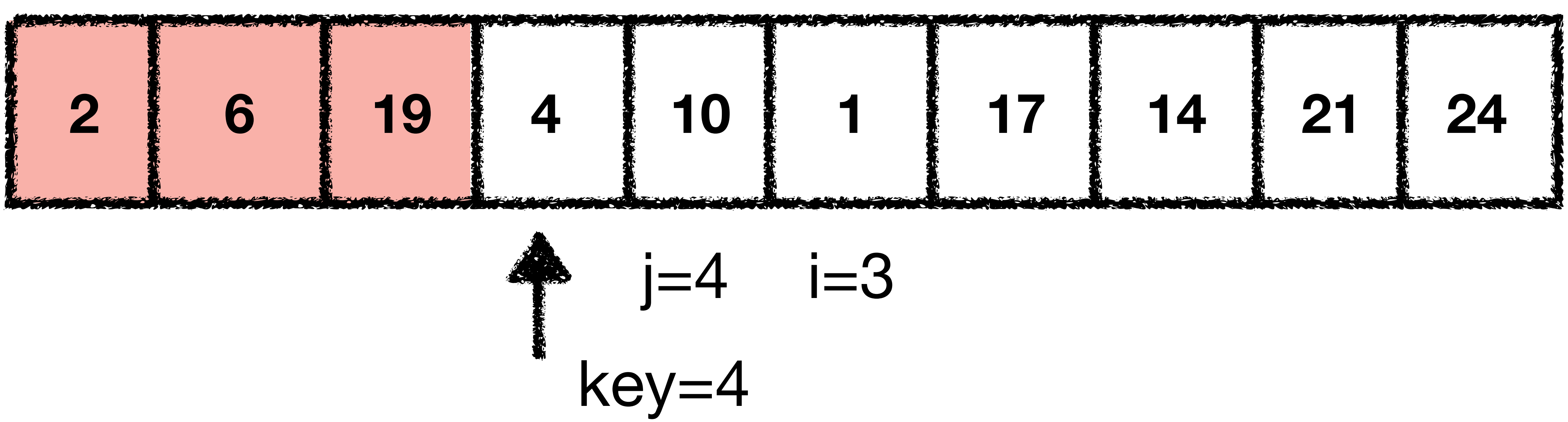


Describing algorithms: Pseudocode

```
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 ← 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).

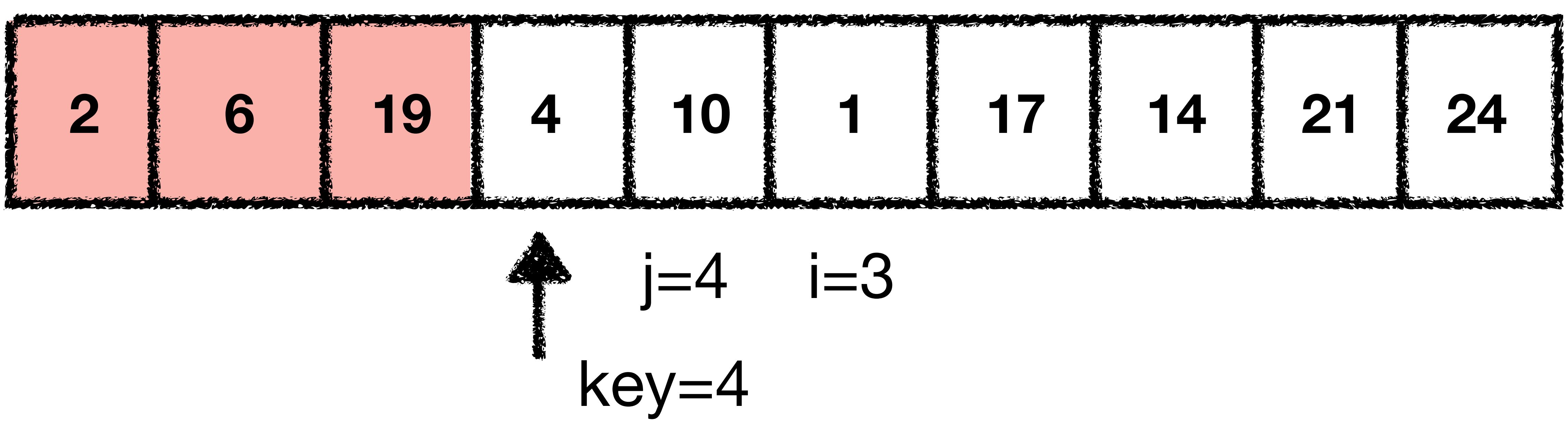


Describing algorithms: Pseudocode

```
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 ← 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).

