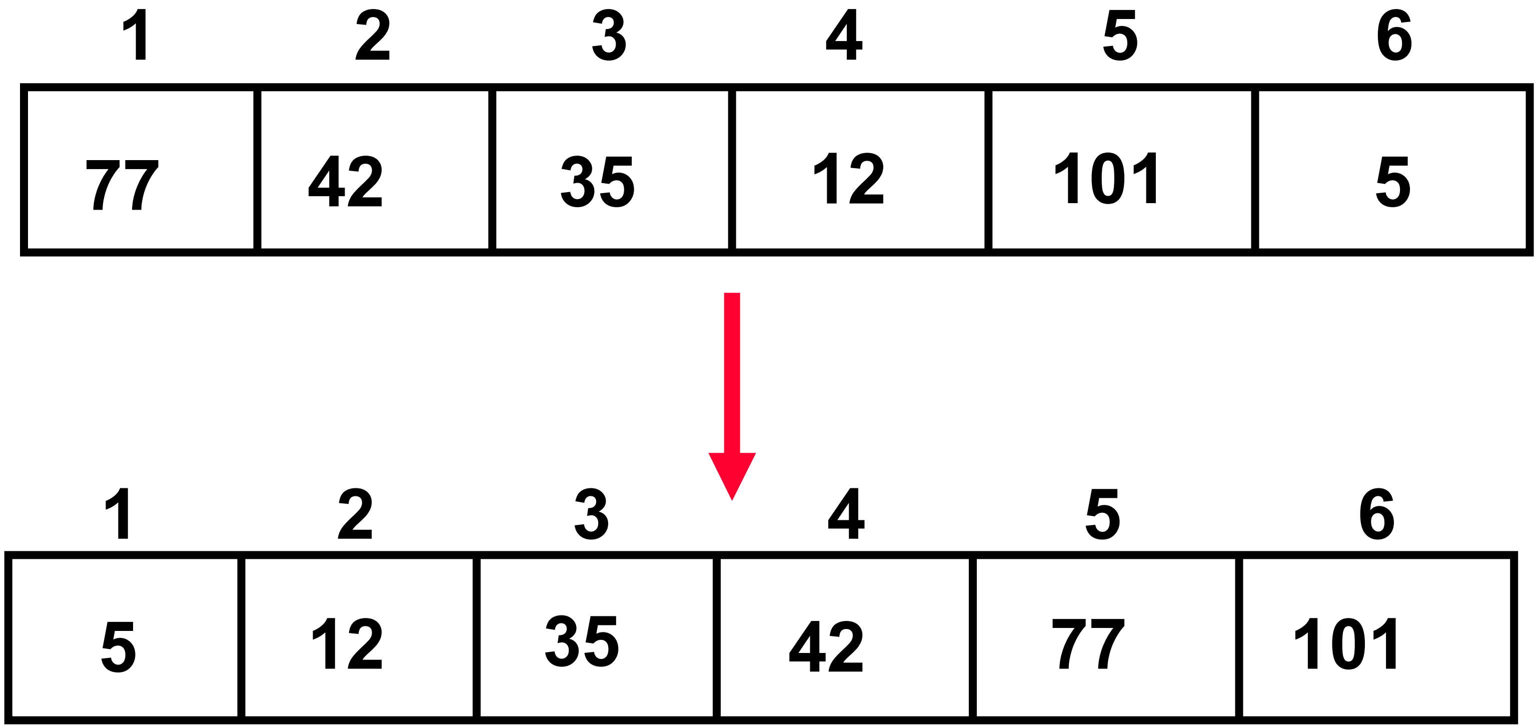
BUBBLE SORT
SELECTION SORT
INSERTION SORT

SORTING

- **Sorting takes an unordered collection and makes it an ordered one.**



1	2	3	4	5	6
77	42	35	12	101	5

1	2	3	4	5	6
5	12	35	42	77	101

Bubble Sort

Bubble Sorting is an algorithm in which we are comparing first two values and put the larger one at higher index.

It takes next two values compare these values and place larger value at higher index.

This process do iteratively until the largest value is not reached at last index. Then start again from zero index up to n-1 index.

The algorithm follows the same steps iteratively unlit elements are not sorted.

Bubble sort is also known as exchange sort. Bubble sort is a simplest sorting algorithm

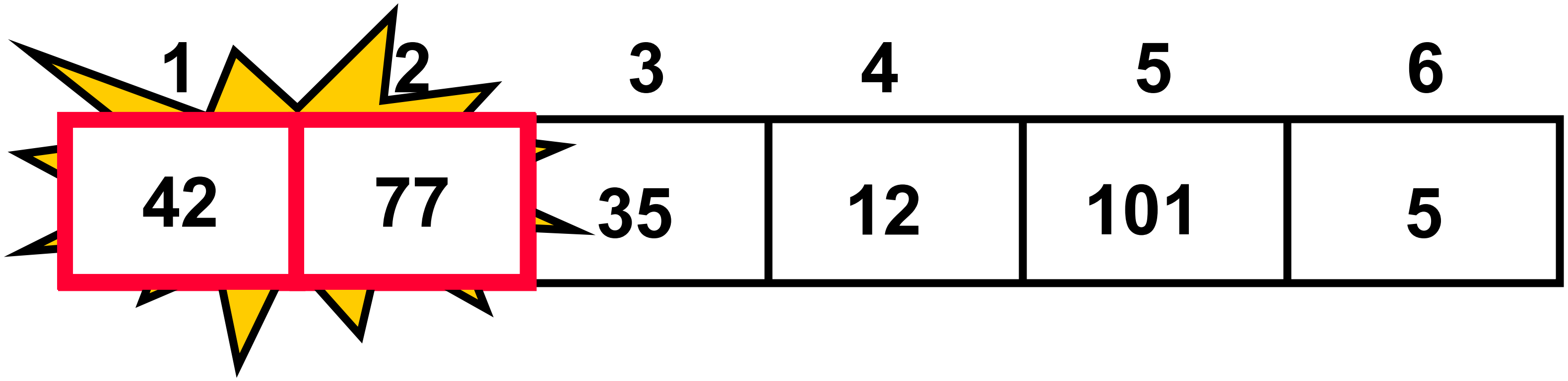
"BUBBLING UP" THE LARGEST ELEMENT

- **Move from the start to the end**
- **"Bubble" the largest value to the end using pair-wise comparisons and swapping**

1	2	3	4	5	6
77	42	35	12	101	5

"BUBBLING UP" THE LARGEST ELEMENT

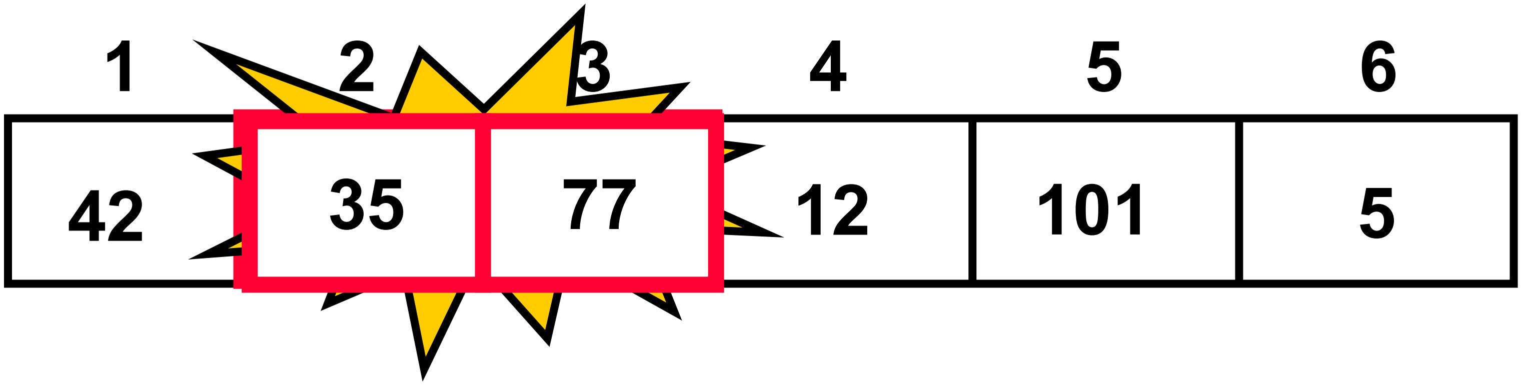
- **Move from the start to the end**
- **"Bubble" the largest value to the end <u>using pair-wise comparisons and swapping</u>**



1	2	3	4	5	6
42	77	35	12	101	5

"BUBBLING UP" THE LARGEST ELEMENT

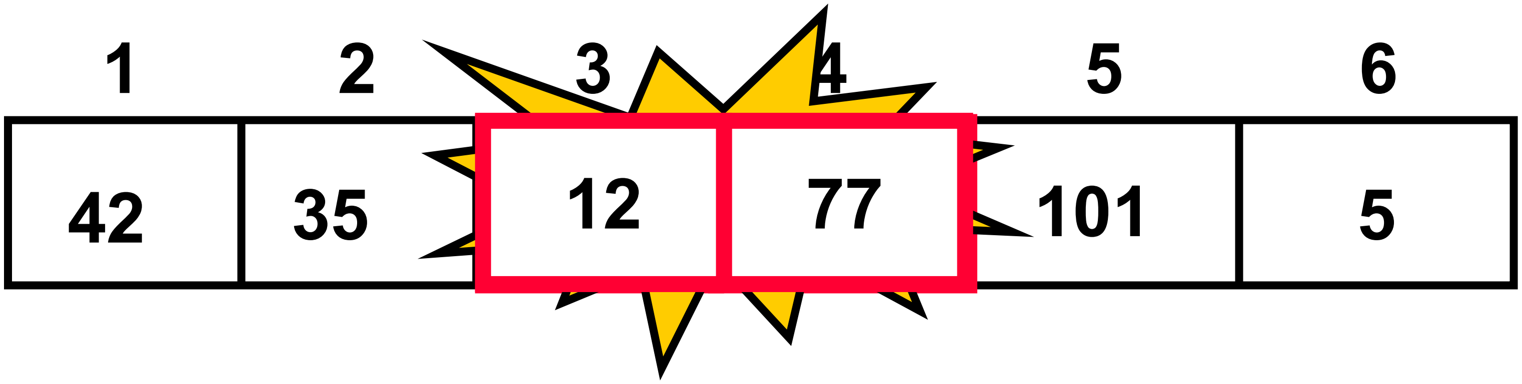
- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



1	2	3	4	5	6
42	35	77	12	101	5

"BUBBLING UP" THE LARGEST ELEMENT

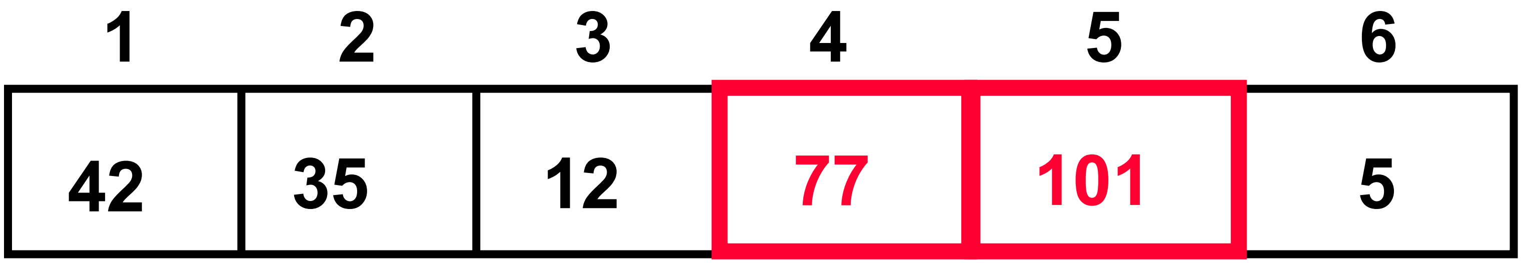
- Move from the start to the end
- "Bubble" the largest value to the end using pair-wise comparisons and swapping



1	2	3	4	5	6
42	35	12	77	101	5

"BUBBLING UP" THE LARGEST ELEMENT

- **Move from the start to the end**
- **“Bubble” the largest value to the end using pair-wise comparisons and swapping**

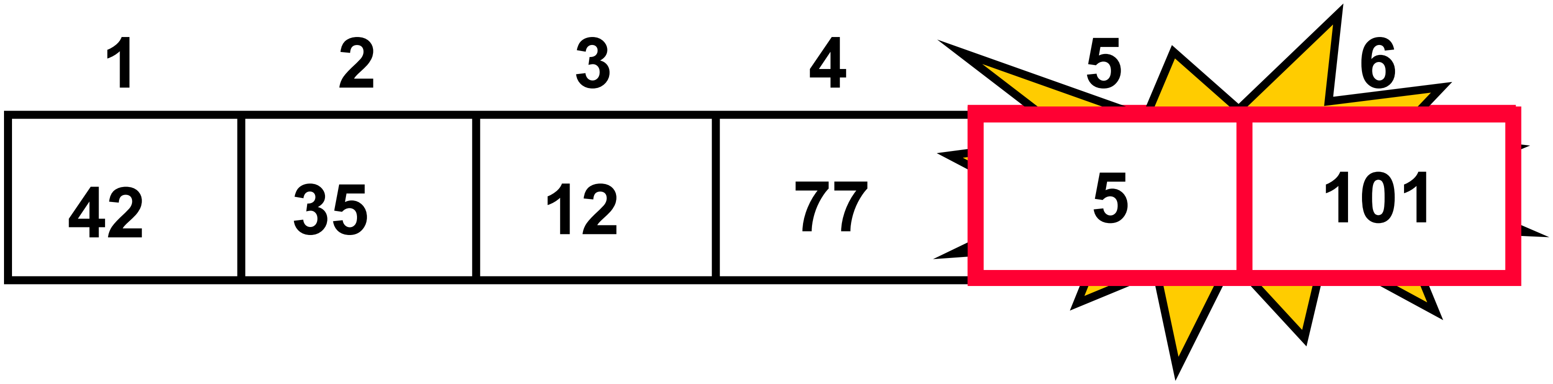


1	2	3	4	5	6
42	35	12	77	101	5

No need to swap

"BUBBLING UP" THE LARGEST ELEMENT

- Move from the start to the end
- "Bubble" the largest value to the end using pair-wise comparisons and swapping



1	2	3	4	5	6
42	35	12	77	5	101

"BUBBLING UP" THE LARGEST ELEMENT

- **Move from the start to the end**
- **"Bubble" the largest value to the end using pair-wise comparisons and swapping**

1	2	3	4	5	6
42	35	12	77	5	101

Largest value correctly placed

- **Notice that only the largest value is correctly placed**
- **All other values are still out of order**
- **So we need to repeat this process**

1	2	3	4	5	6
42	35	12	77	5	101

Largest value correctly placed

REPEAT "BUBBLE UP" HOW MANY TIMES?

- **If we have N elements…**
- **And if each time we bubble an element, we place it in its correct location…**
- **Then we repeat the "bubble up" process N – 1 times.**
- **This guarantees we'll correctly place all N elements.**

“BUBBLING” ALL THE ELEMENTS

N - 1

1	2	3	4	5	6
42	35	12	77	5	101

1	2	3	4	5	6
35	12	42	5	77	101

1	2	3	4	5	6
12	35	5	42	77	101

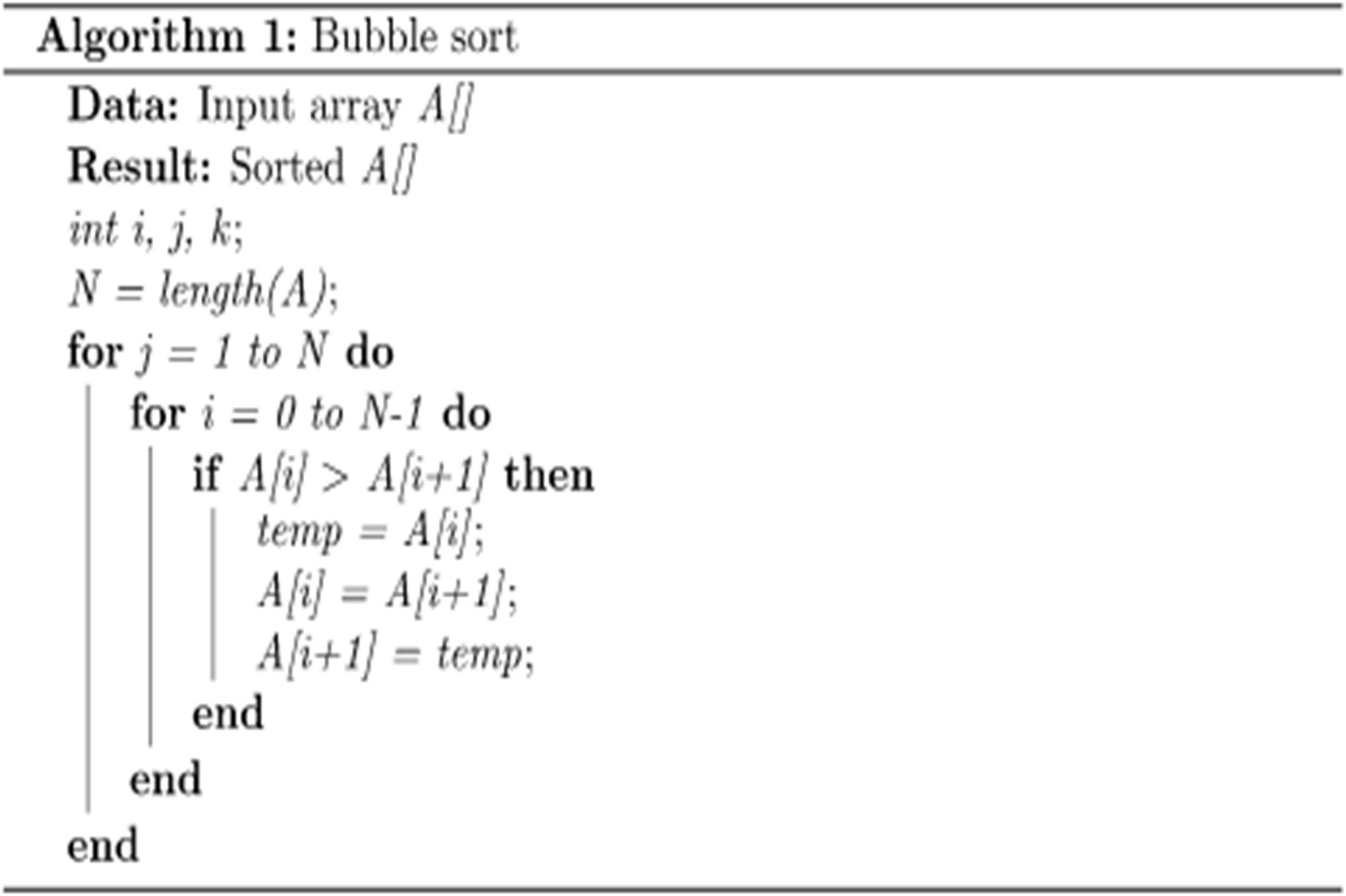
1	2	3	4	5	6
12	5	35	42	77	101

1	2	3	4	5	6
5	12	35	42	77	101

SUMMARY

- **"Bubble Up" algorithm will move largest value to its correct location (to the right)**
- **Repeat "Bubble Up" until all elements are correctly placed:**
  - **Maximum of N-1 times**
  - **Can finish early if no swapping occurs**

