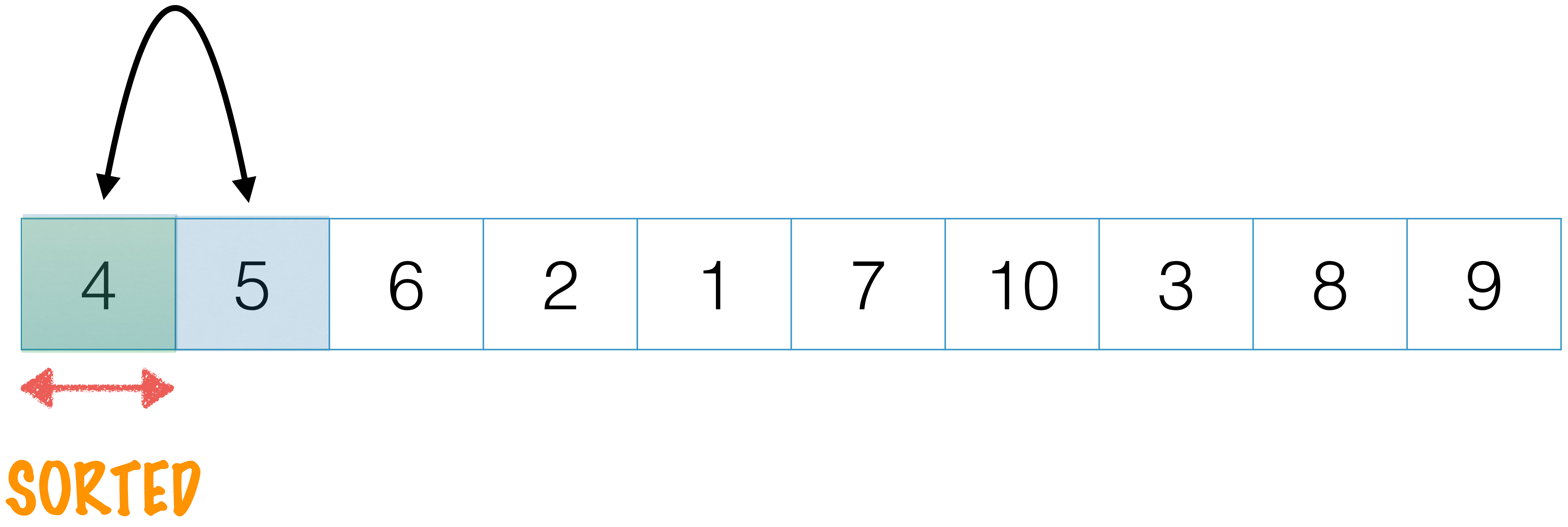
INSERTION SORT

START WITH A SORTED SUB-LIST OF SIZE 1

INSERT THE NEXT ELEMENT INTO THE SORTED SUB-LIST AT THE RIGHT POSITION. NOW THE SORTED SUB-LIST HAS 2 ELEMENTS

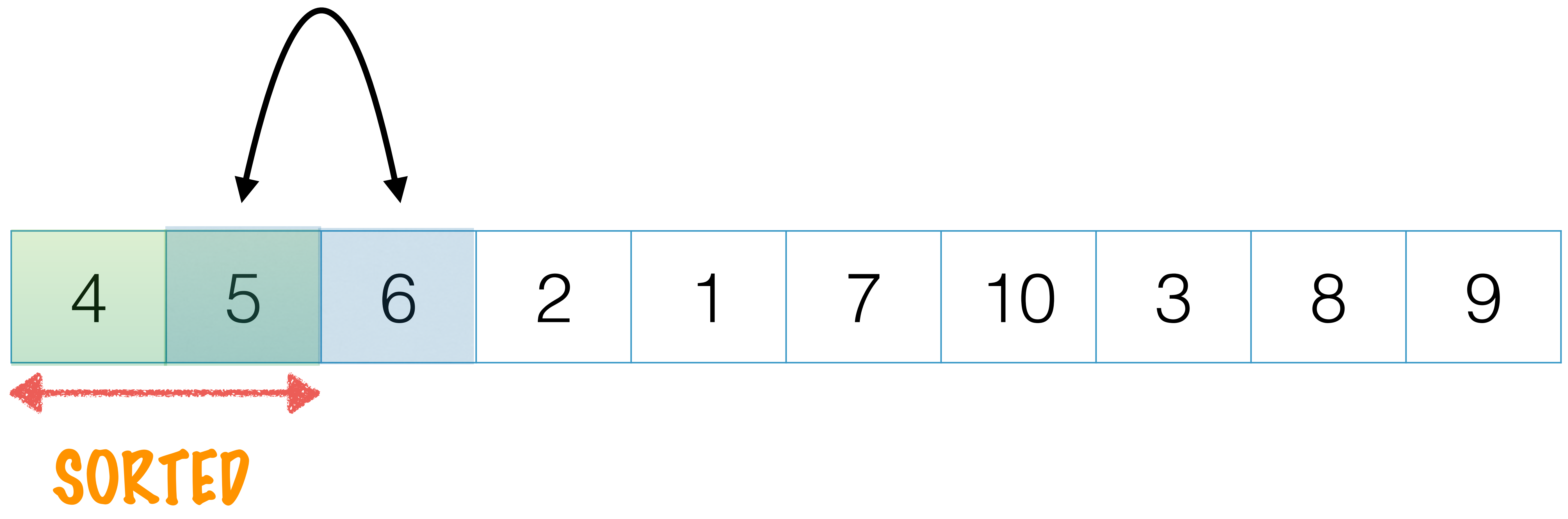
THIS CONTINUES TILL THE ENTIRE LIST IS SORTED.