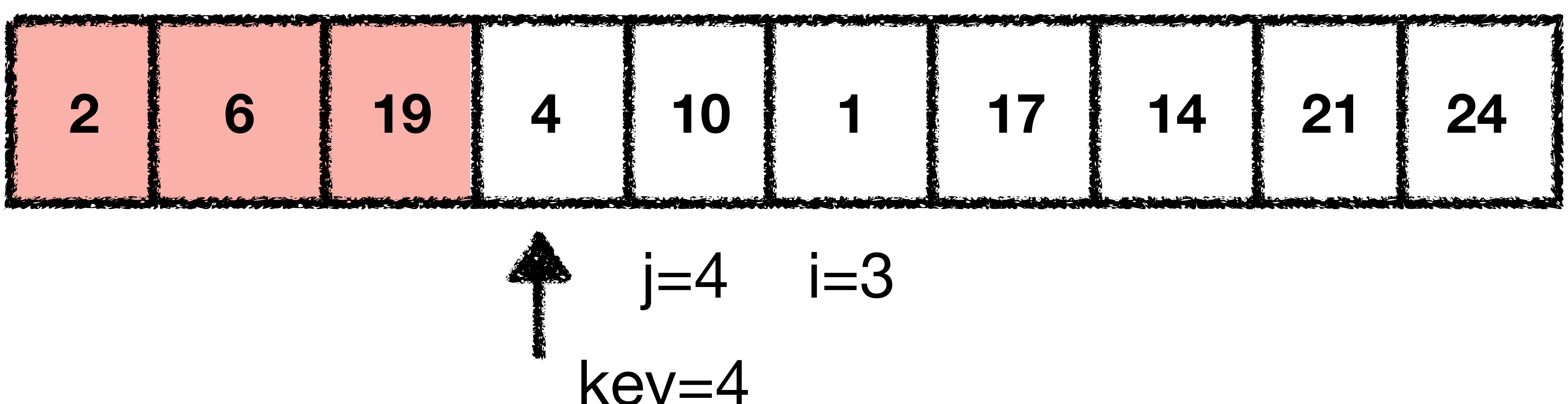


Describing algorithms: Pseudocode

```
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 ← 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).

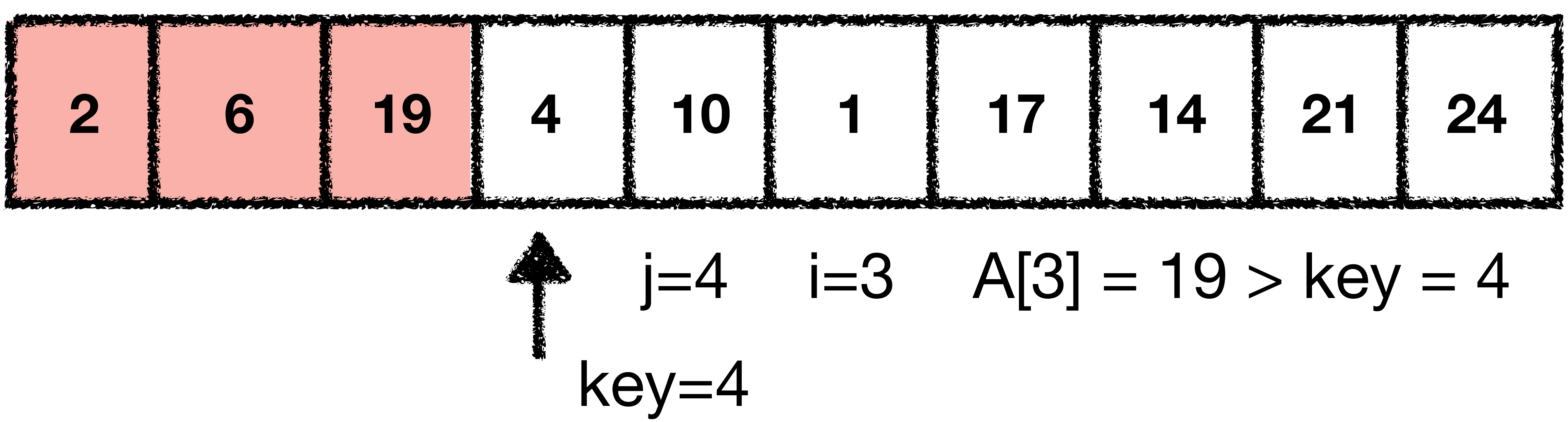


Describing algorithms: Pseudocode