BUBBLE SORT ALGORITHM



Algorithm 1: Bubble sort

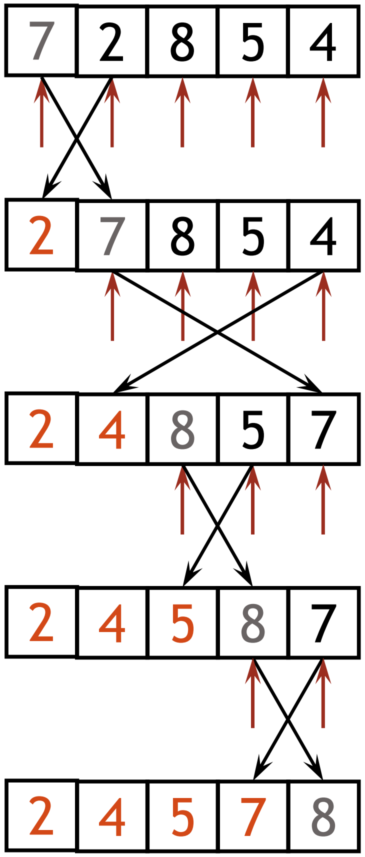
```
Data: Input array $A[]$
Result: Sorted $A[]$
$int\ i, j, k;$
$N = length(A);$
for $j = 1$ to $N$ do
    for $i = 0$ to $N-1$ do
        if $A[i] > A[i+1]$ then
            $temp = A[i];$
            $A[i] = A[i+1];$
            $A[i+1] = temp;$
        end
    end
end
```

SELECTION SORT

SELECTION SORT

- Given an array of length n,
  - Search elements 0 through n-1 and select the smallest
    - Swap it with the element in location 0
  - Search elements 1 through n-1 and select the smallest
    - Swap it with the element in location 1
  - Search elements 2 through n-1 and select the smallest
    - Swap it with the element in location 2
  - Search elements 3 through n-1 and select the smallest
    - Swap it with the element in location 3
  - Continue in this fashion until there’s nothing left to search