INSERTION SORT

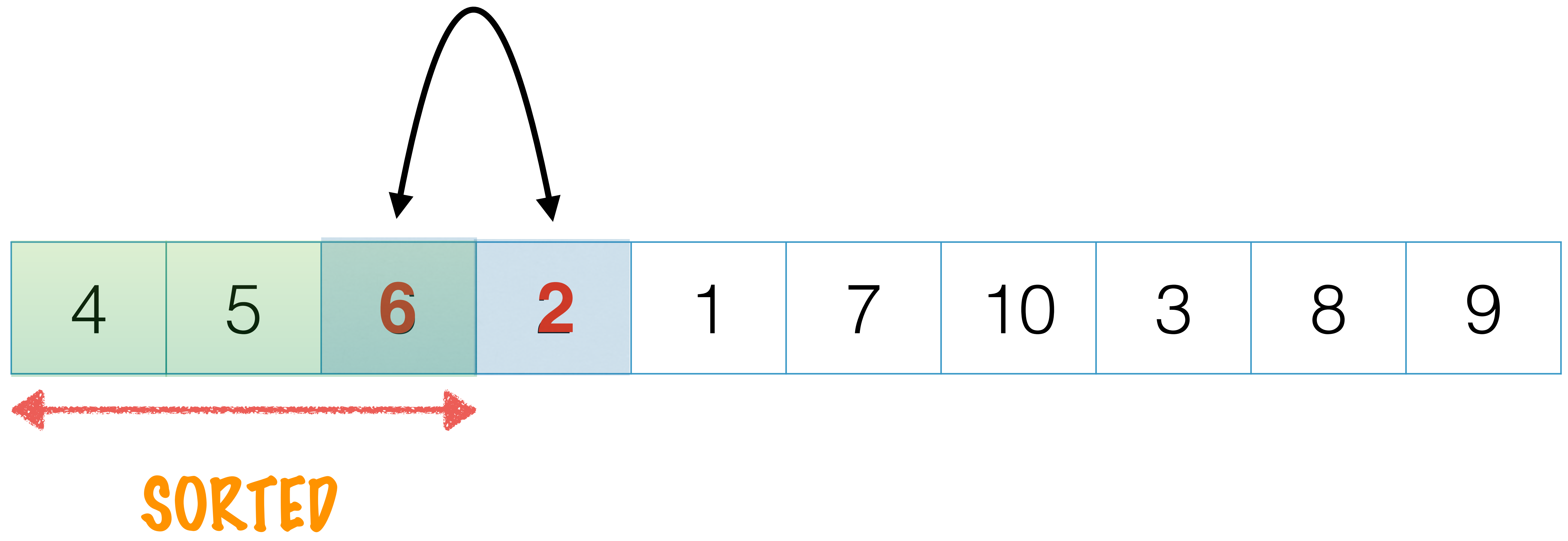


THE SORTED LIST STARTS WITH 1 ELEMENT, A LIST OF ONE ELEMENT IS ALWAYS SORTED

INSERTION SORT



INSERTION SORT



THE SIZE OF THE SORTED LIST IS SLOWLY INCREASING, IT NOW HAS 3 ELEMENTS

INSERTION SORT

4	5	**2**	**6**	1	7	10	3	8	9

INSERTION SORT

4	**5**	**2**	6	1	7	10	3	8	9

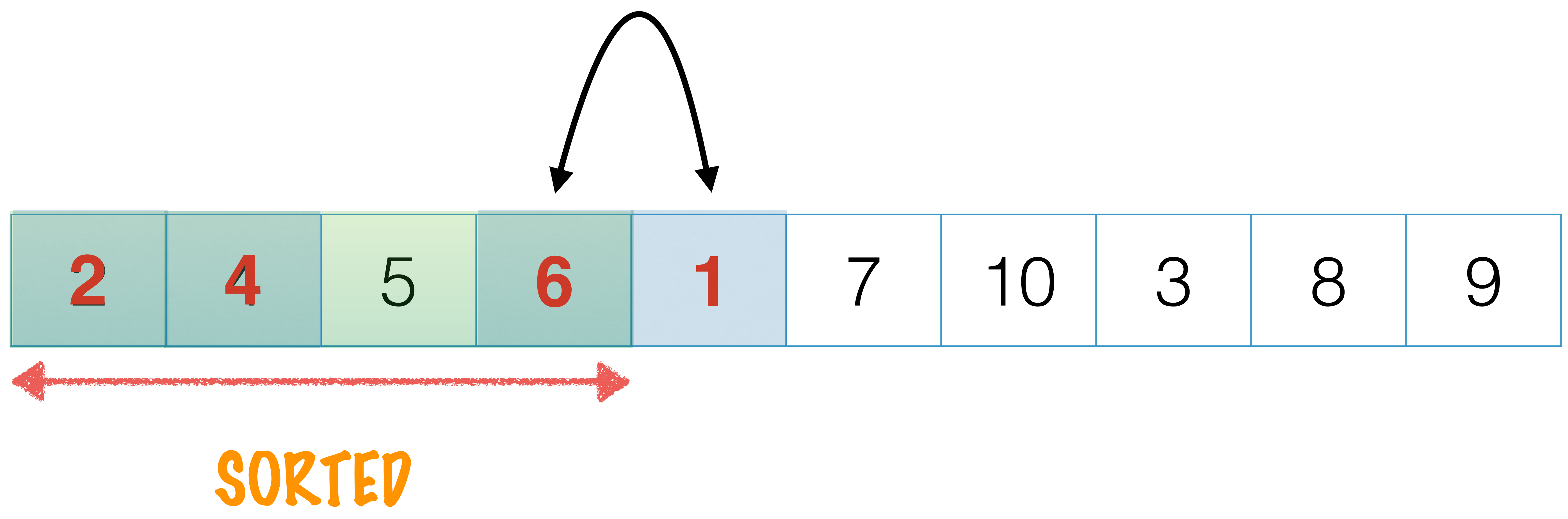
INSERTION SORT

4	2	5	6	1	7	10	3	8	9

INSERTION SORT

4	2	5	6	1	7	10	3	8	9

INSERTION SORT



INSERTION SORT

2	4	5	**1**	**6**	7	10	3	8	9