```
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 ← 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).

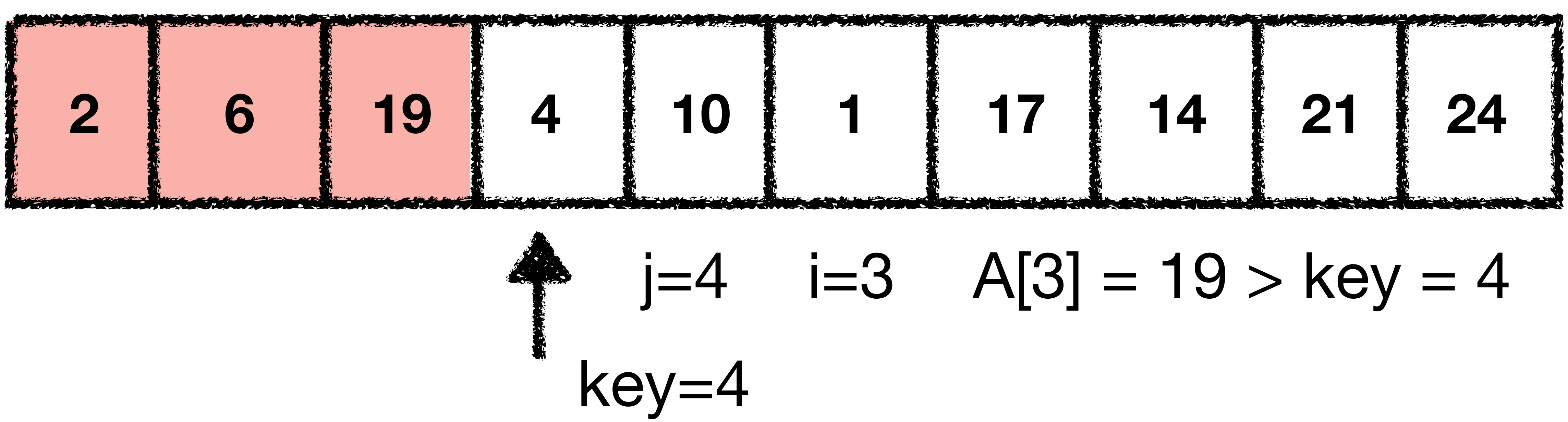


Describing algorithms: Pseudocode

```
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 ← 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 ← 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

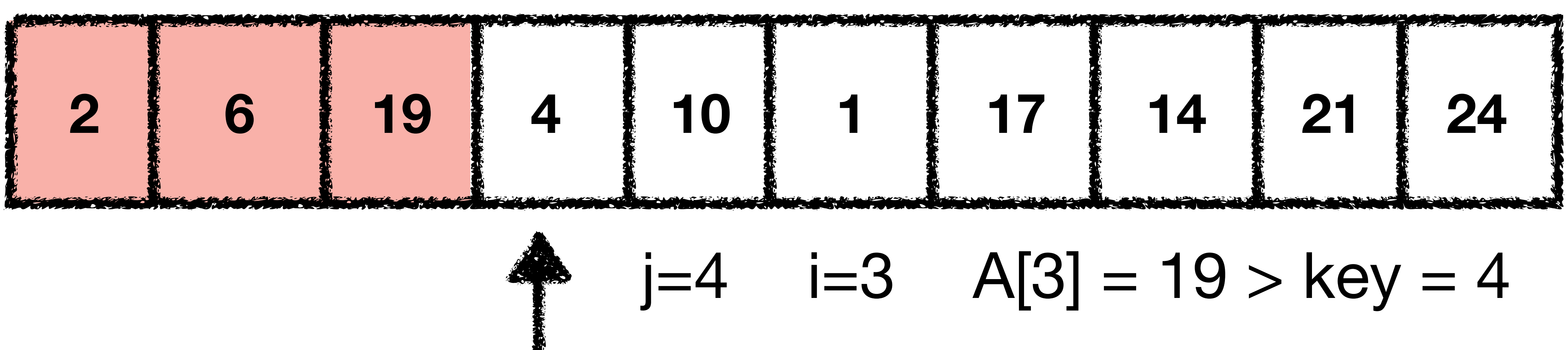
A[4] = 19, i = 2

Describing algorithms: Pseudocode