EXAMPLE AND ANALYSIS OF SELECTION SORT



Selection Sort

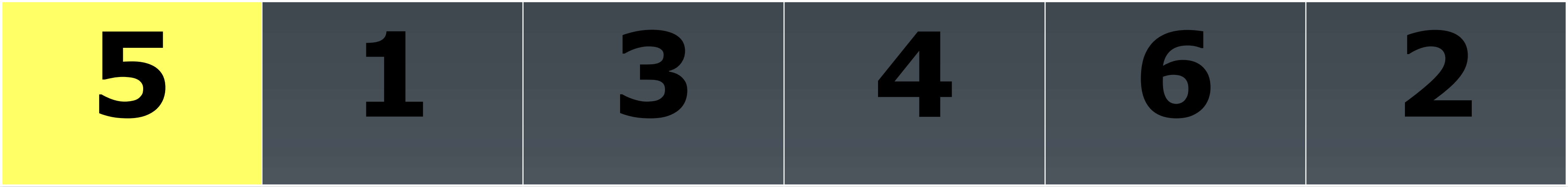
5	**1**	**3**	**4**	**6**	**2**

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



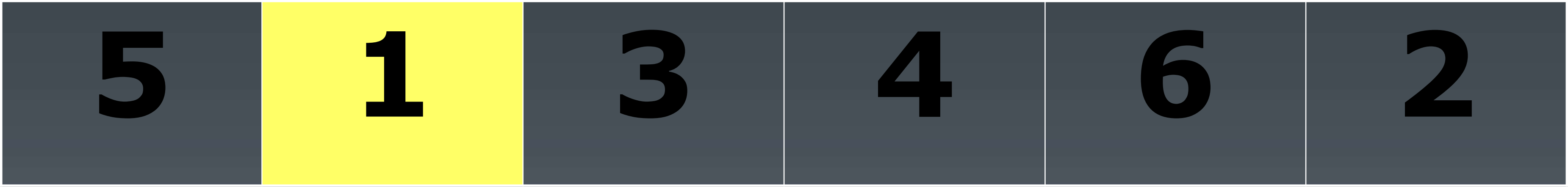
5	**1**	**3**	**4**	**6**	**2**

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



5	1	3	4	6	2

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

5	1	3	4	6	2

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

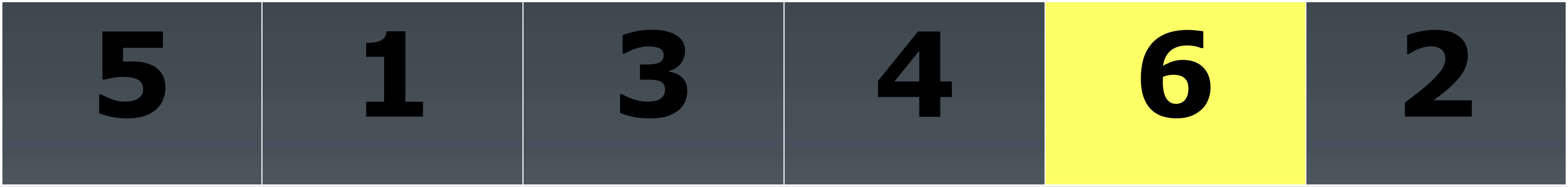
5	1	3	4	6	2

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



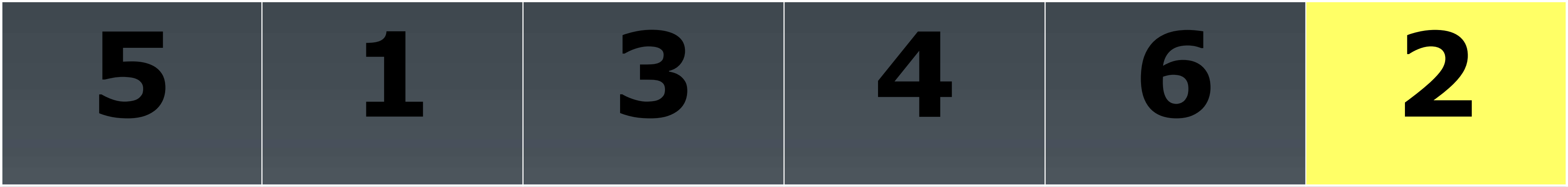
5	1	3	4	6	2

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



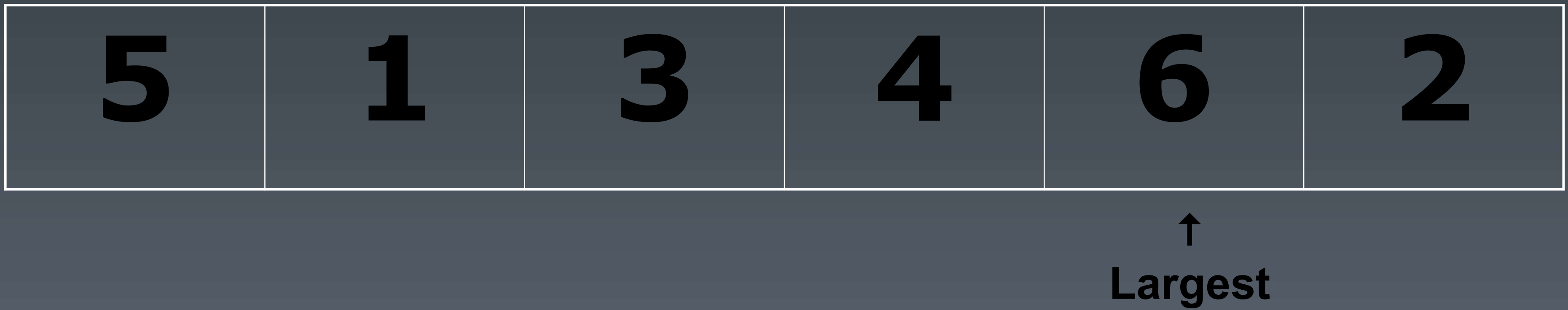
5	**1**	**3**	**4**	**6**	**2**

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



5	**1**	**3**	**4**	**6**	**2**

↑

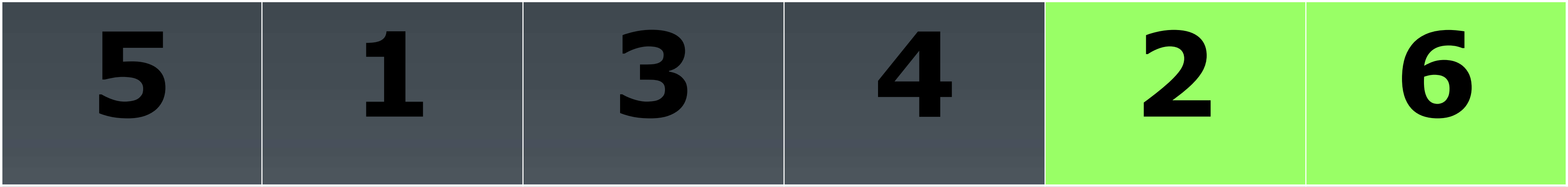
Largest

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



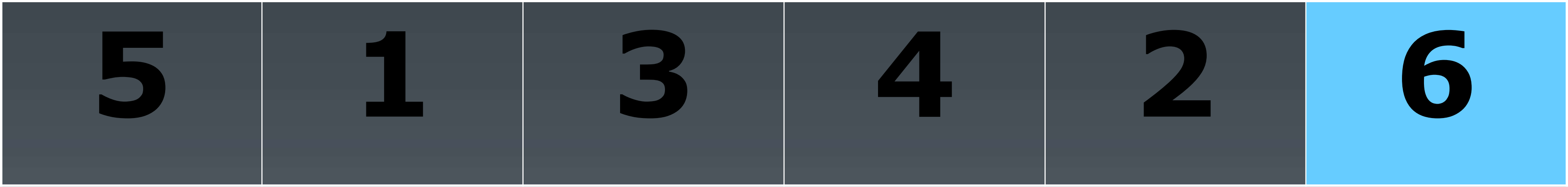
5	1	3	4	2	6

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



5	1	3	4	2	6

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

5	1	3	4	2	6

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

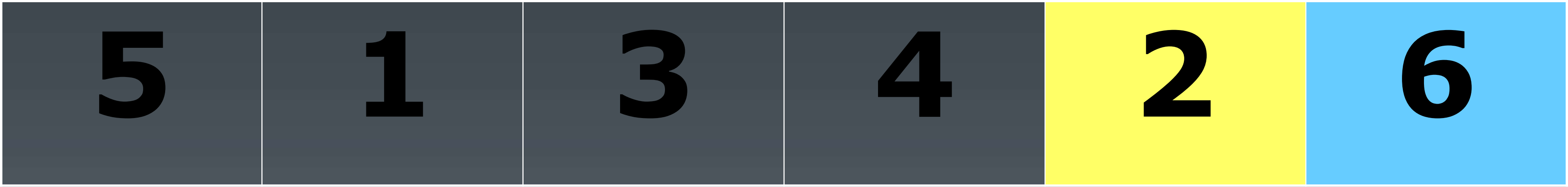
5	1	3	4	2	6

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



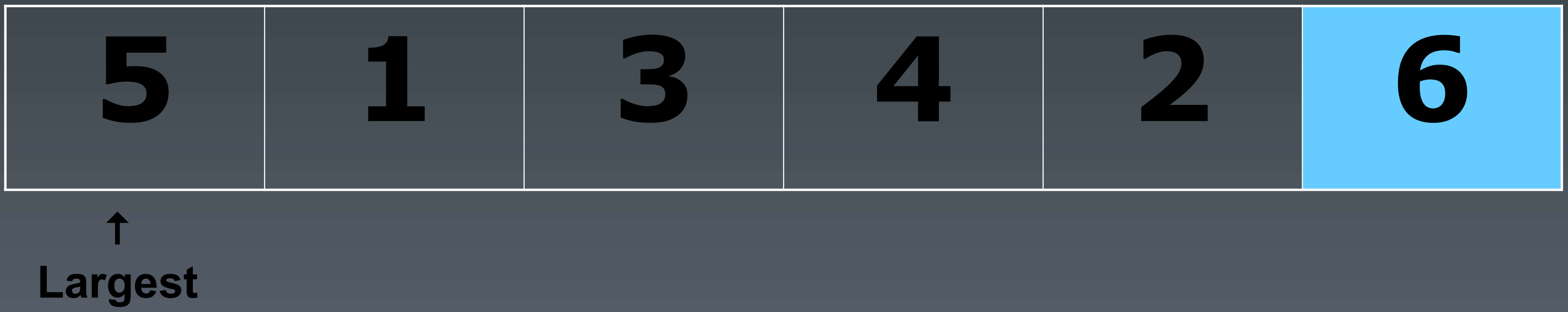
5	1	3	4	2	6

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



5	1	3	4	2	6

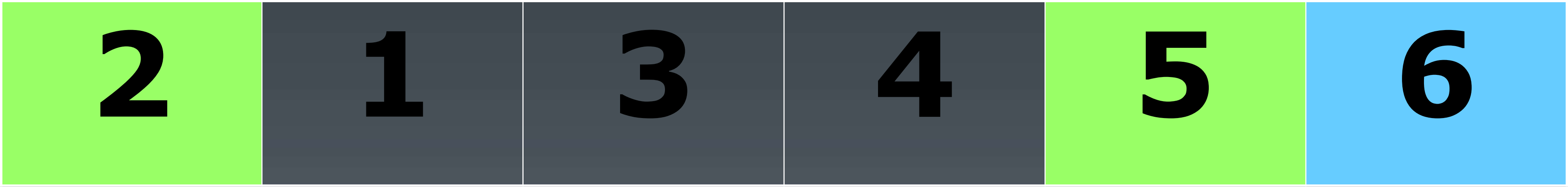
↑
Largest

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



2	1	3	4	5	6

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

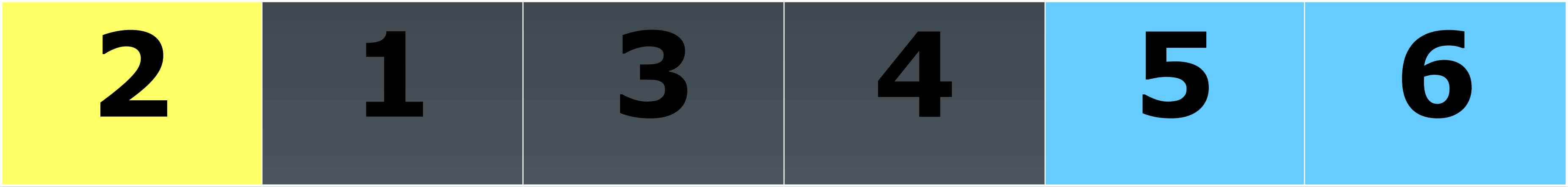
2	1	3	4	5	6

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



2	1	3	4	5	6

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

2	1	3	4	5	6

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

2	1	3	4	5	6

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

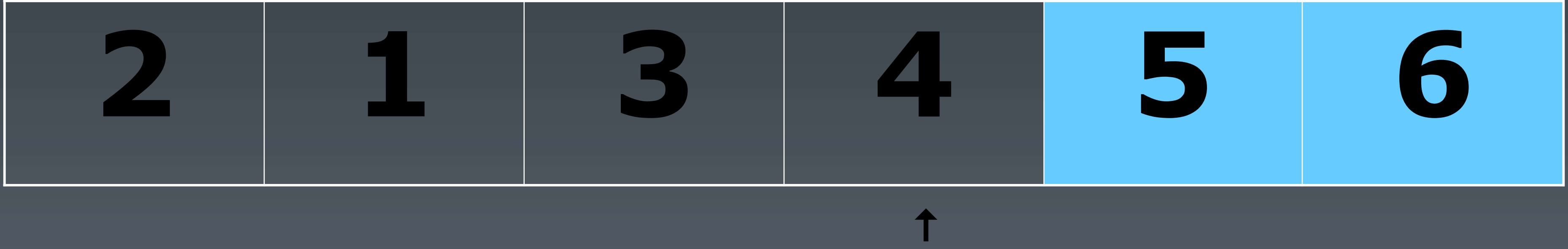
2	1	3	4	5	6

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



2	1	3	4	5	6

↑ **Largest**

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

2	1	3	4	5	6

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

2	1	3	4	5	6

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

2	1	3	4	5	6

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

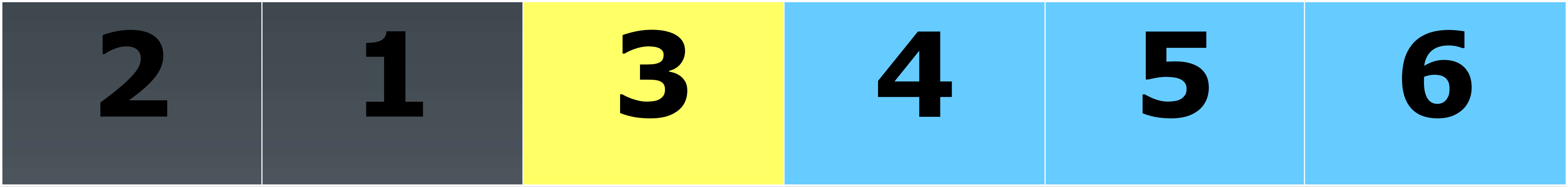
2	1	3	4	5	6

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