INSERTION SORT

2	4	5	1	6	7	10	3	8	9

INSERTION SORT

2	4	**1**	**5**	6	7	10	3	8	9

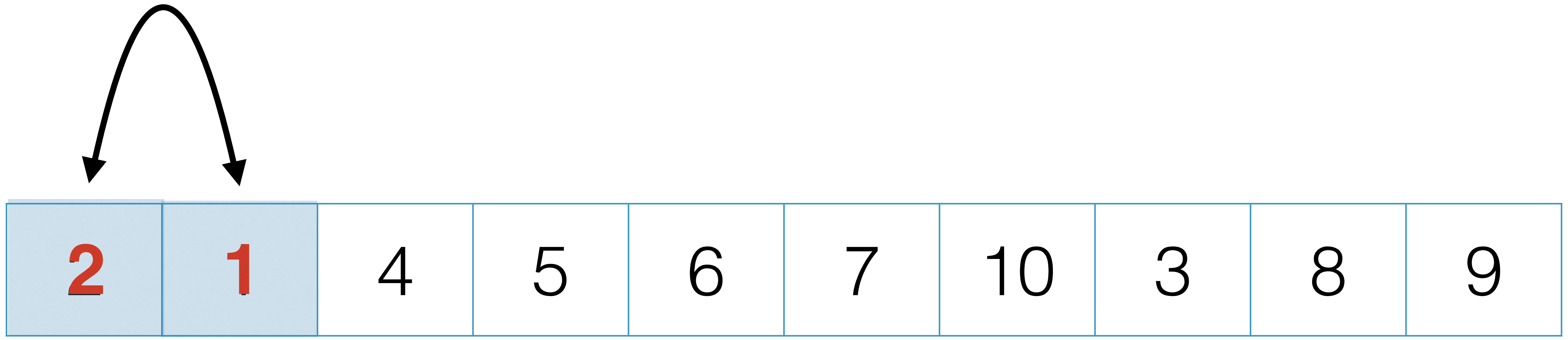
INSERTION SORT

2	**4**	**1**	5	6	7	10	3	8	9

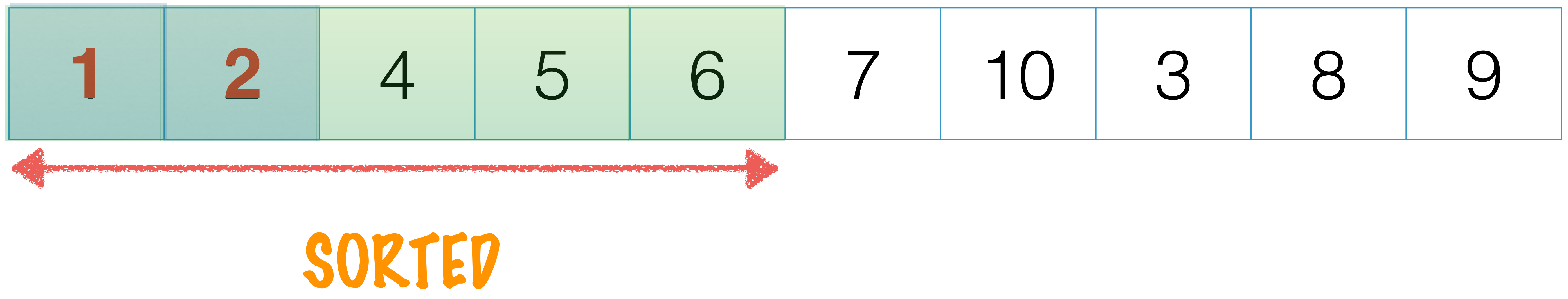
INSERTION SORT

2	1	4	5	6	7	10	3	8	9

INSERTION SORT



INSERTION SORT

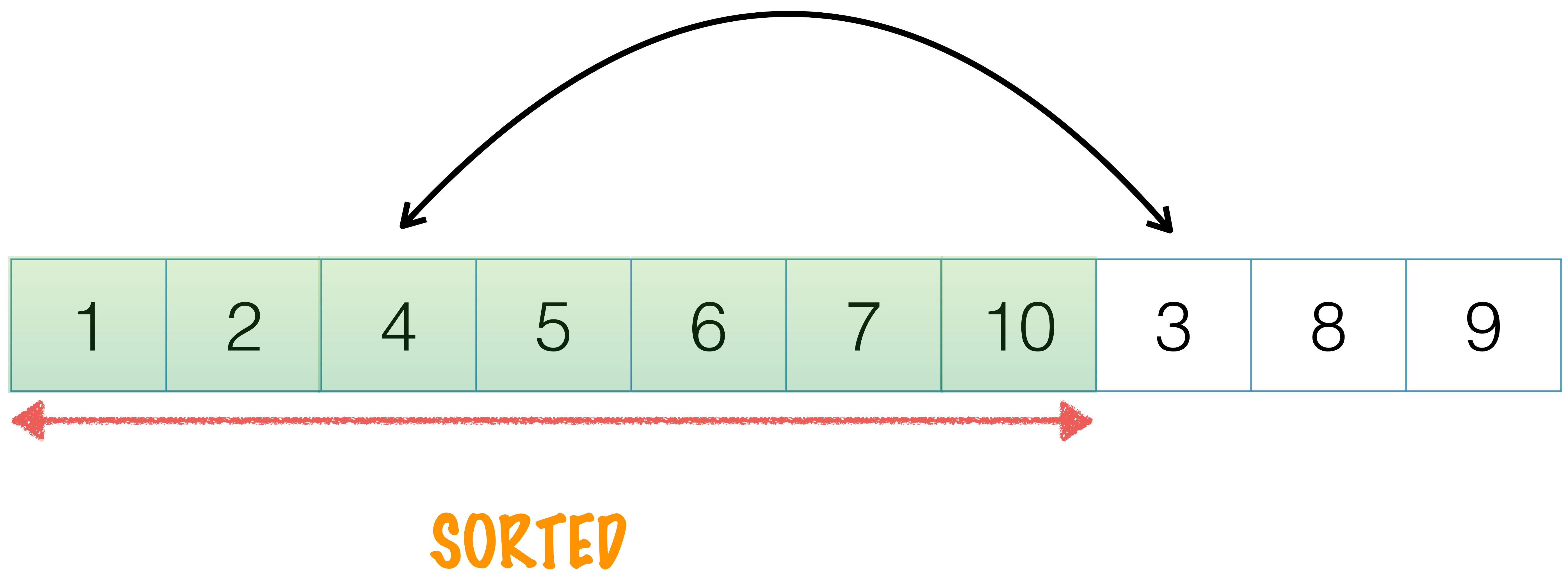


INSERTION SORT

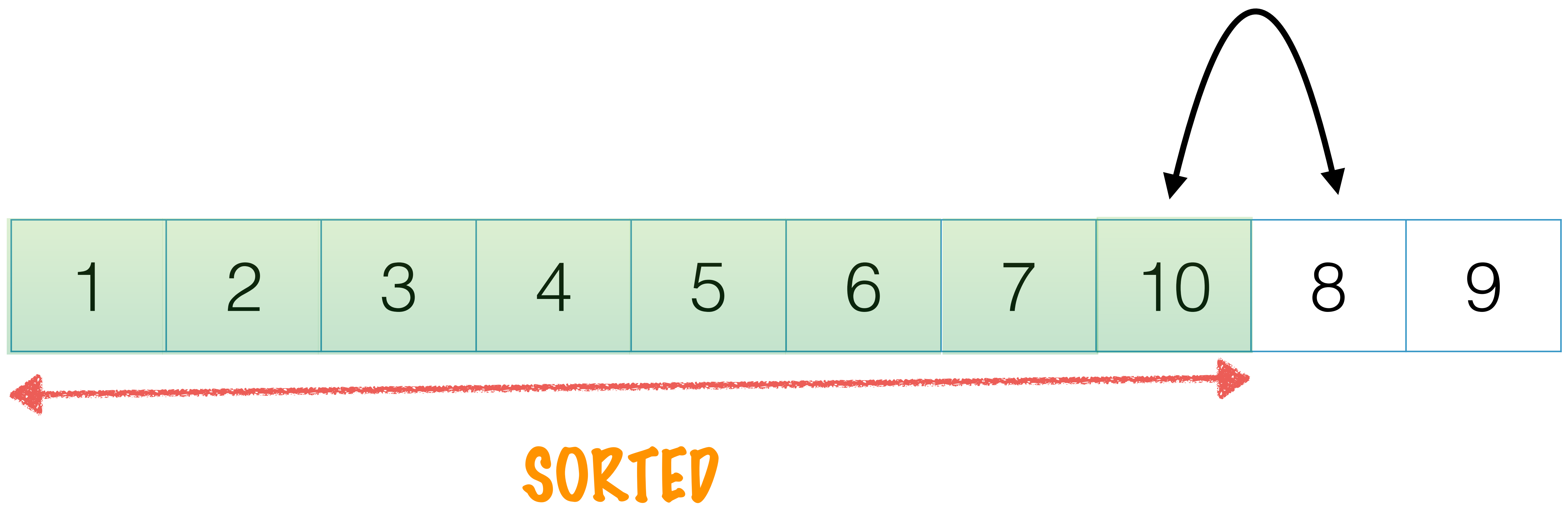
1	2	4	5	6	7	10	3	8	9

SORTED

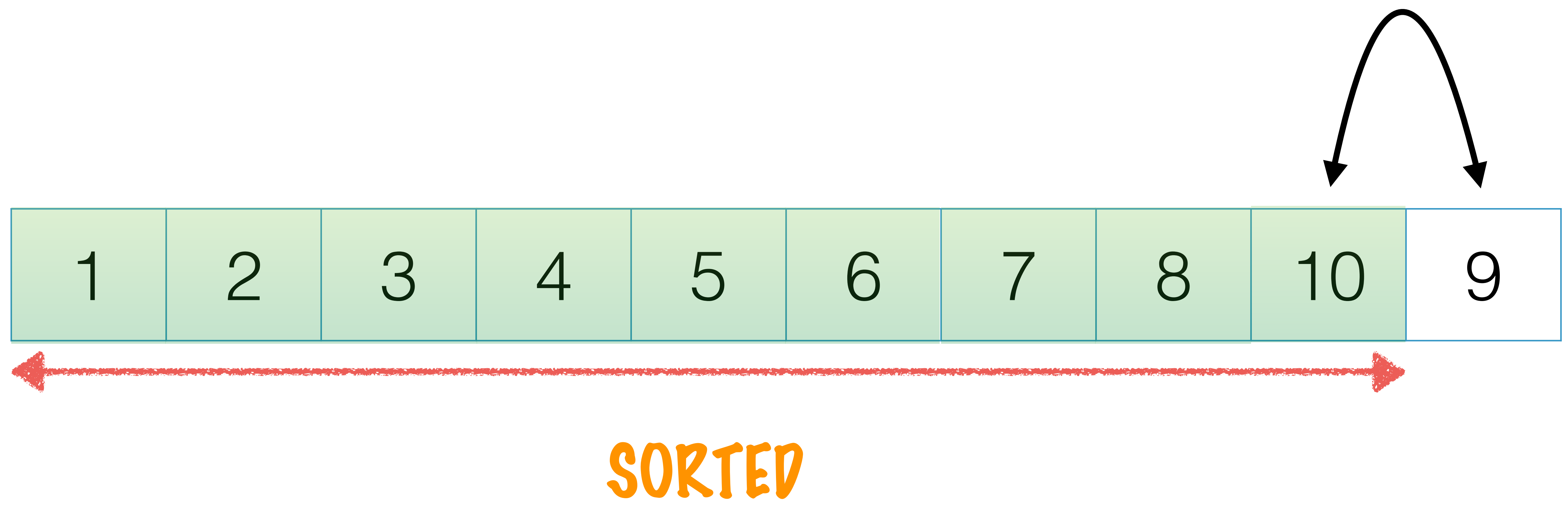
INSERTION SORT



INSERTION SORT



INSERTION SORT



INSERTION SORT