```
INSERTION_SORT (A)

FOR j ← 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).



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

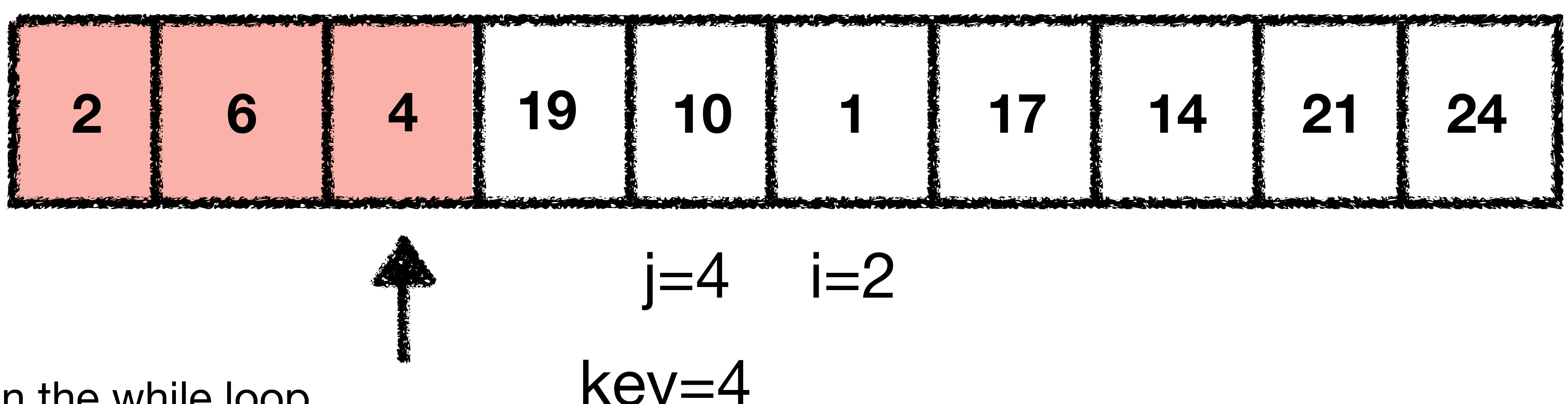
key=4 A[4] = 19, i = 2

Describing algorithms: Pseudocode

```
INSERTION_SORT (A)

FOR j ← 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).