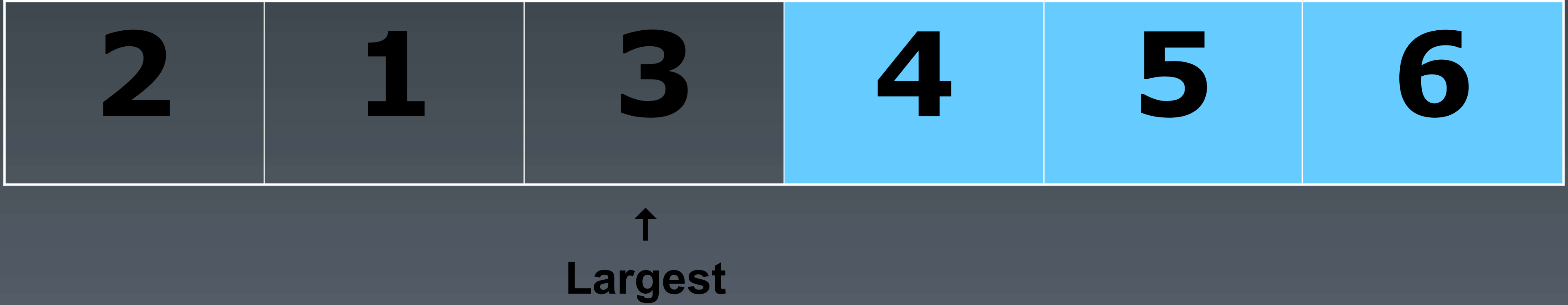
2	1	3	4	5	6

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

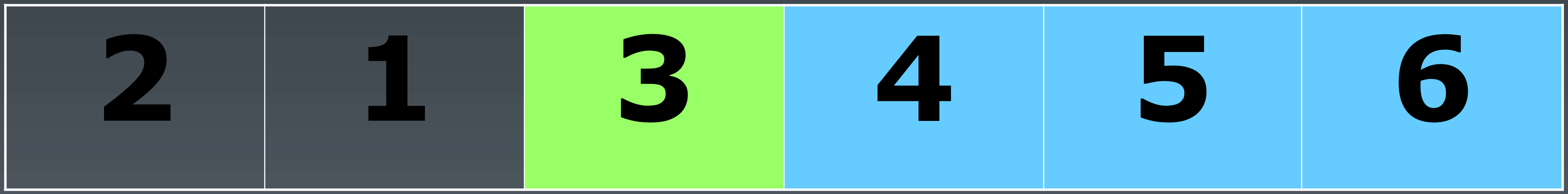


2	1	3	4	5	6

↑
Largest

- 🟨 Comparison
- 🟩 Data Movement
- 🟦 Sorted

Selection Sort



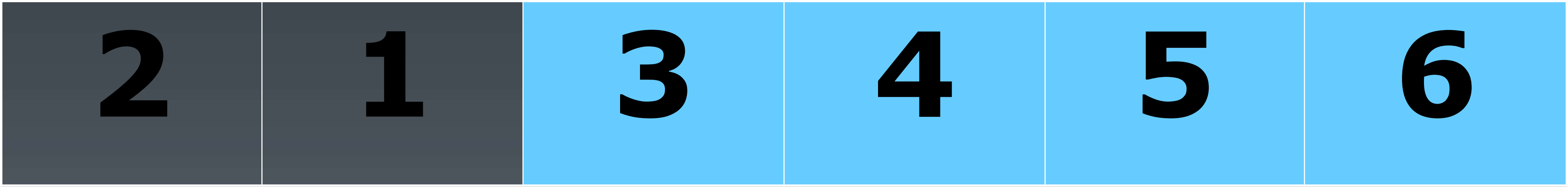
2	1	3	4	5	6

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



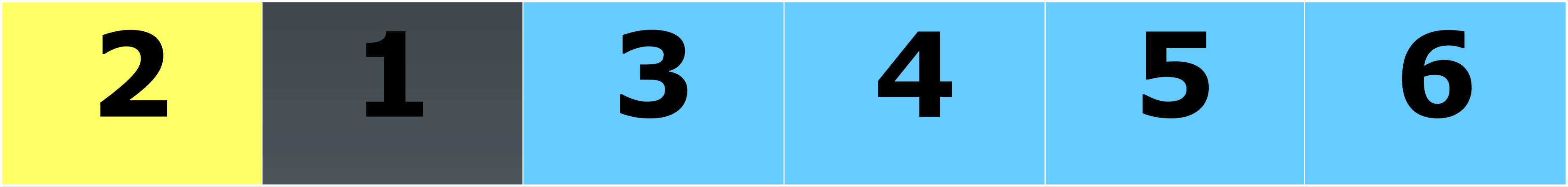
2	1	3	4	5	6

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



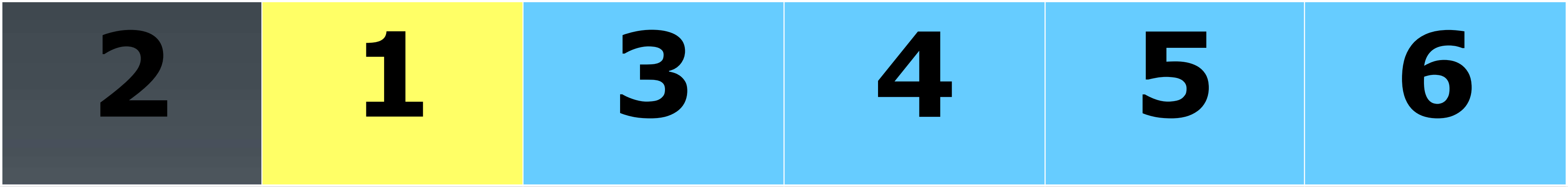
2	1	3	4	5	6

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



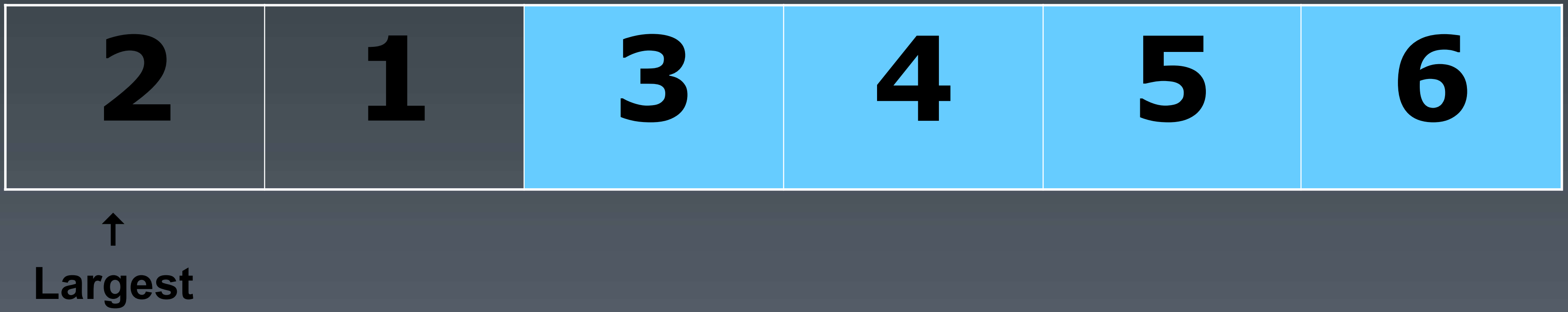
2	1	3	4	5	6

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



2	1	3	4	5	6

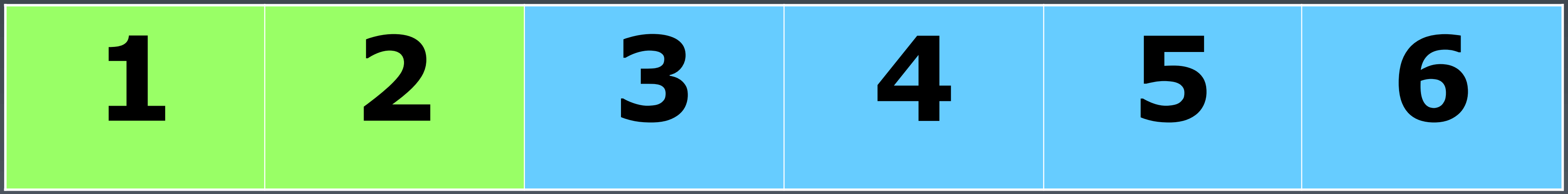
↑ **Largest**

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



1	2	3	4	5	6

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

1	**2**	**3**	**4**	**5**	**6**

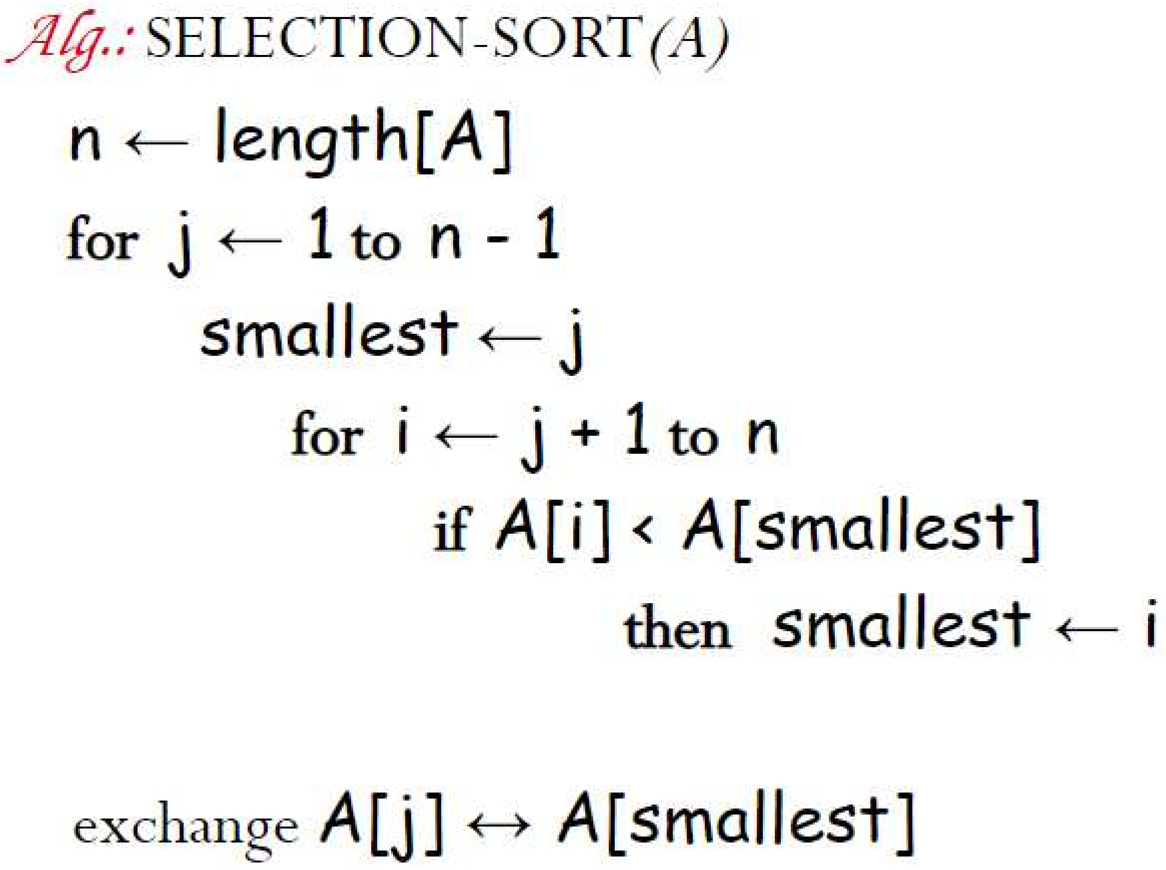
DONE!

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort: Algorithm



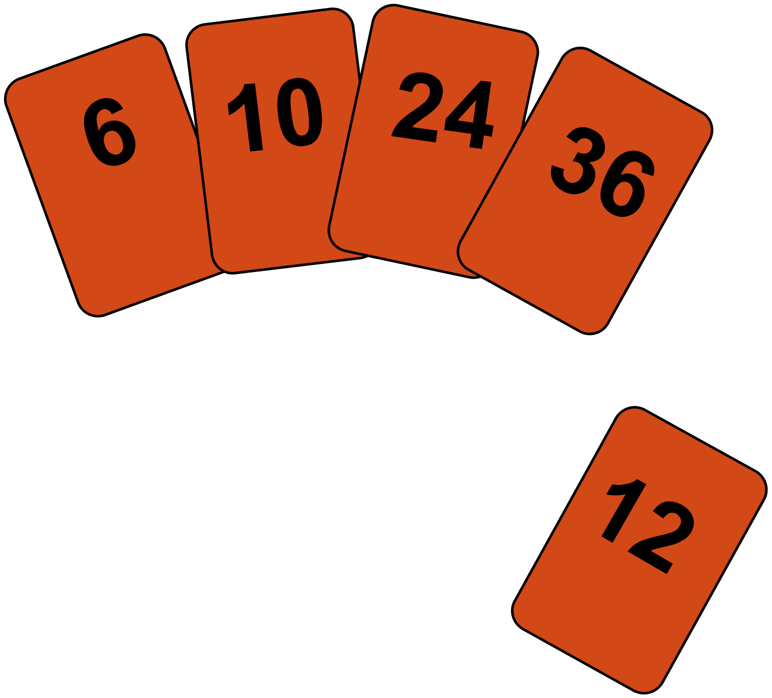
```
Alg.: SELECTION-SORT(A)
  n ← length[A]
  for j ← 1 to n - 1
        smallest ← j
            for i ← j + 1 to n
                  if A[i] < A[smallest]
                           then smallest ← i

  exchange A[j] ↔ A[smallest]
```

INSERTION SORT

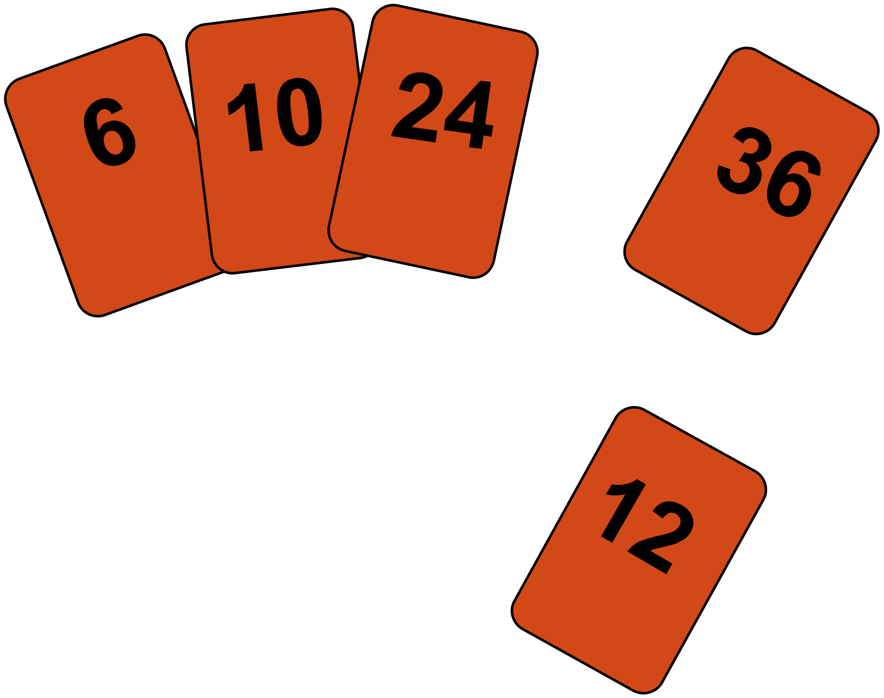
The Idea of the insertion sort is similer to the Idea of sorting the Playing cards .

INSERTION SORT

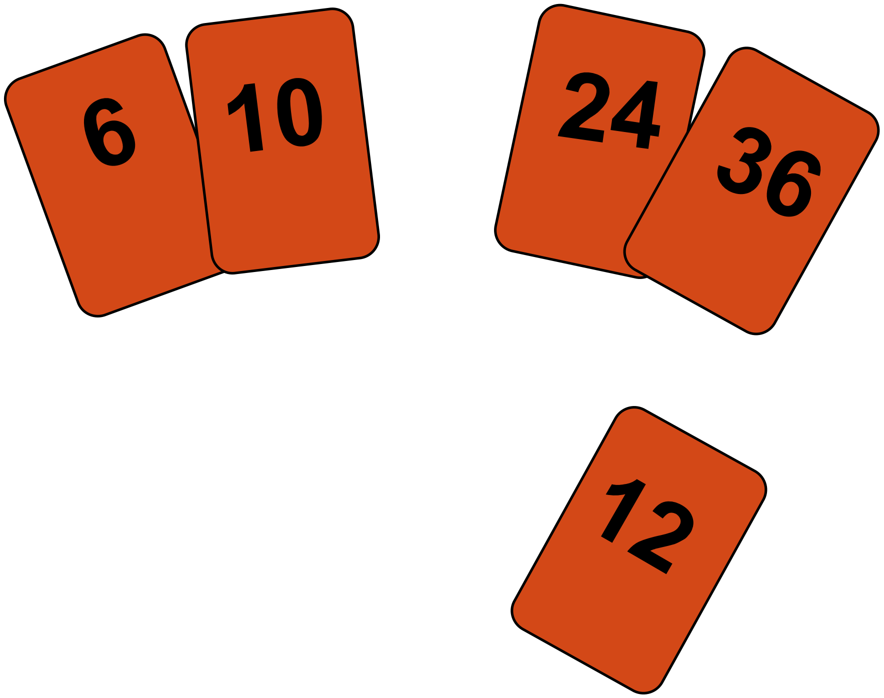


To insert 12, we need to make room for it by moving first 36 and then 24.

INSERTION SORT



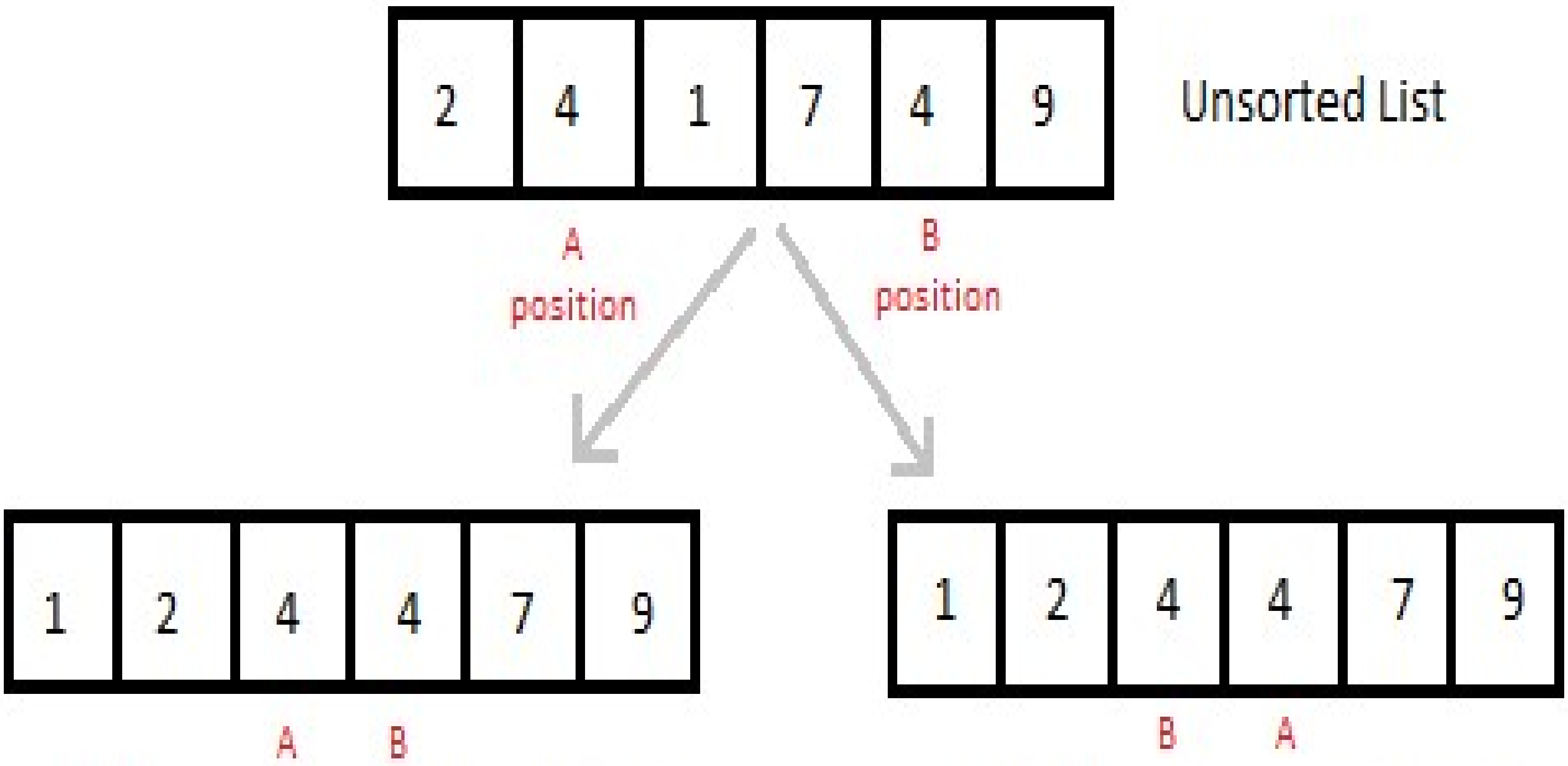
INSERTION SORT



INSERTION SORTING

It is a simple Sorting algorithm which sorts the array by shifting elements one by one. Following are some of the important characteristics of Insertion Sort.

- It has one of the simplest implementation
- It is efficient for smaller data sets, but very inefficient for larger lists.
- Insertion Sort is adaptive, that means it reduces its total number of steps if given a partially sorted list, hence it increases its efficiency.
- It is **Stable**, as it does not change the relative order of elements with equal keys



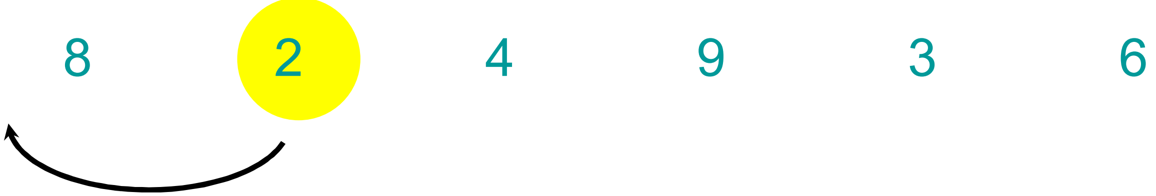
Stable Sort, because the order of equal elements is maintained in sorted list.

UnStable Sort, because the order of equal elements is not maintained in the sorted list.