THE LIST IS NOW FULLY SORTED!

BY INSERTING INTO A SORTED SUB-LIST AT EVERY STEP THE SUB-LIST SOON GROWS TO BE THE ENTIRE LIST

INSERTION SORT - CODE

GO UP TO THE SECOND TO LAST ELEMENT

```
public static void insertionSort(int[] listToSort) {
    for (int i = 0; i < listToSort.length - 1; i++) {
        for (int j = i + 1; j > 0; j--) {
            if (listToSort[j] < listToSort[j - 1]) {
                swap(listToSort, j, j - 1);
            } else {
                break;
            }
            print(listToSort);
        }
    }
}
```

BUBBLE THE ELEMENT OUTSIDE THE SORTED SUB-LIST TO THE RIGHT POSITION

CONSIDER EVERYTHING UPTIL THE `ith` ELEMENT SORTED

IF NO SWAP WAS PERFORMED THE ELEMENT HAS BEEN MOVED TO THE RIGHT POSITION SO BREAK OUT OF THE LOOP

THIS SORT FIRST ASSUMES A SORTED LIST OF SIZE 1 AND INSERTS ADDITIONAL ELEMENTS IN THE RIGHT POSITION

IN THE WORST CASE (IF THE LIST IS ORIGINALLY SORTED IN DESCENDING ORDER) "N" ELEMENTS ARE CHECKED AND SWAPPED FOR EVERY SELECTED ELEMENT TO GET TO THE RIGHT POSITION