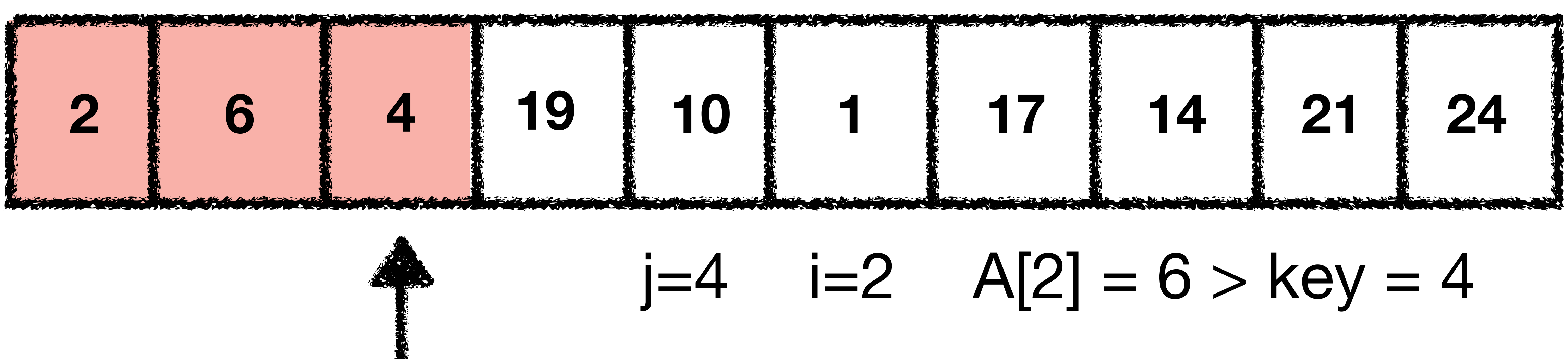
still in the while loop

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).



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

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

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

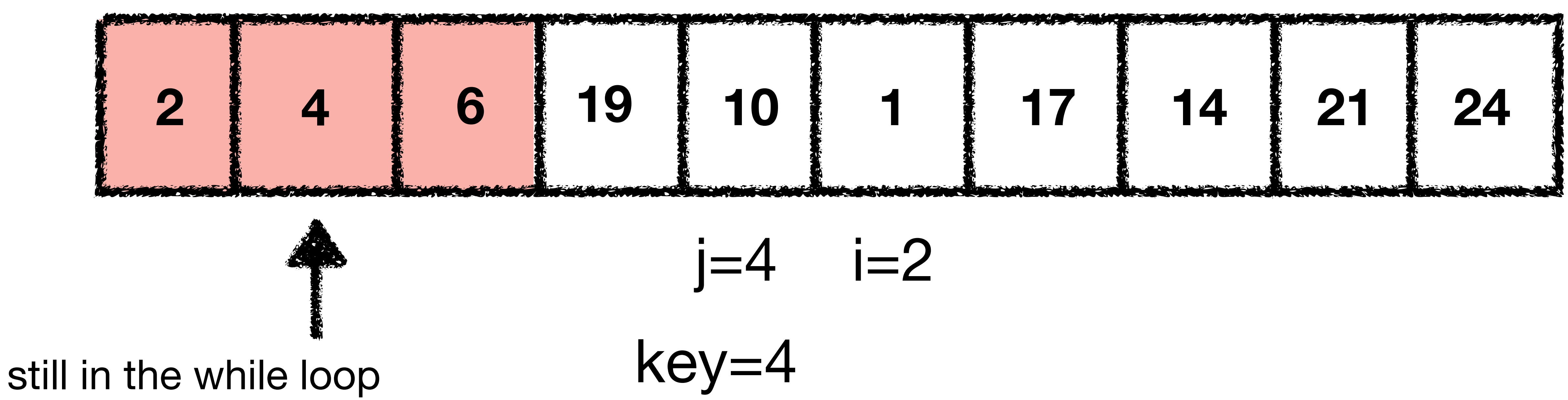