EXAMPLE OF INSERTION SORT

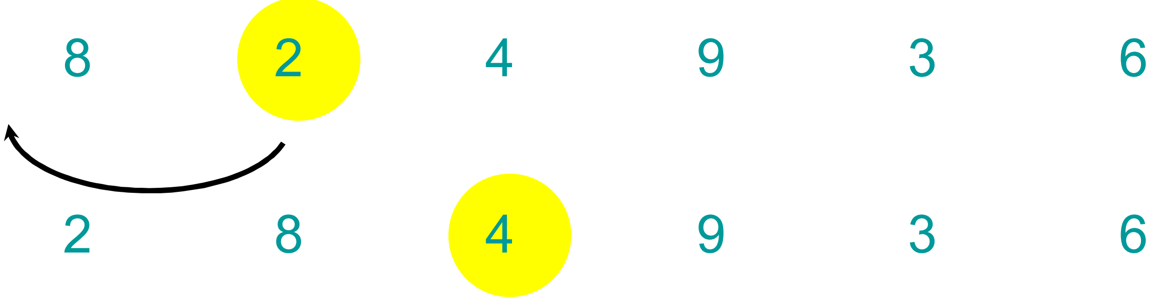
8 2 4 9 3 6

EXAMPLE OF INSERTION SORT



8 2 4 9 3 6

EXAMPLE OF INSERTION SORT



8 2 4 9 3 6

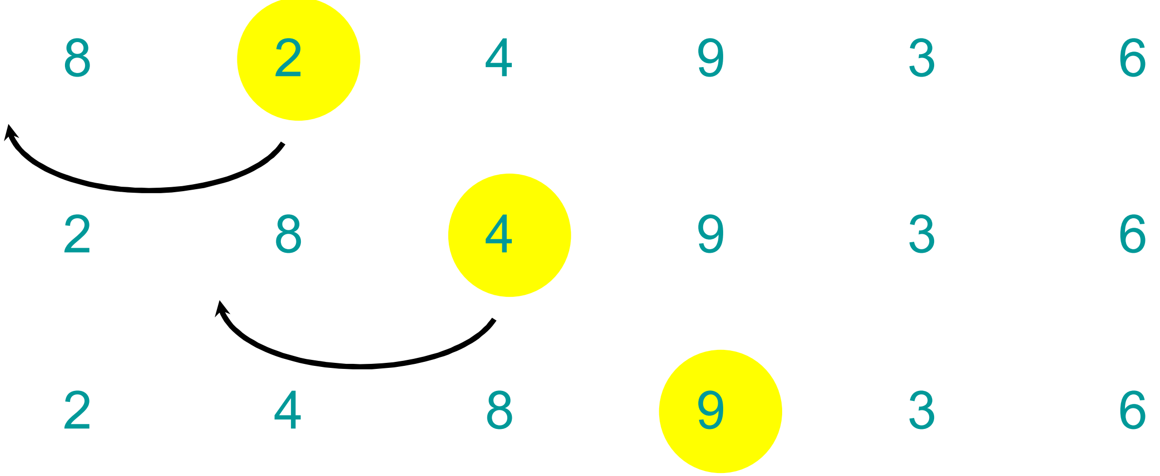
2 8 4 9 3 6

EXAMPLE OF INSERTION SORT

8 2 4 9 3 6

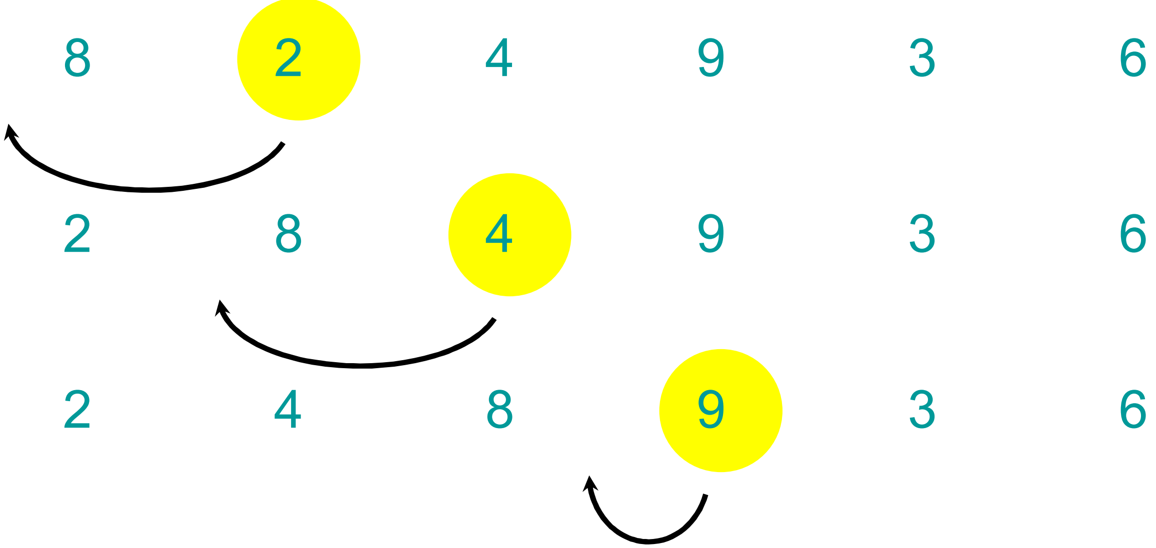
2 8 4 9 3 6

EXAMPLE OF INSERTION SORT



8	2	4	9	3	6
2	8	4	9	3	6
2	4	8	9	3	6

EXAMPLE OF INSERTION SORT

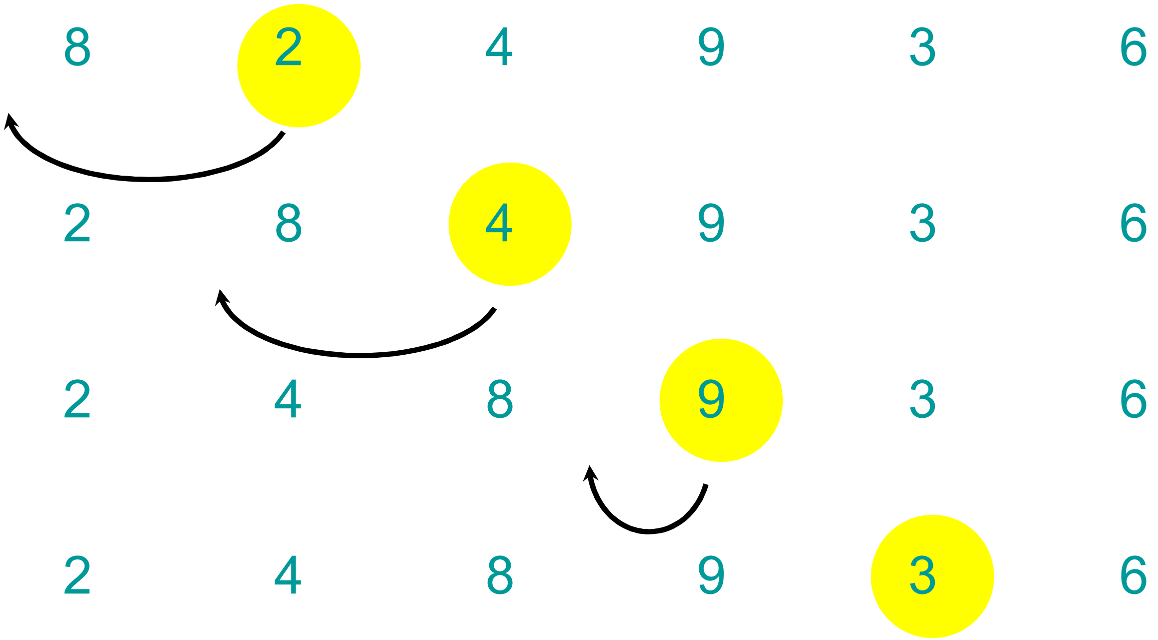


8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

EXAMPLE OF INSERTION SORT



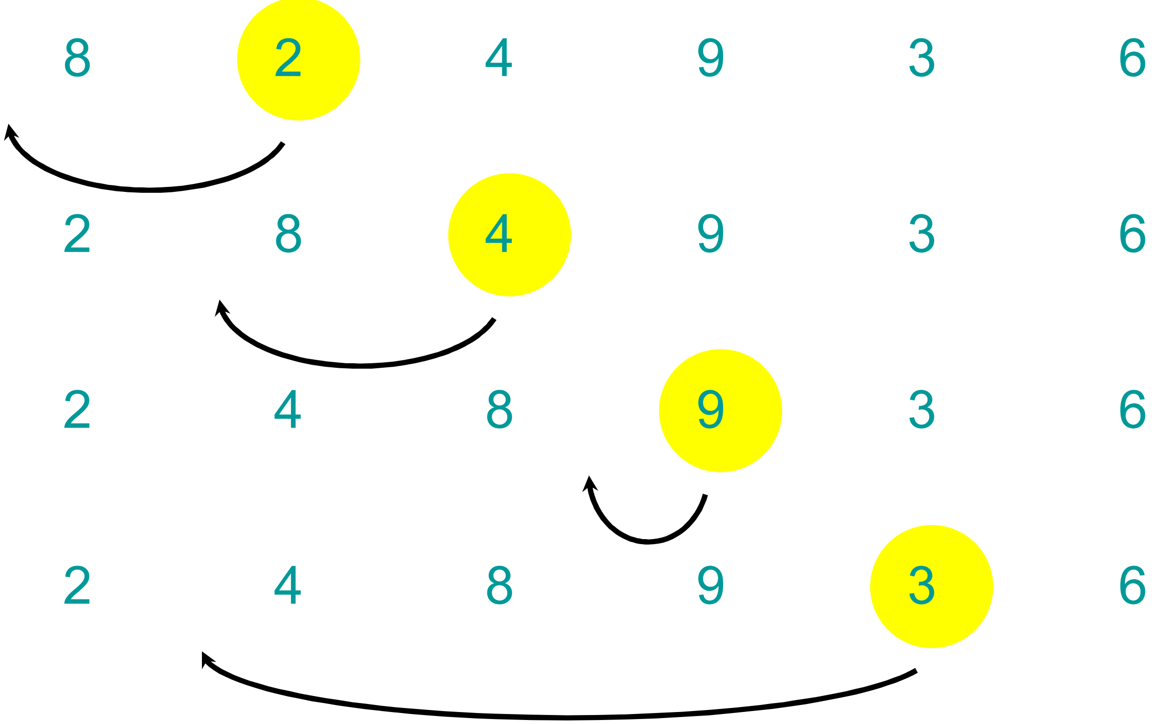
8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

2 4 8 9 3 6

EXAMPLE OF INSERTION SORT



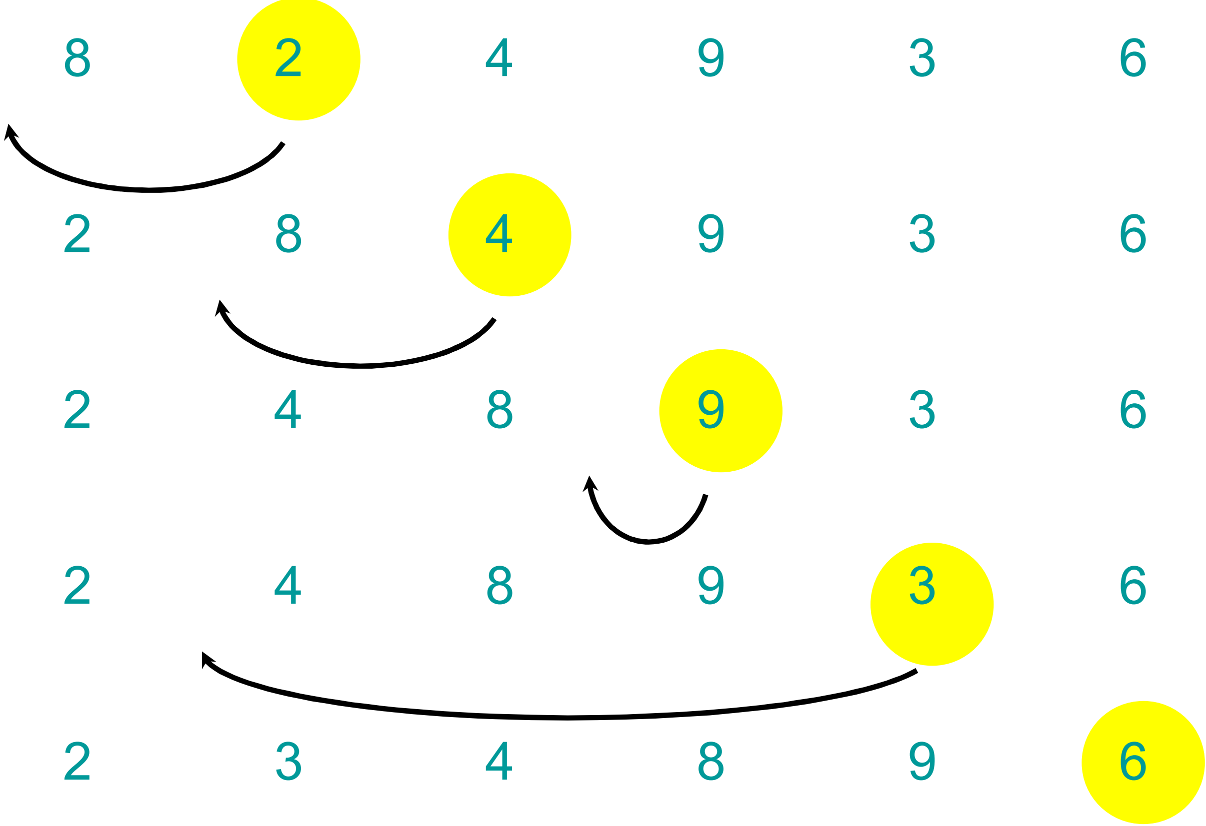
8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