CHECKING "N" ELEMENTS FOR EACH OF "N" SELECTED ELEMENTS

THE COMPLEXITY OF INSERTION SORT IS $O(N^2)$

IT IS A STABLE SORT - AS ENTITIES BUBBLE TO THE CORRECT POSITION IN THE SUBLIST THAT IS SORTED. THE LIST THE ORIGINAL ORDER OF ENTITIES ARE MAINTAINED FOR EQUAL ELEMENTS

IT TAKES $O(1)$ EXTRA SPACE, IT SORTS IN PLACE

IT MAKES $O(N^2)$ COMPARISONS AND $O(N^2)$ SWAPS

THIS IS SIMILAR TO BUBBLE SORT, IT IS ADAPTIVE IN THAT NEARLY SORTED LISTS COMPLETE VERY QUICKLY

IT HAS VERY LOW OVERHEAD AND IS TRADITIONALLY THE SORT OF CHOICE WHEN USED WITH FASTER ALGORITHMS WHICH FOLLOW THE DIVIDE AND CONQUER APPROACH

INSERTION SORT VS BUBBLE SORT

1. BUBBLE SORT REQUIRES AN ADDITIONAL PASS OVER ALL ELEMENTS TO ENSURE THAT THE LIST IS FULLY SORTED

2. BUBBLE SORT HAS TO DO N COMPARISONS AT EVERY STEP. INSERTION SORT CAN STOP COMPARISON ELEMENTS WHEN THE RIGHT POSITION IN THE SORTED LIST IS FOUND

3. BUBBLE SORT PERFORMS POORLY WITH MODERN HARDWARE BECAUSE OF THE NUMBER OF WRITES AND SWAPS THAT IT PERFORMS. RESULTS IN CACHE MISSES SO HAS GREATER OVERHEAD THAN INSERTION SORT