still in the while loop

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).



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

key=4

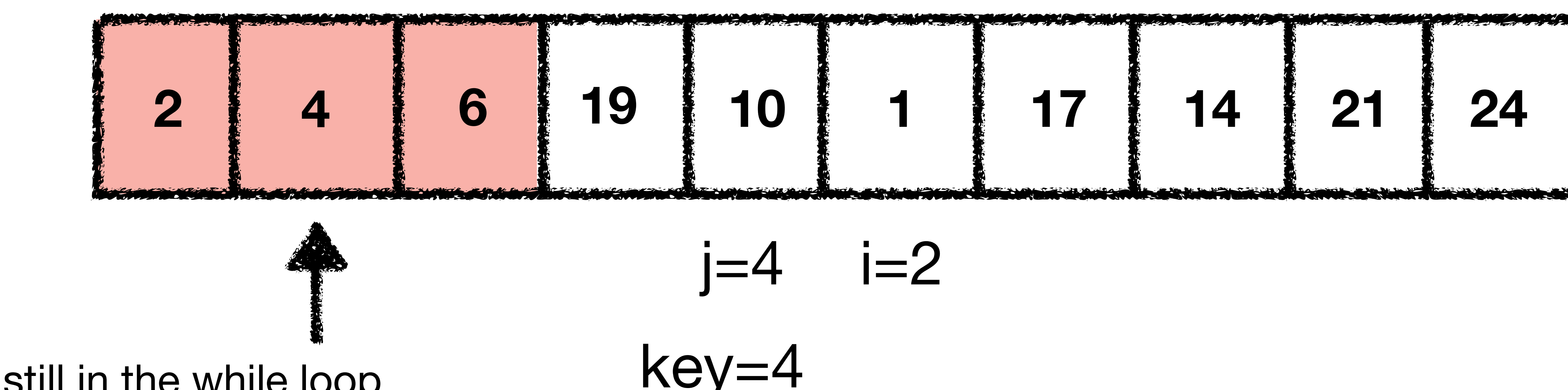
still in the while loop

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).

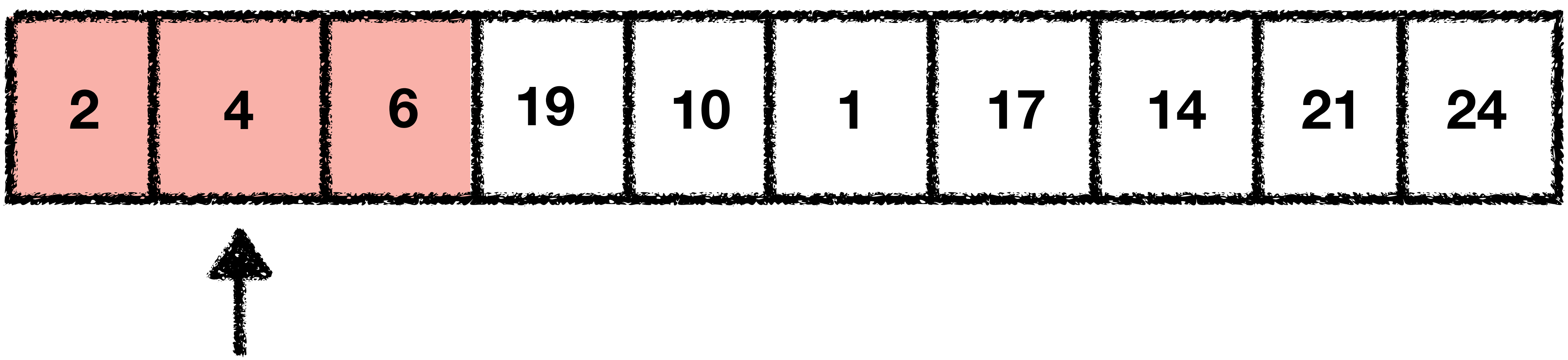


Describing algorithms: Pseudocode