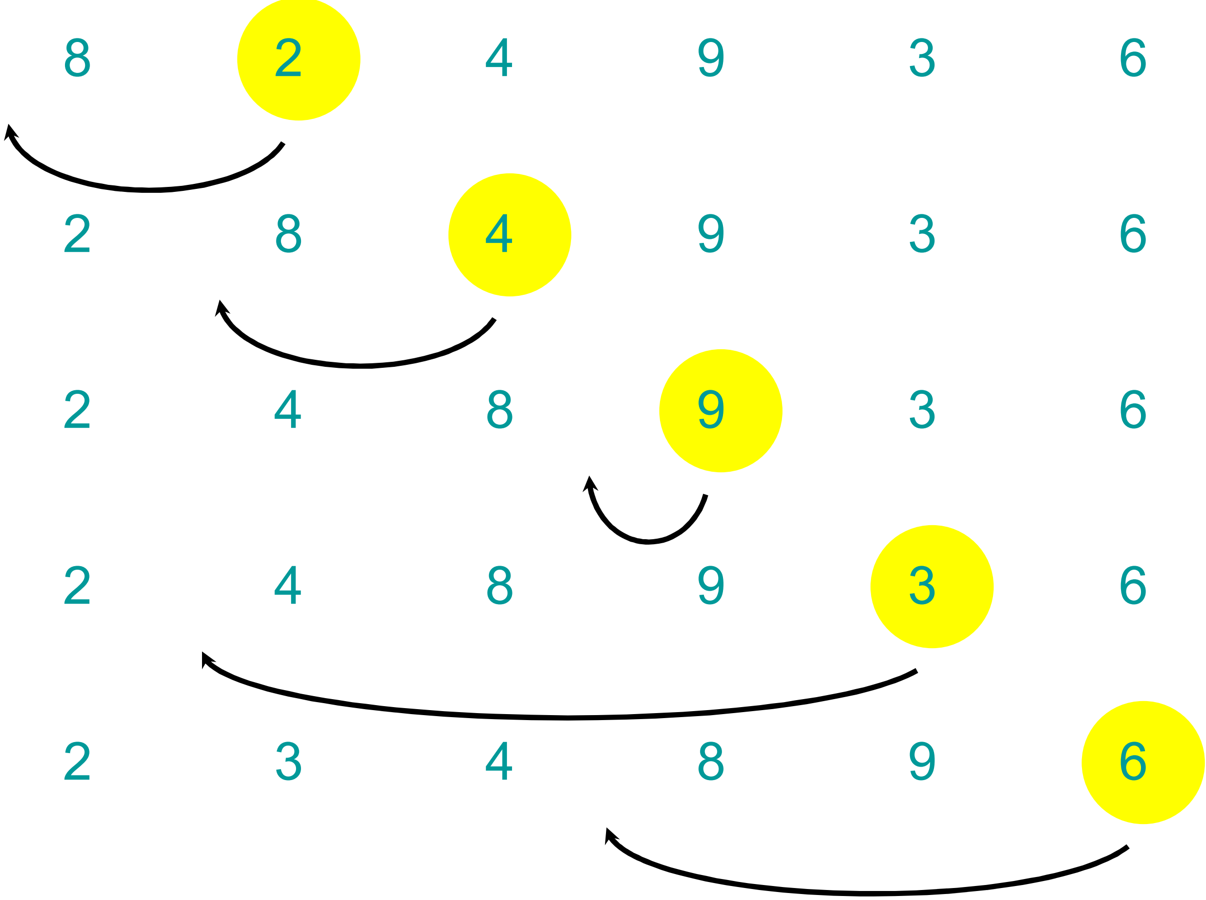
2 4 8 9 3 6

EXAMPLE OF INSERTION SORT



8	2	4	9	3	6
2	8	4	9	3	6
2	4	8	9	3	6
2	4	8	9	3	6
2	3	4	8	9	6

EXAMPLE OF INSERTION SORT



8 2 4 9 3 6

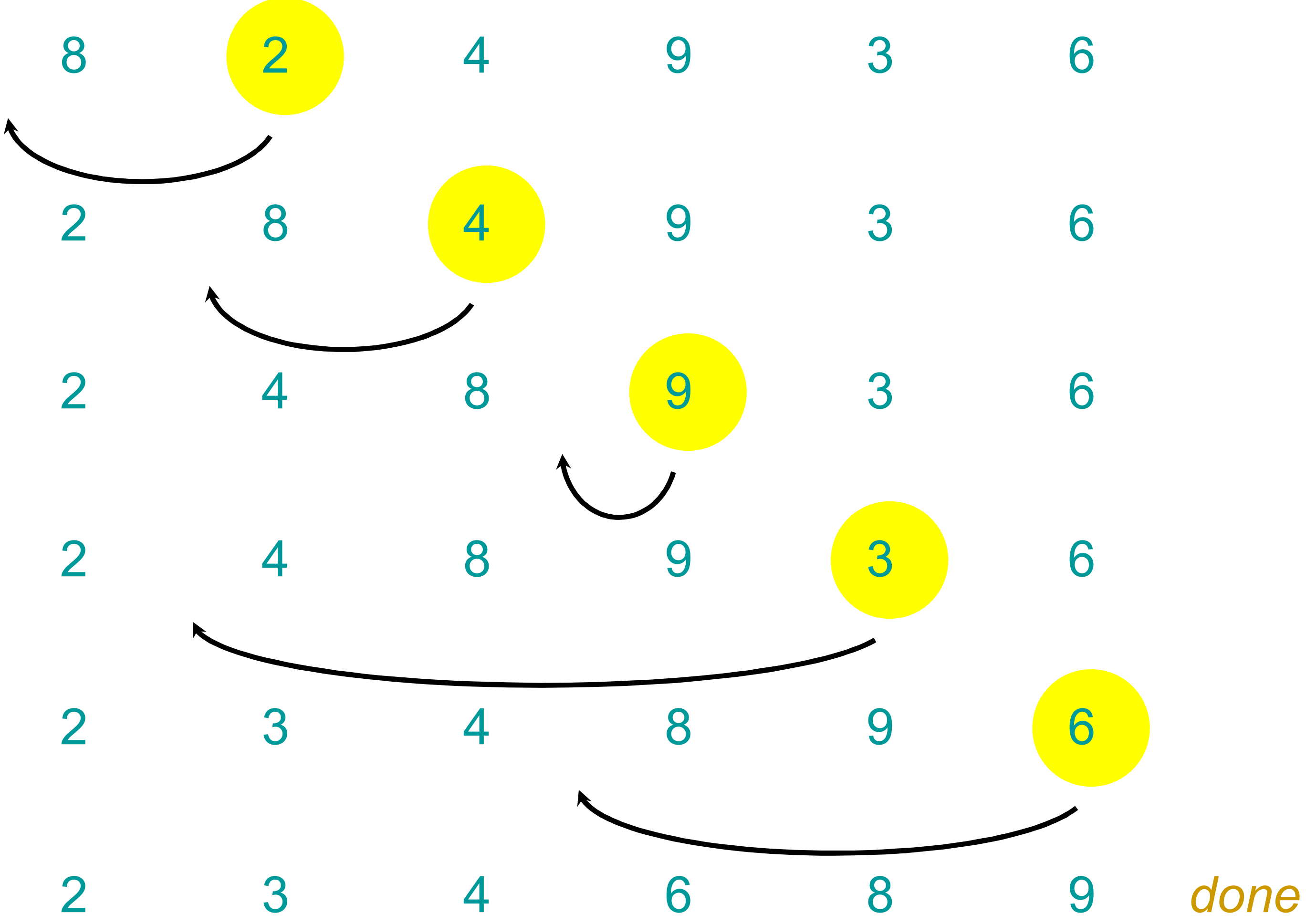
2 8 4 9 3 6

2 4 8 9 3 6

2 4 8 9 3 6

2 3 4 8 9 6

EXAMPLE OF INSERTION SORT



8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

2 4 8 9 3 6

2 3 4 8 9 6

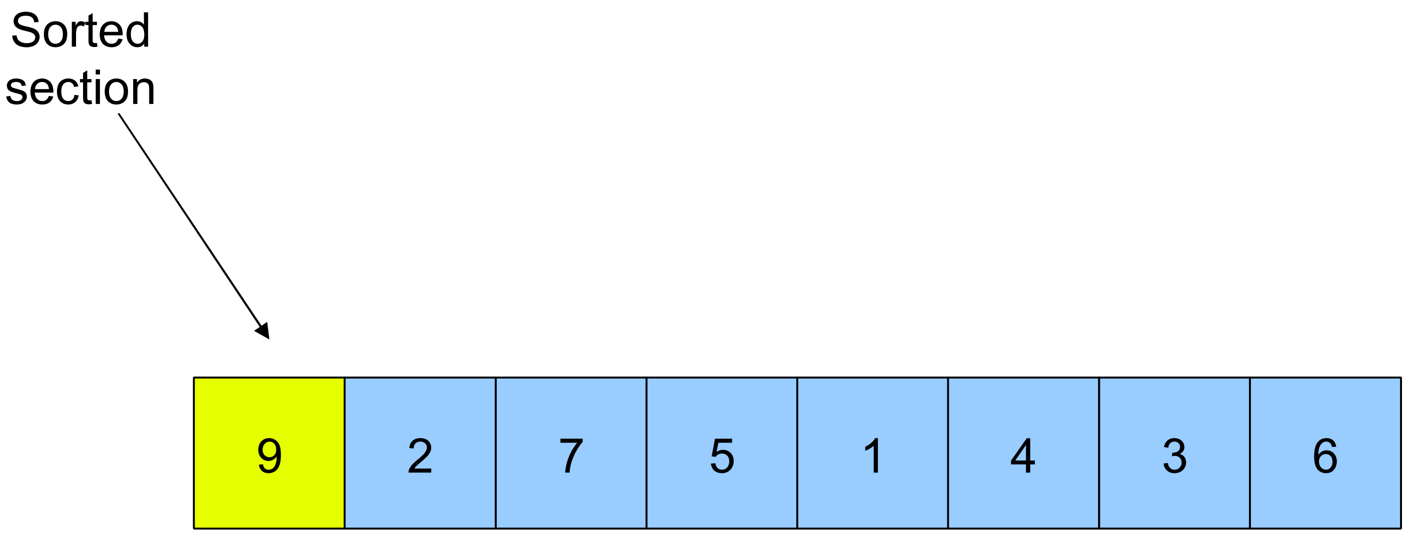
2 3 4 6 8 9 *done*

INSERTION SORT

Example :

9	2	7	5	1	4	3	6

INSERTION SORT

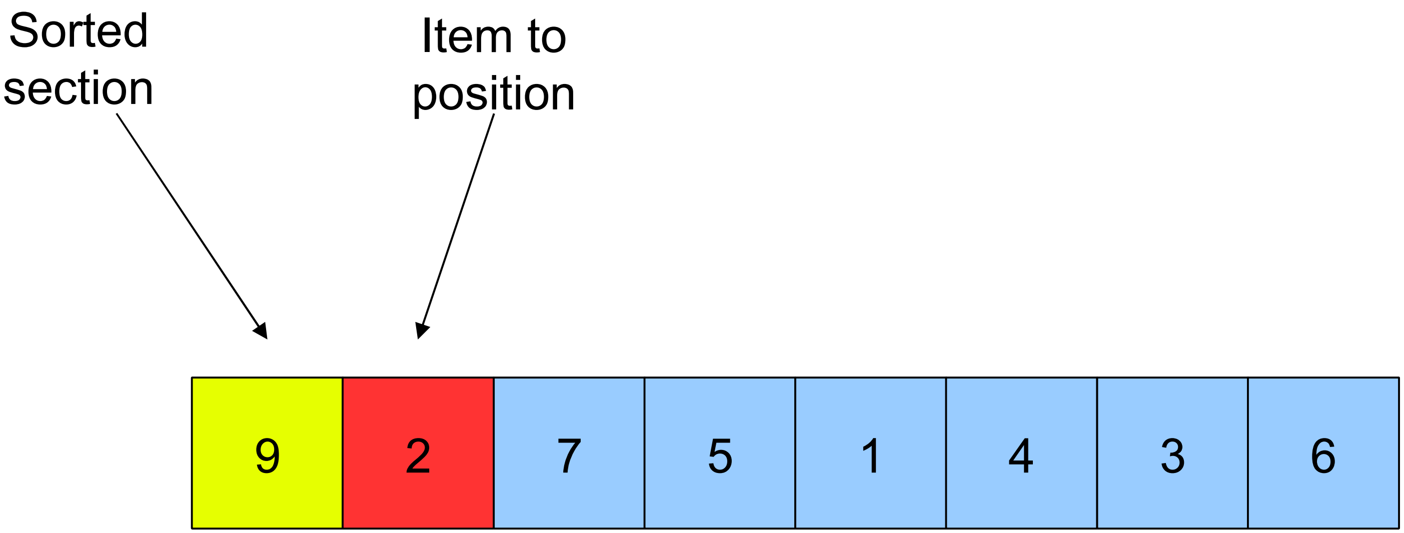


Sorted section

9	2	7	5	1	4	3	6

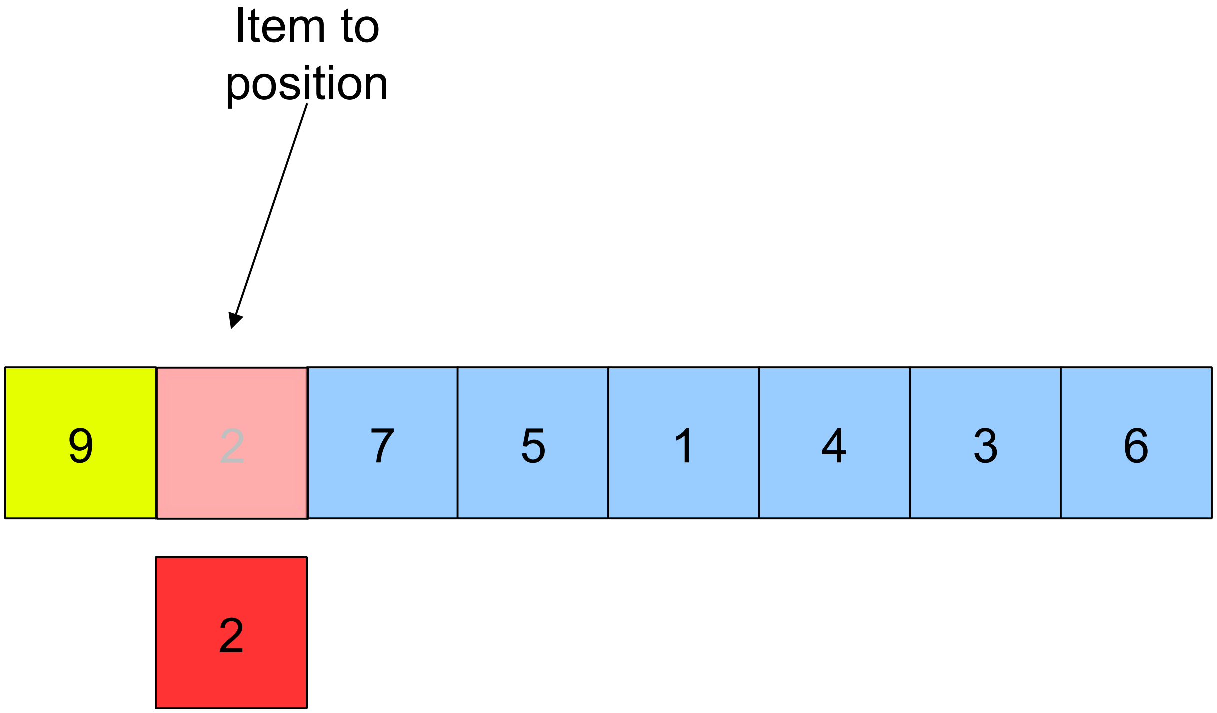
We start by dividing the array in two sections: a sorted section and an unsorted section. We put the first element as the only element in the sorted section, and the rest of the array is the unsorted section.

INSERTION SORT



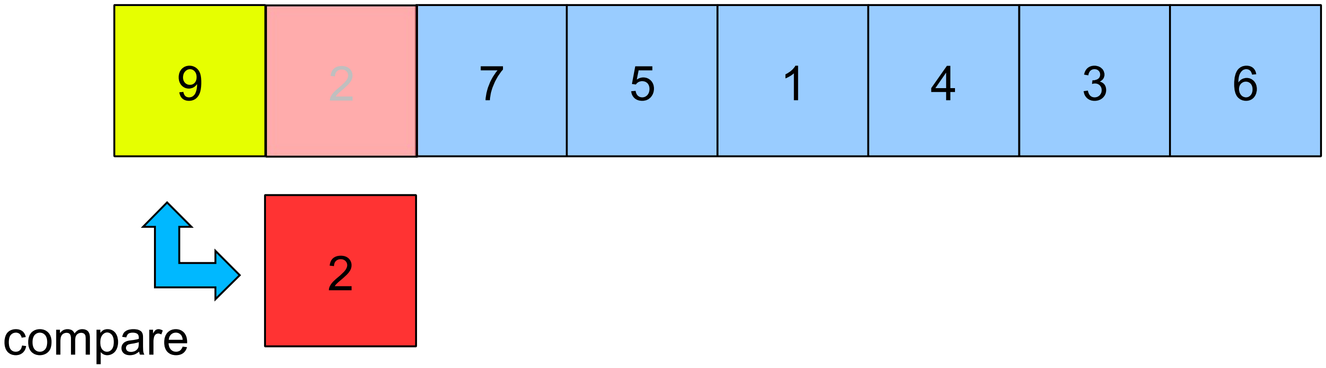
The first element in the unsorted section is the next element to be put into the correct position.

INSERTION SORT



We copy the element to be placed into another variable so it doesn't get overwritten.

INSERTION SORT



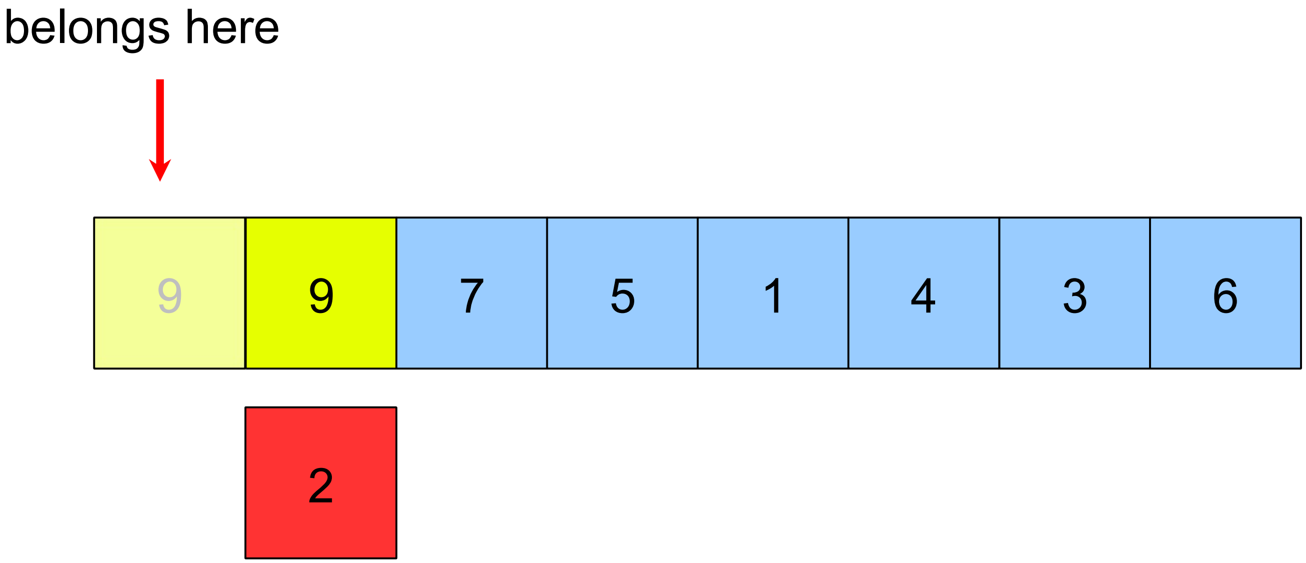
9	2	7	5	1	4	3	6

2

compare

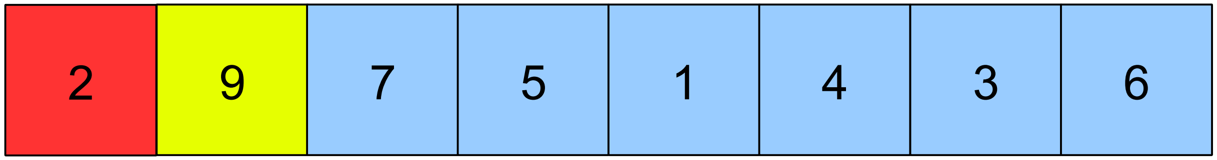
If the previous position is more than the item being placed, copy the value into the next position

INSERTION SORT



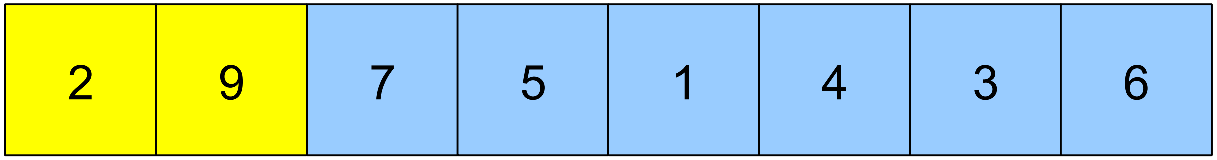
If there are no more items in the sorted section to compare with, the item to be placed must go at the front.

INSERTION SORT



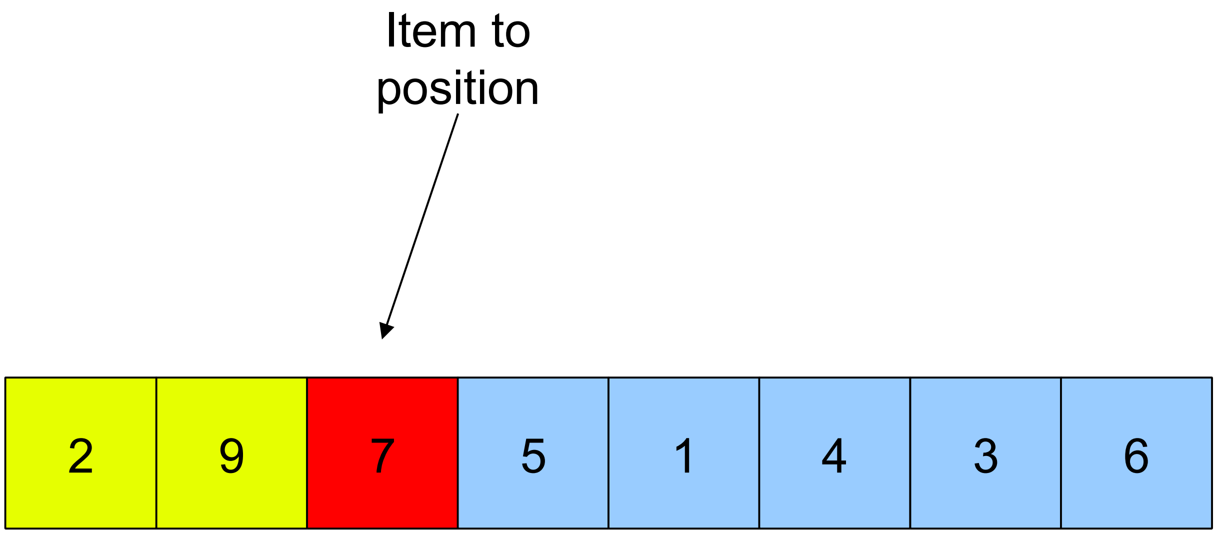
2	9	7	5	1	4	3	6

INSERTION SORT



2	9	7	5	1	4	3	6

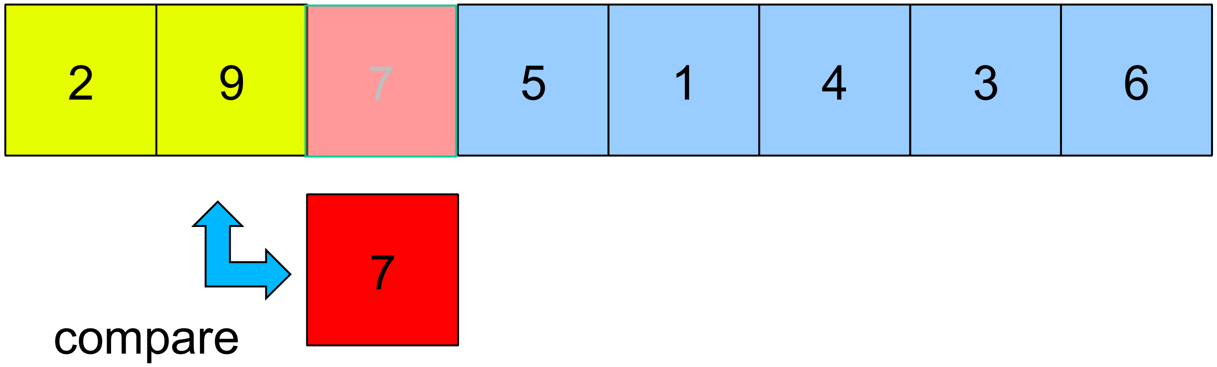
INSERTION SORT



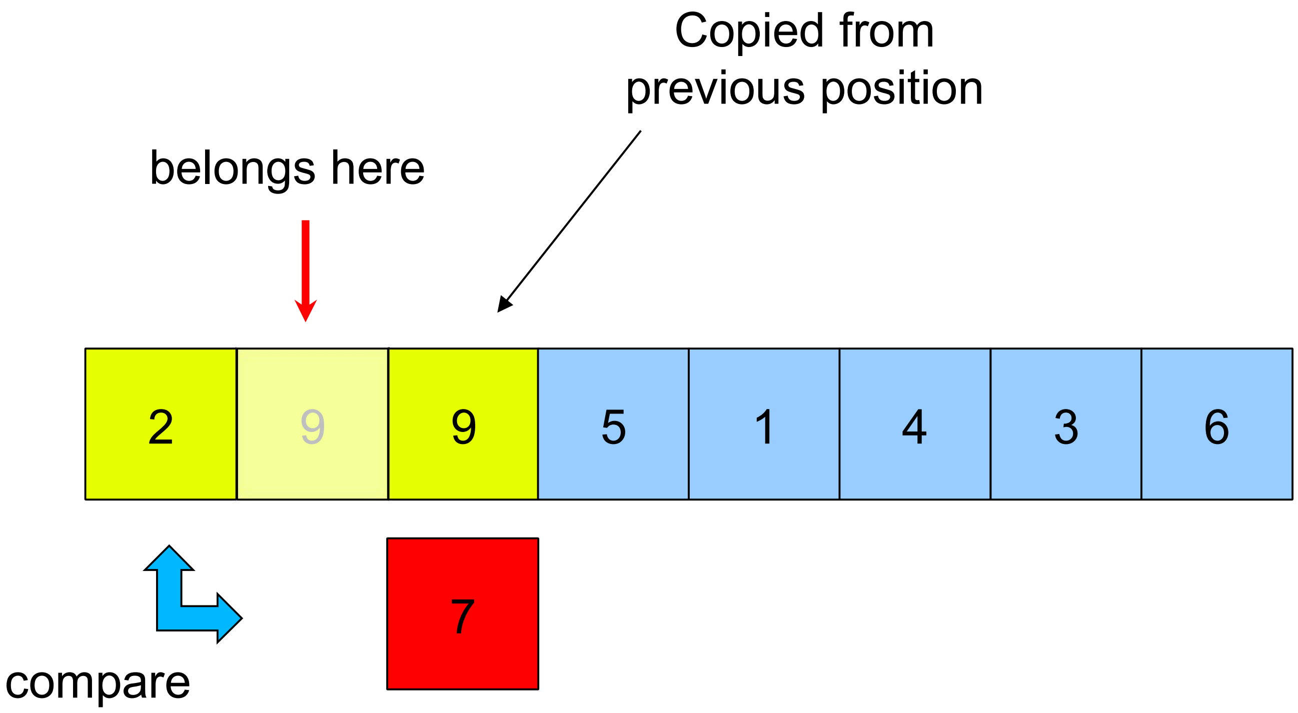
Item to position

2	9	7	5	1	4	3	6

INSERTION SORT



INSERTION SORT



Copied from
previous position

belongs here

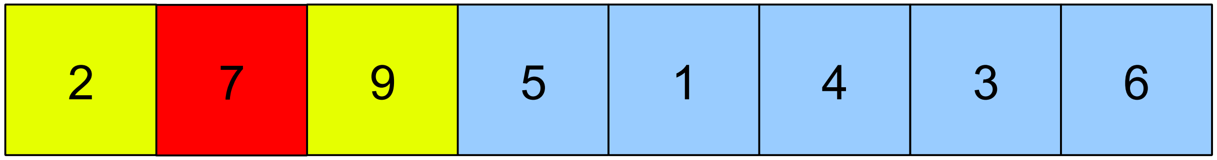
2	9	9	5	1	4	3	6

7

compare

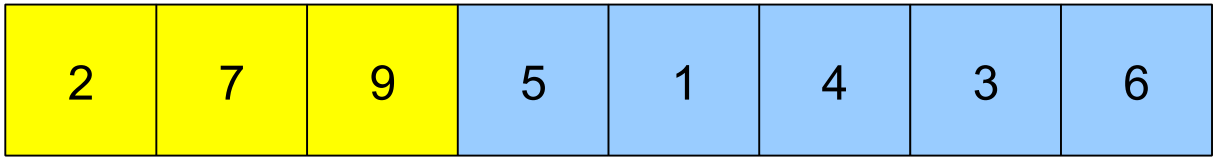
If the item in the sorted section is less than the item to place, the item to place goes *after* it in the array.

INSERTION SORT



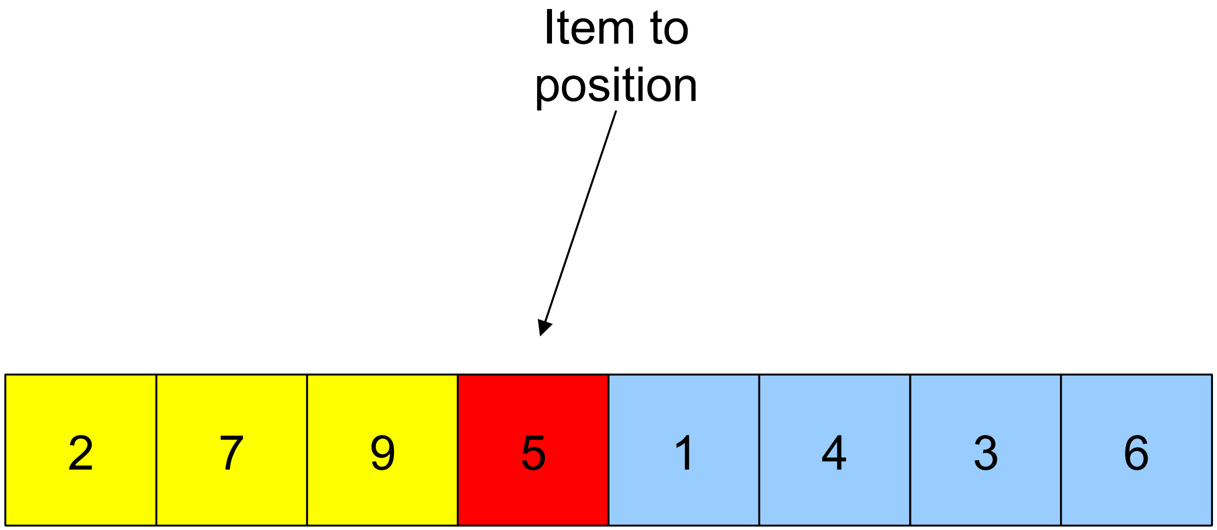
2	7	9	5	1	4	3	6

INSERTION SORT



2	7	9	5	1	4	3	6

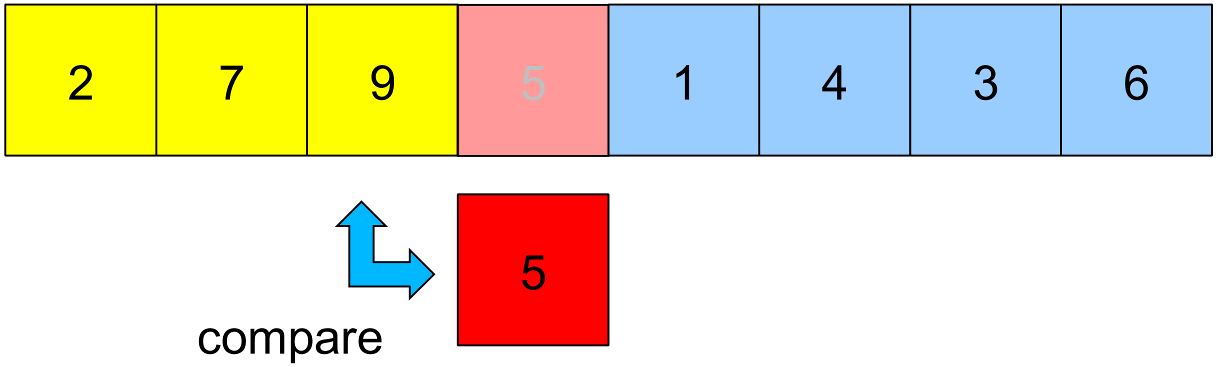
INSERTION SORT



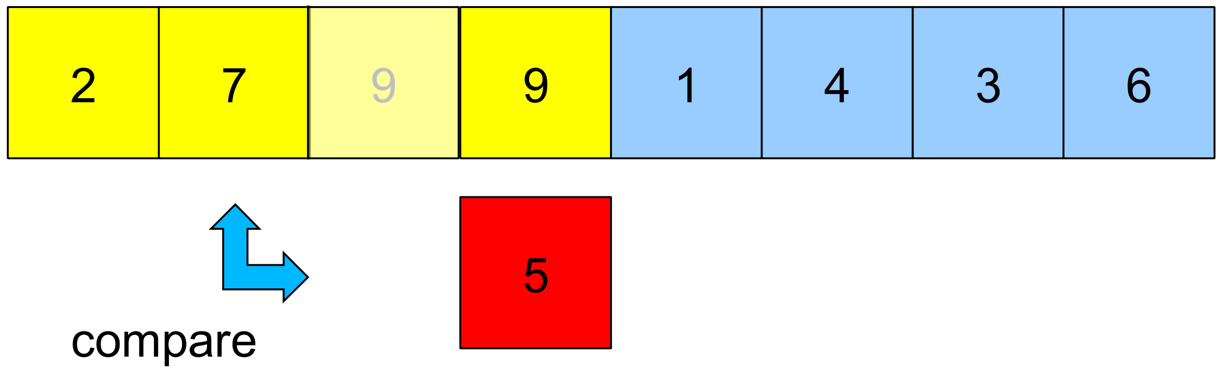
Item to position

2	7	9	5	1	4	3	6

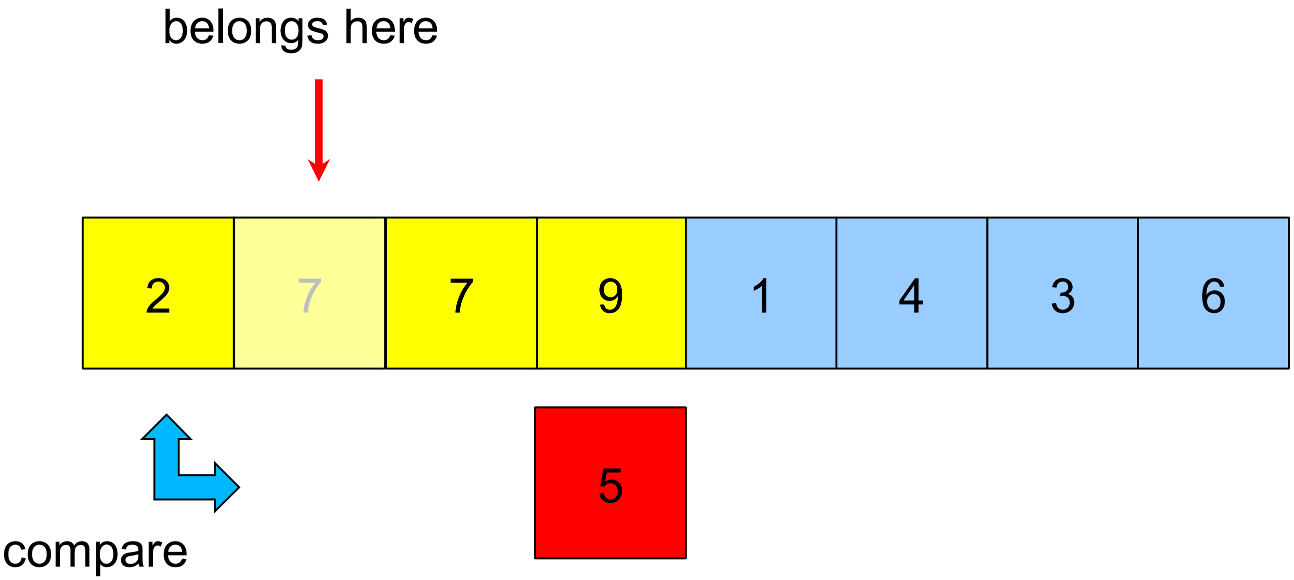
INSERTION SORT



INSERTION SORT



INSERTION SORT



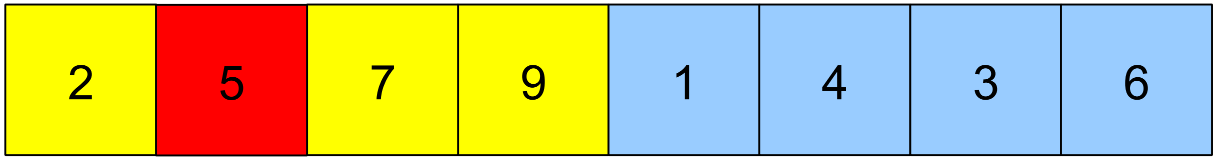
belongs here

2	7	7	9	1	4	3	6

5

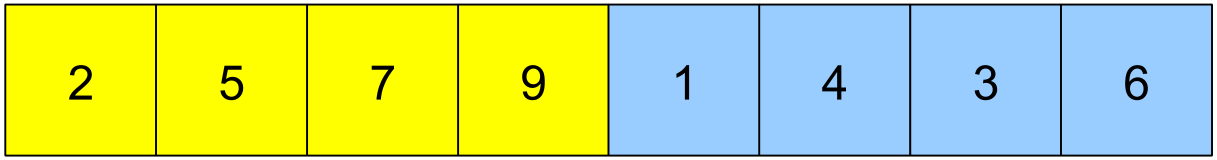
compare

INSERTION SORT



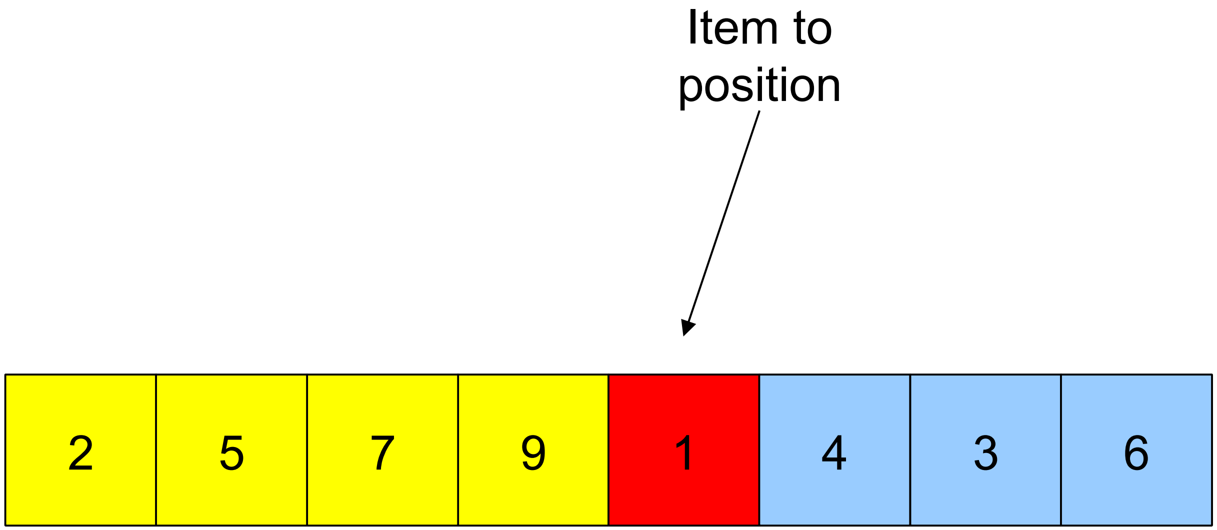
2	5	7	9	1	4	3	6

INSERTION SORT



2	5	7	9	1	4	3	6

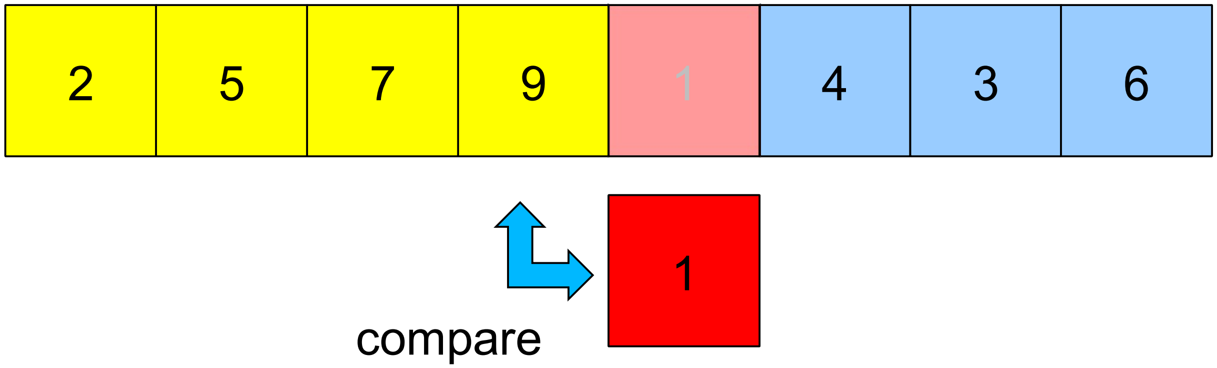
INSERTION SORT



Item to position

2	5	7	9	1	4	3	6

INSERTION SORT

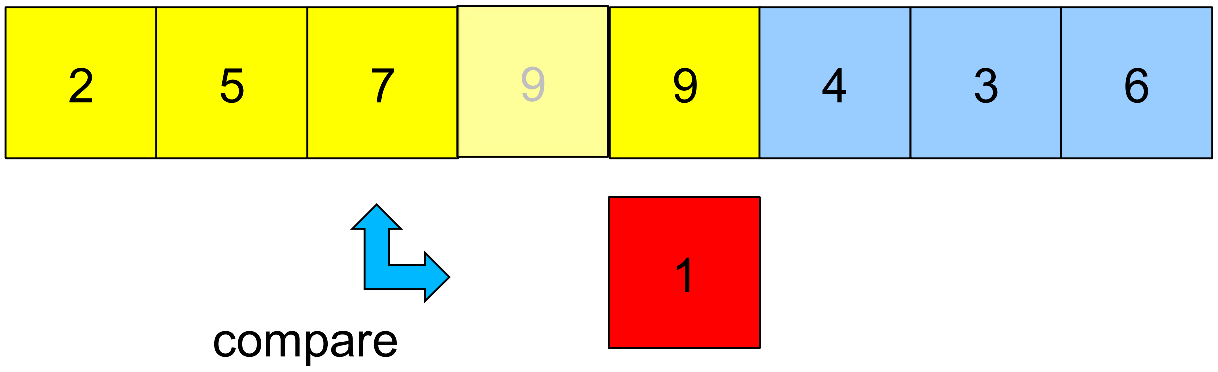


2	5	7	9	1	4	3	6

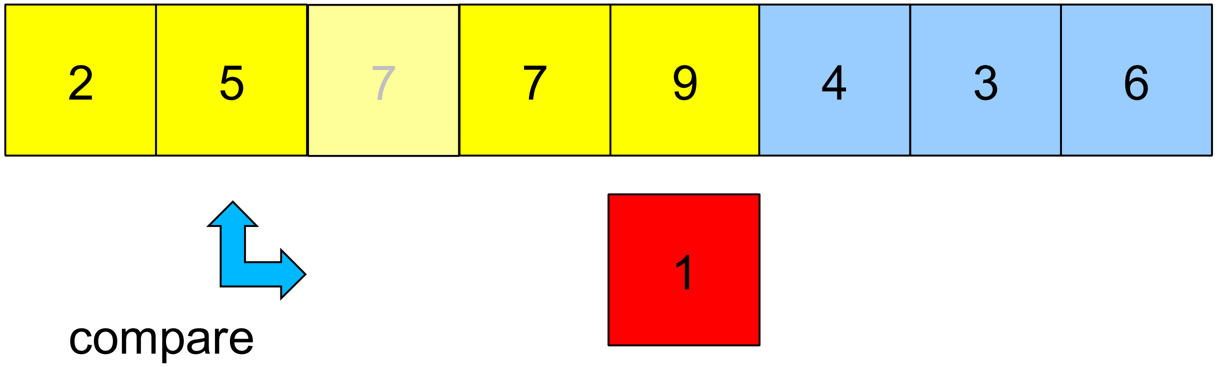
1

compare

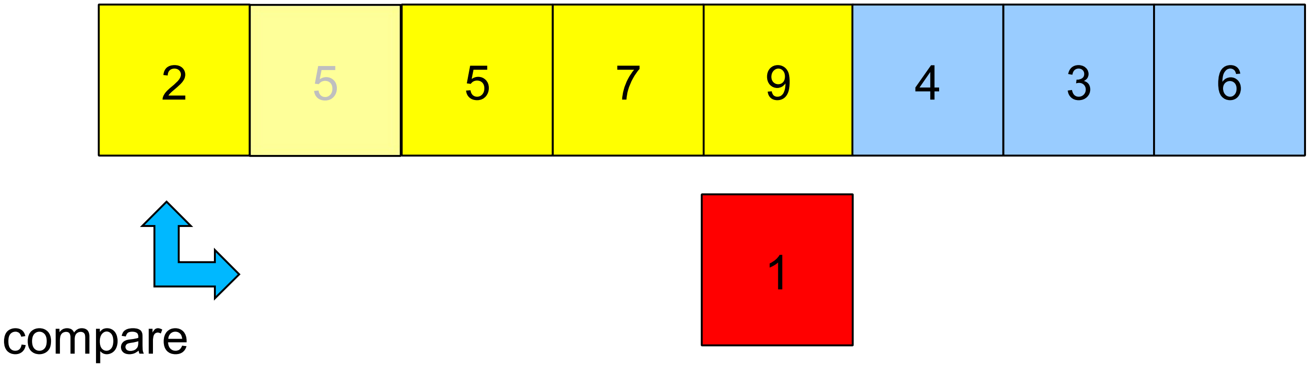
INSERTION SORT



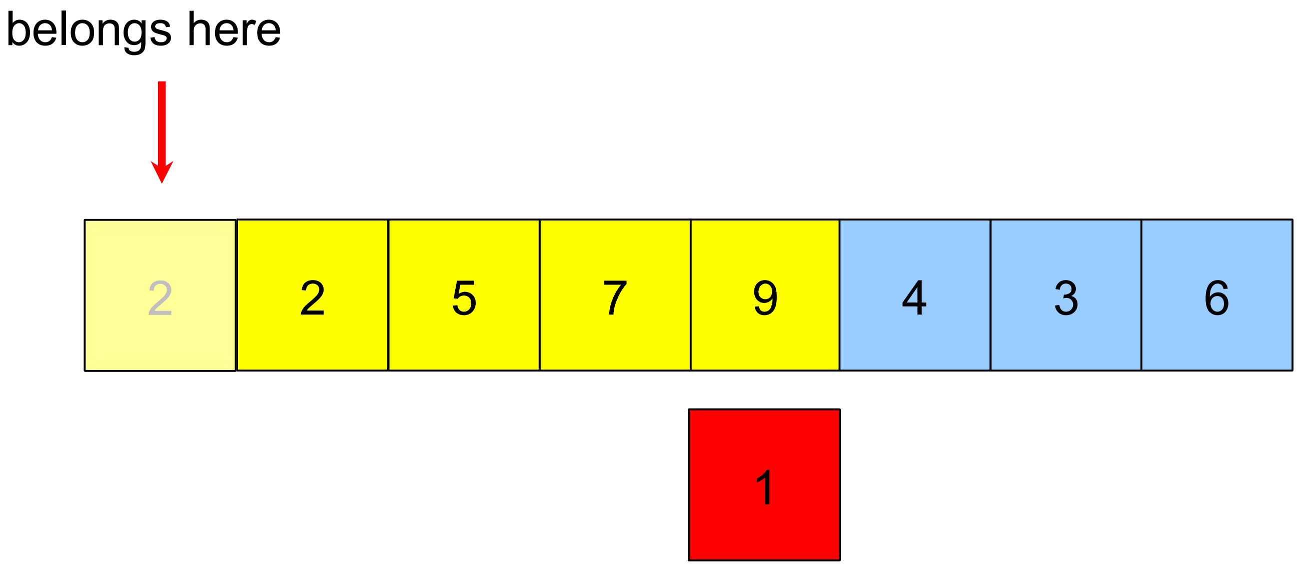
INSERTION SORT



INSERTION SORT



INSERTION SORT

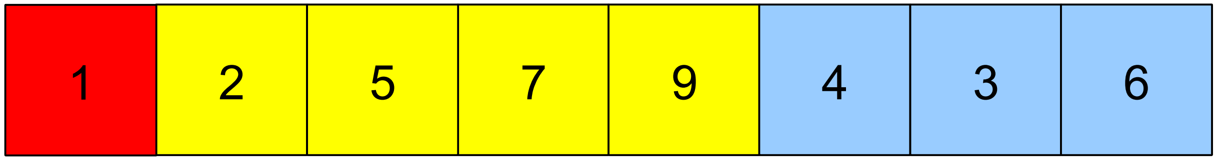


belongs here

2	2	5	7	9	4	3	6

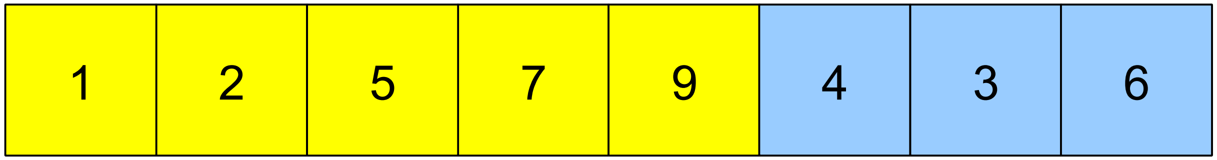
1

INSERTION SORT



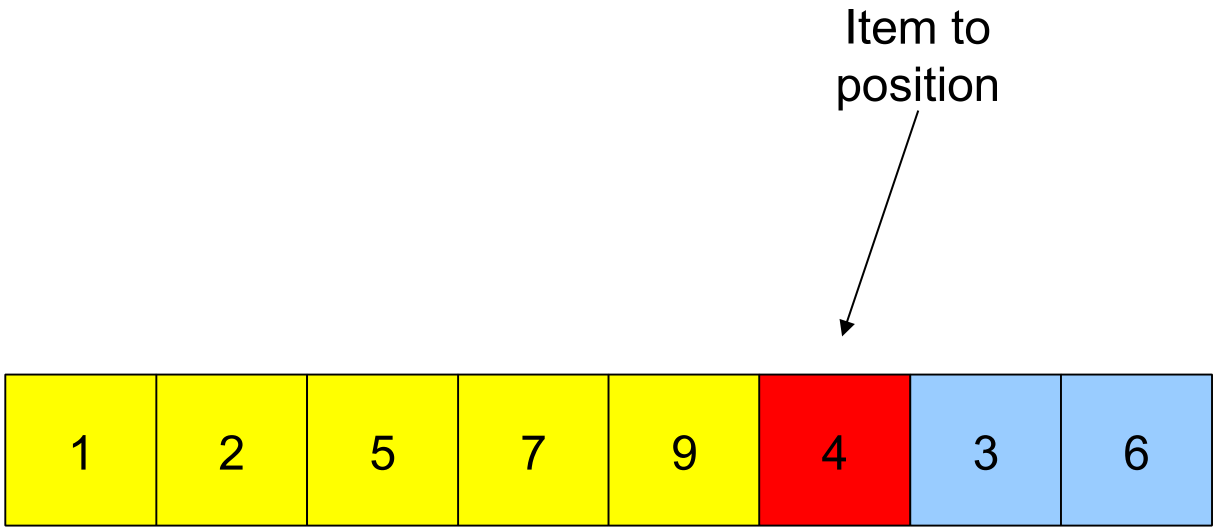
1	2	5	7	9	4	3	6

INSERTION SORT



1	2	5	7	9	4	3	6

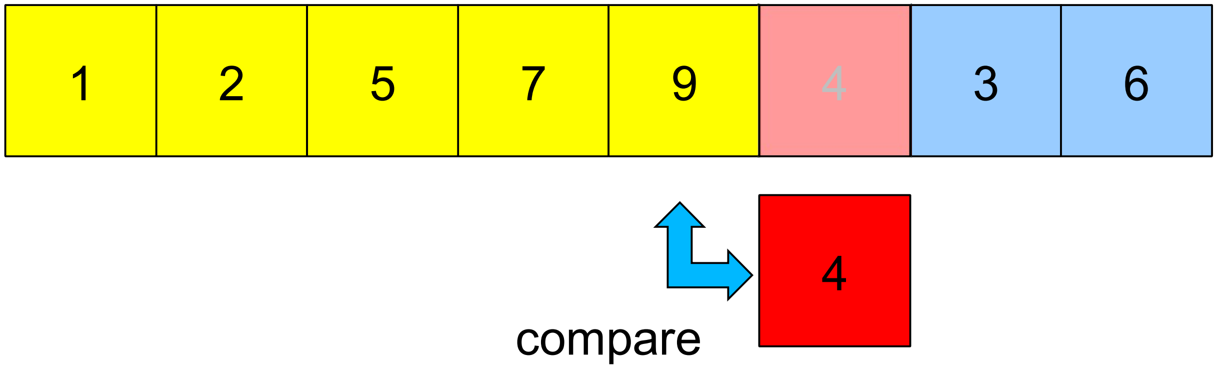
INSERTION SORT



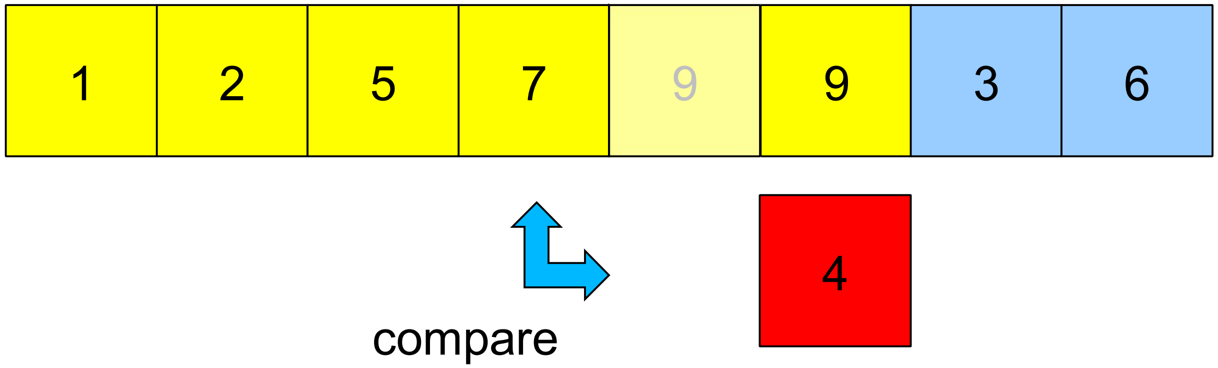
Item to position

1	2	5	7	9	4	3	6

INSERTION SORT

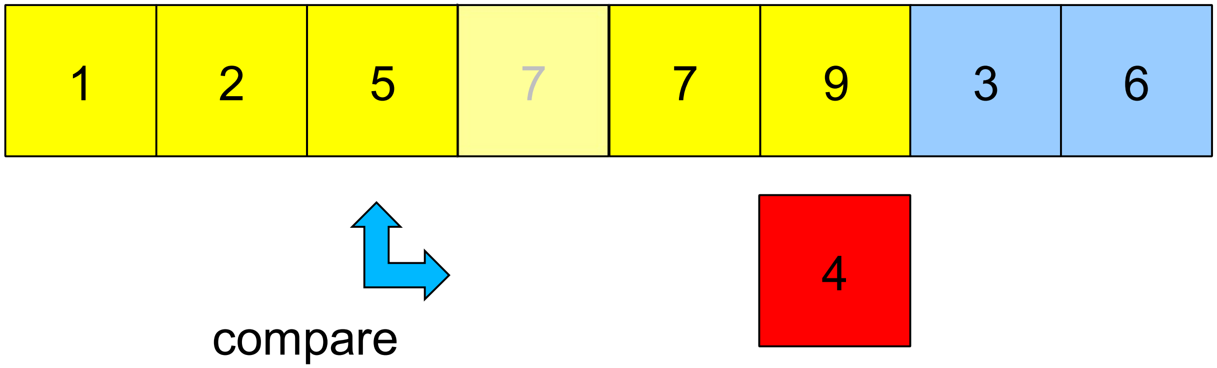


INSERTION SORT



1	2	5	7	9	9	3	6

INSERTION SORT

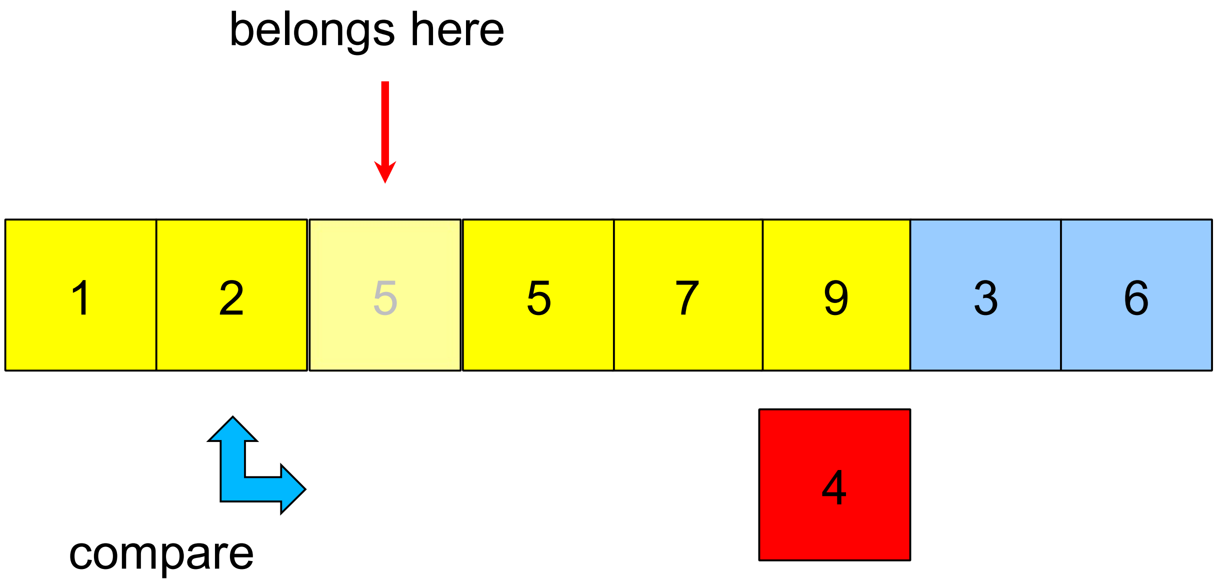


1	2	5	7	7	9	3	6

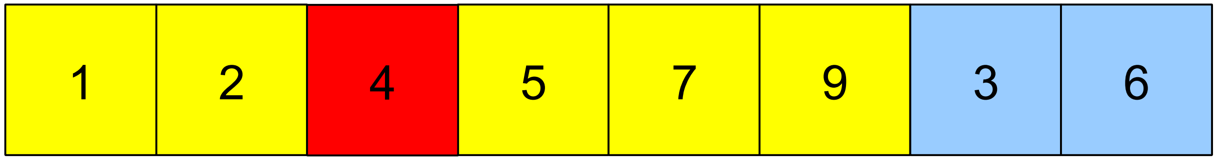
4

compare

INSERTION SORT