```
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).



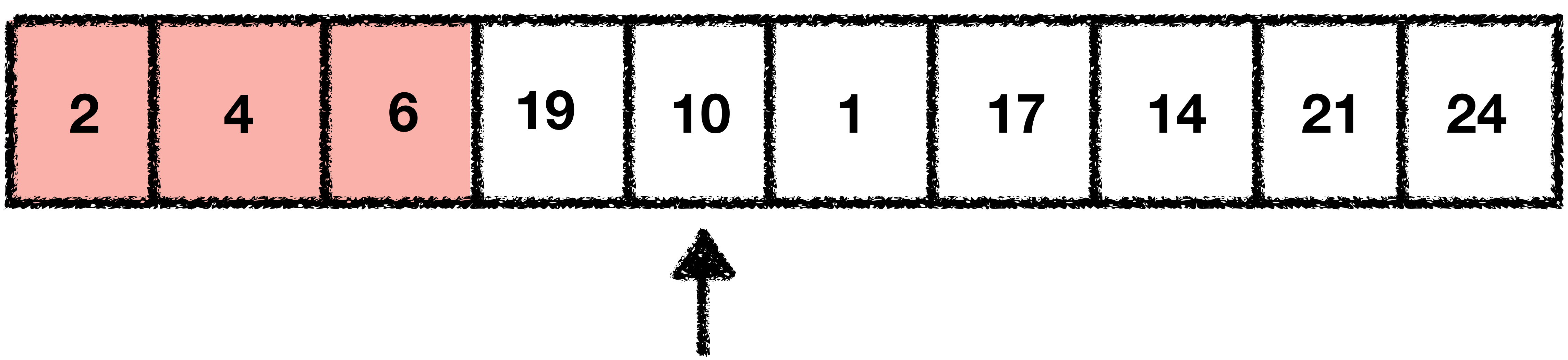
still in the while loop

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).



j=5

still in the while loop

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

Correctness

- How do we prove that our algorithm is always correct?

Correctness

- How do we prove that our algorithm is always correct?
  - Proof techniques (induction, proof-by-contradiction etc).

Correctness

- How do we prove that our algorithm is always correct?
  - Proof techniques (induction, proof-by-contradiction etc).
- For those of you that took INF2-IADS: We did this a lot there!

Running Time /
Time Complexity

Running Time / Time Complexity

- Different computers have different speeds.

Running Time / Time Complexity

- Different computers have different speeds.
- **Random Access Machine (RAM) model.**

Running Time / Time Complexity

- Different computers have different speeds.
- **Random Access Machine (RAM) model.**
- Instructions:

Running Time / Time Complexity

- Different computers have different speeds.
- **Random Access Machine (RAM) model.**
- Instructions:
  - Arithmetic (add, subtract, multiply, etc).

Running Time / Time Complexity

- Different computers have different speeds.
- **Random Access Machine (RAM) model.**
- Instructions:
  - Arithmetic (add, subtract, multiply, etc).
  - Data movement (load, store, copy, etc).

Running Time / Time Complexity

- 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).

Running Time / Time Complexity

- 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.

Running Time / Time Complexity

- 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 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
```

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 ← 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
             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 $A[1 \,.\,.\, j-1]$}
        $i \leftarrow j - 1$ **n-1 times**
        WHILE $i > 0$ and $A[i] >$ key $\sum_{j=2}^{n} t_j$ **times**
            DO $A[i+1] \leftarrow A[i]$
                $i \leftarrow i - 1$
        $A[i+1] \leftarrow$ 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 $A[1\,..\,j-1]$}
        $i \leftarrow j - 1$ **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

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 $A[1\;..\;j-1]$}
        $i \leftarrow j - 1$ **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

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

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

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                       Σ_{j=2}^{n} t_j times
             DO A[i +1] ← A[i]                          Σ_{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?

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,** $t_j = 1$

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              Σ_{j=2}^{n} t_j  times
              DO A[i +1] ← A[i]
                 i ← i − 1                      Σ_{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?

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? **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      Σ_{j=2}^{n} t_j  times
                    DO A[i +1] ← A[i]       Σ_{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 ← 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   Σ_{j=2}^{n} t_j  times
            DO A[i +1] ← A[i]
               i ← i − 1          Σ_{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,** $t_j = 1$ 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 / Space Complexity

Memory Usage / Space Complexity

- Each memory cell can hold one element of the input.

Memory Usage / Space Complexity

- 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).

Memory Usage / Space Complexity

- 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).
- Q: 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 (usually the uniform distribution - all inputs equally likely).

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

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              Σ_{j=2}^{n} t_j  times
              DO A[i +1] ← A[i]
                 i ← i − 1                      Σ_{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)

1.   FOR j ← 2 TO length[A]                          n times
2.        DO key ← A[j]                              n-1 times
3.             {Put A[j] into the sorted sequence A[1 . . j − 1]}
4.             i ← j − 1                              n-1 times
5.             WHILE i > 0 and A[i] > key             Σ_{j=2}^{n} t_j times
6.                  DO A[i +1] ← A[i]                 Σ_{j=2}^{n} (t_j − 1) times
7.                     i ← i − 1
8.             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 $A[1\,..\,j-1]$}
    $i \leftarrow j - 1$ 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].

Example: Average Running Time of InsertionSort

INSERTION_SORT (A)

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

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 (next lecture).