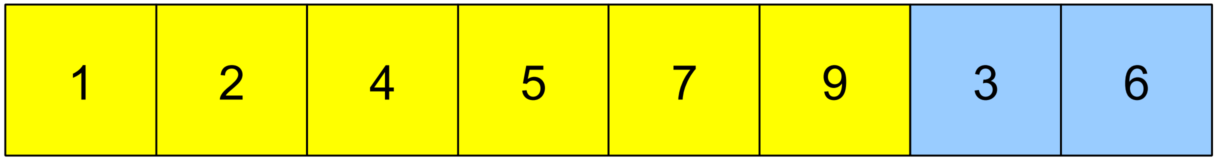


INSERTION SORT



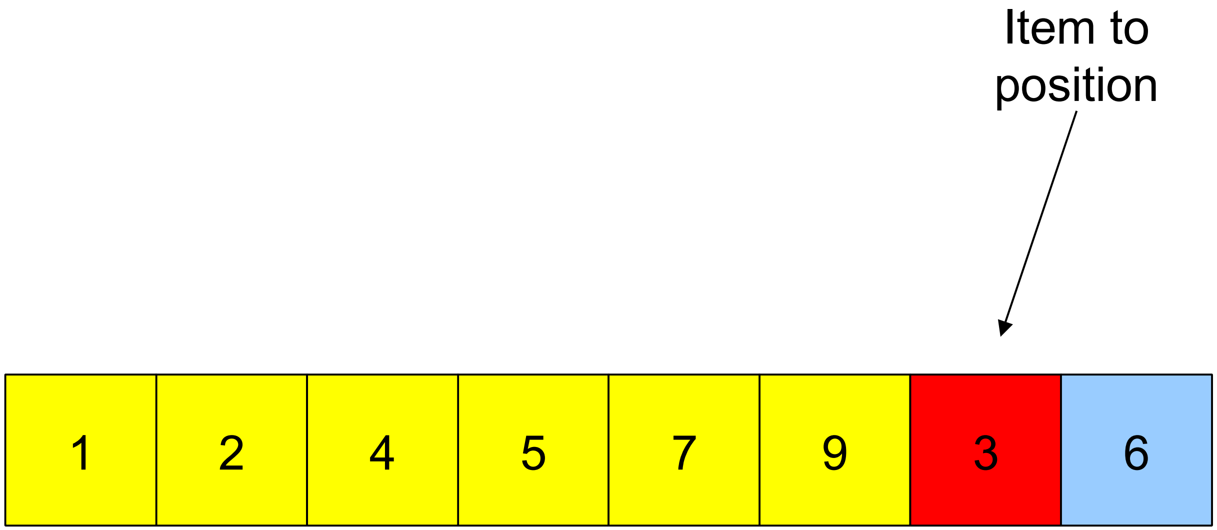
1	2	4	5	7	9	3	6

INSERTION SORT



1	2	4	5	7	9	3	6

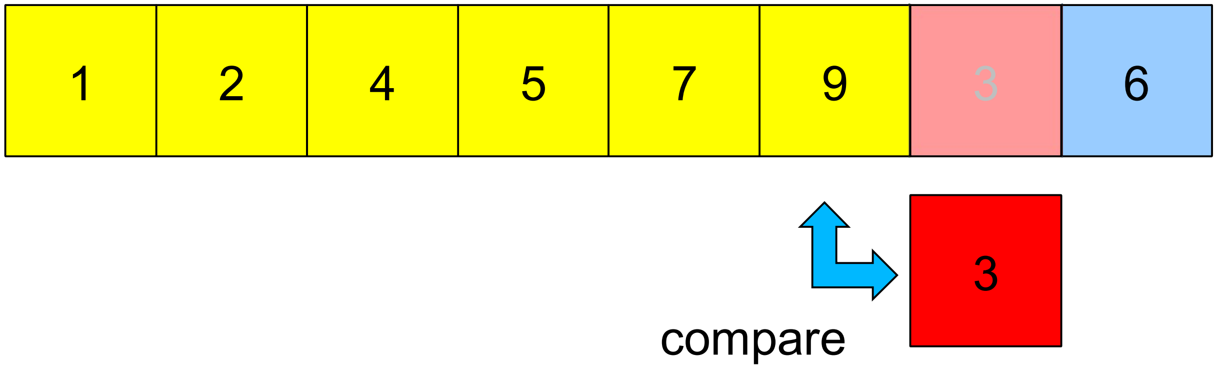
INSERTION SORT



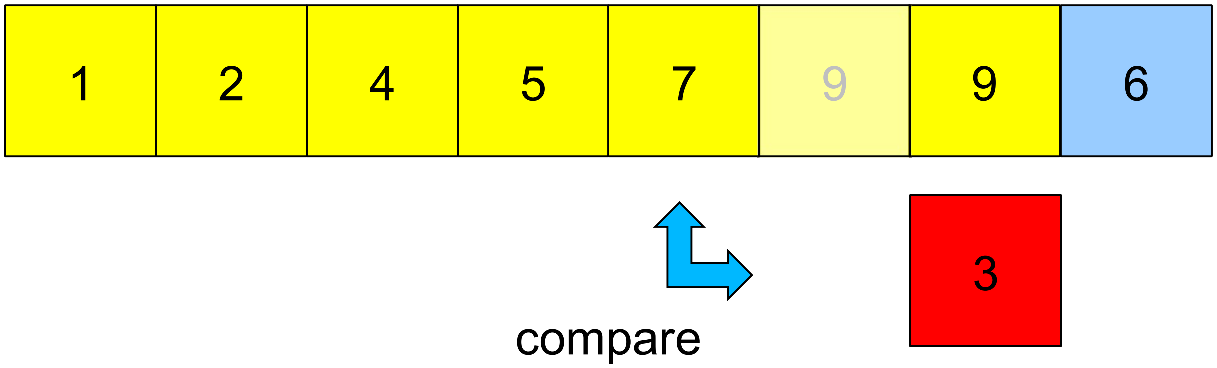
Item to position

1	2	4	5	7	9	3	6

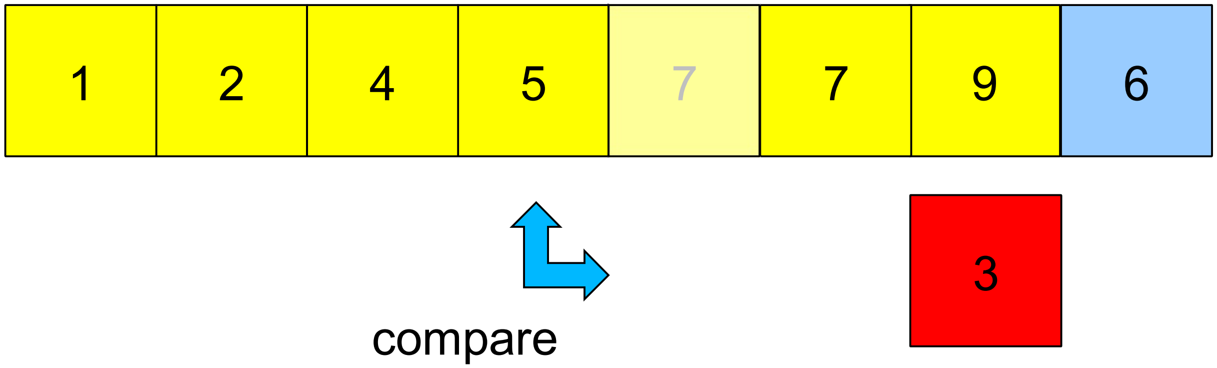
INSERTION SORT



INSERTION SORT



INSERTION SORT

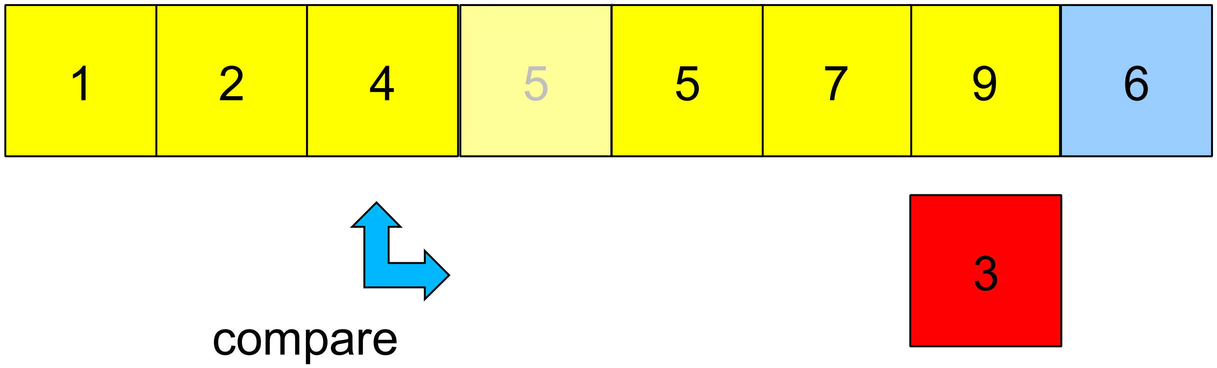


1	2	4	5	7	7	9	6

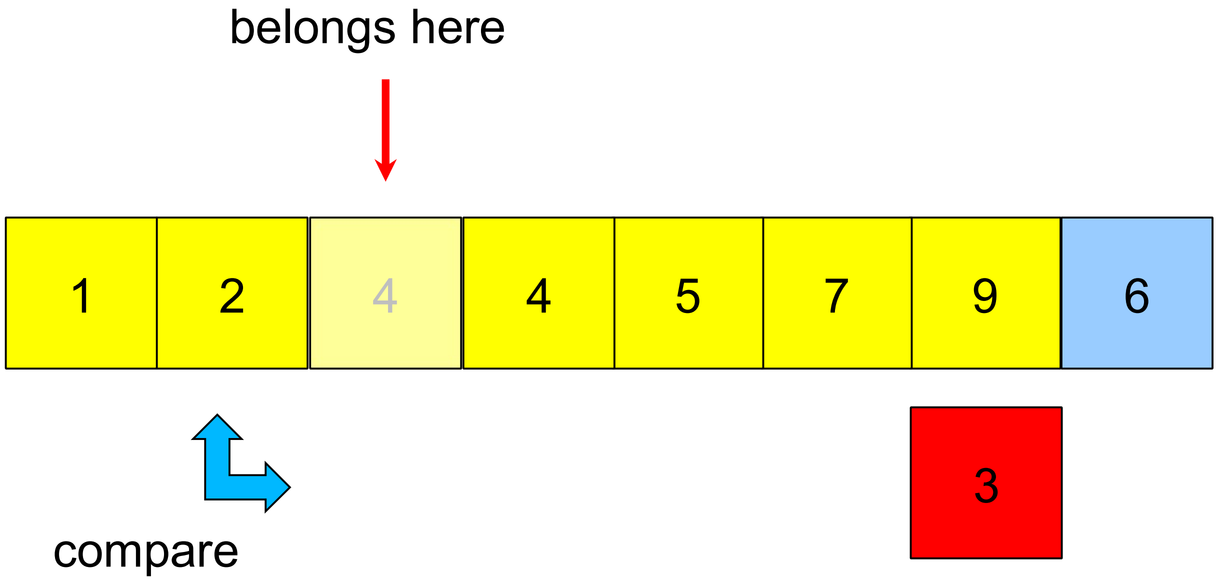
3

compare

INSERTION SORT



INSERTION SORT



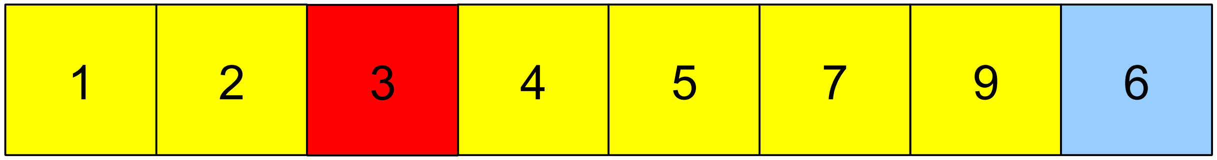
belongs here

1	2	4	4	5	7	9	6

3

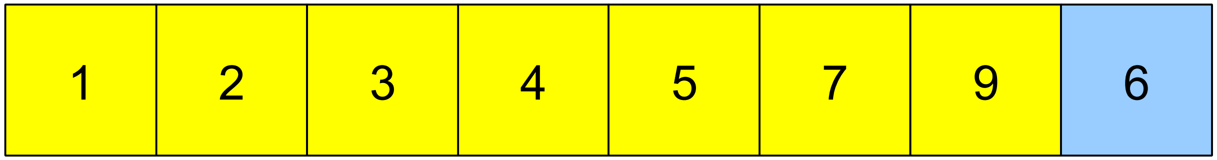
compare

INSERTION SORT



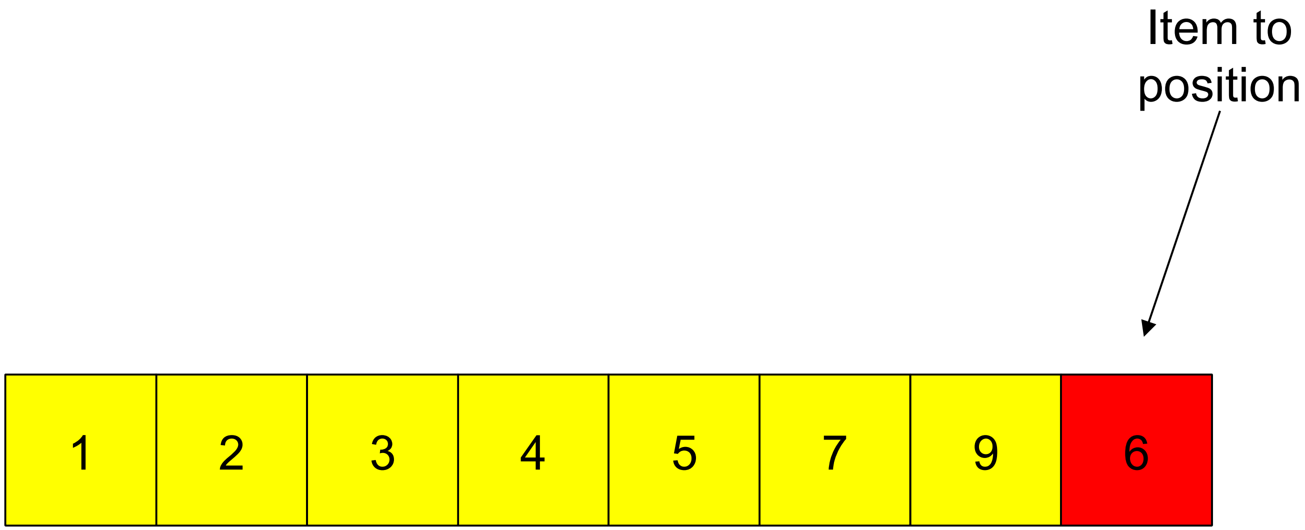
1	2	3	4	5	7	9	6

INSERTION SORT



1	2	3	4	5	7	9	6

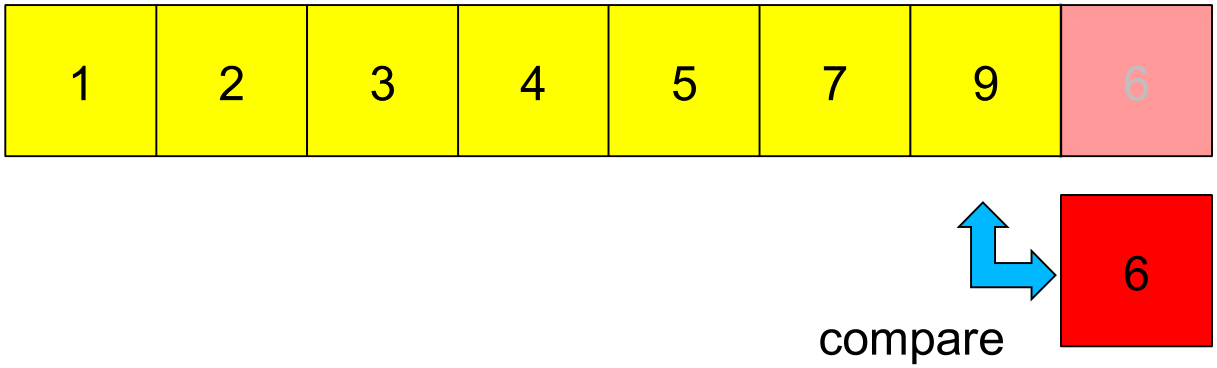
INSERTION SORT



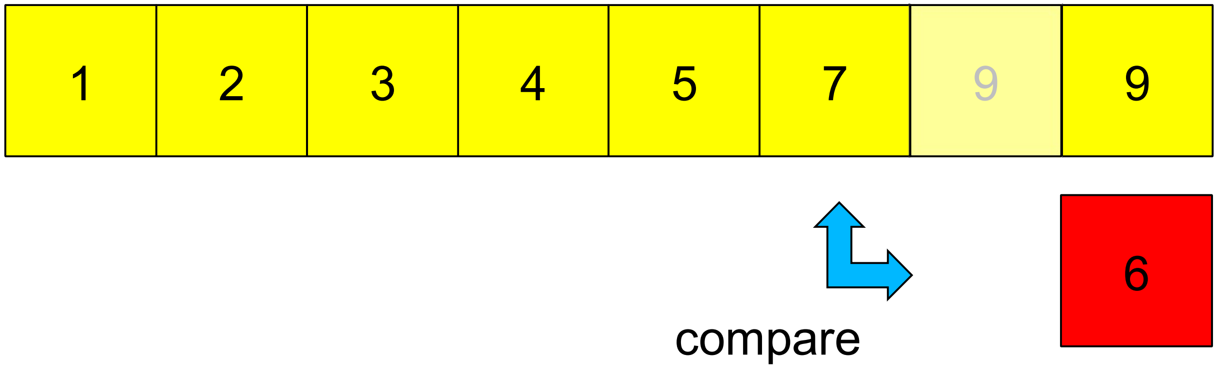
Item to position

1	2	3	4	5	7	9	6

INSERTION SORT



INSERTION SORT

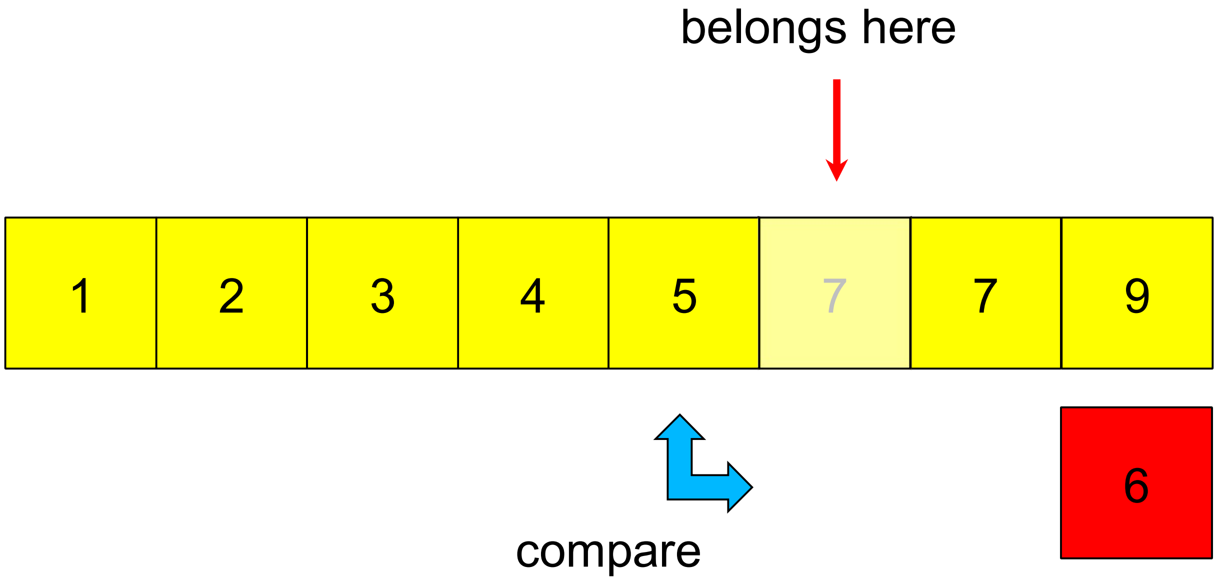


1	2	3	4	5	7	9	9

6

compare

INSERTION SORT



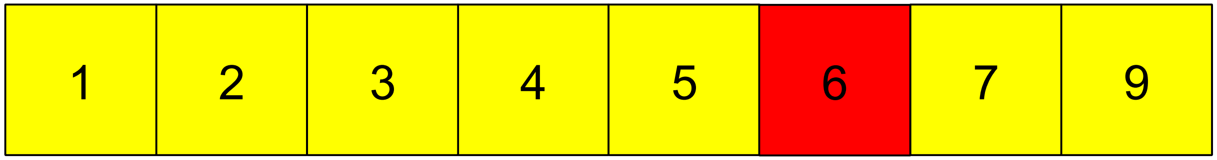
belongs here

1	2	3	4	5	7	7	9

6

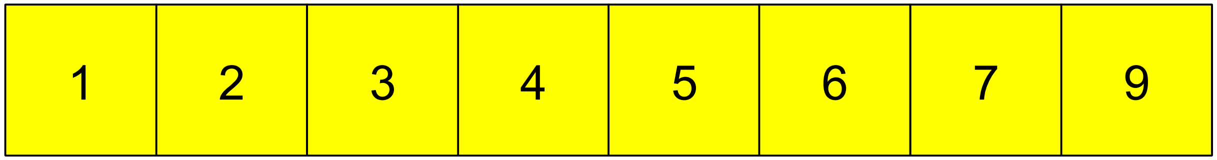
compare

INSERTION SORT



1	2	3	4	5	6	7	9

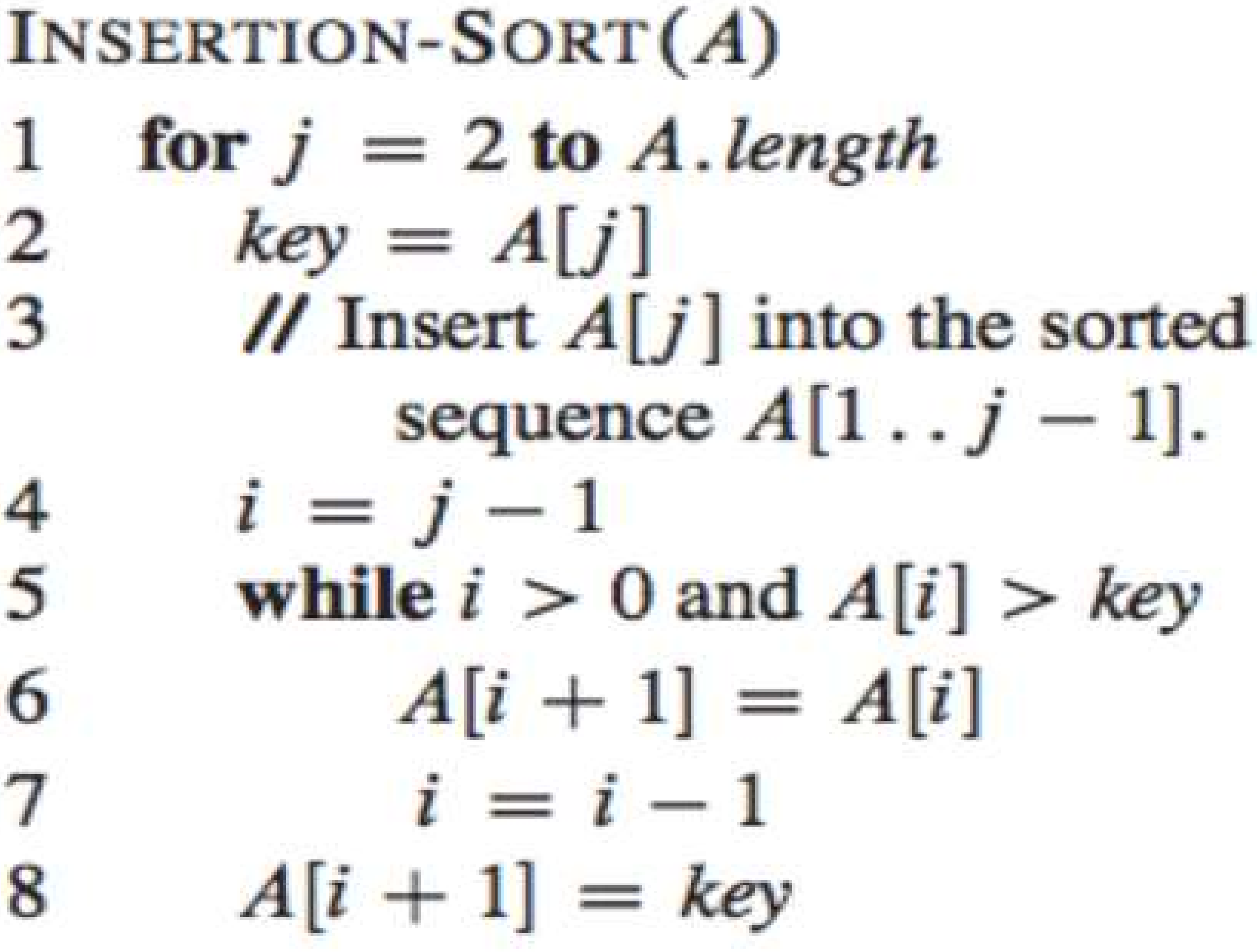
INSERTION SORT



1	2	3	4	5	6	7	9

SORTED!

INSERTION SORT ALGORITHM



```
INSERTION-SORT($A$)
for $j = 2$ to $A.length$
    $key = A[j]$
    // Insert $A[j]$ into the sorted
          sequence $A[1..j-1]$.
    $i = j - 1$
    while $i > 0$ and $A[i] > key$
        $A[i+1] = A[i]$
        $i = i - 1$
    $A[i+1] = key$
```