Sorting Algorithms

(Algorithms and Data Structures)

Sorting Algorithms

- a sorting algorithm is an algorithm that puts elements of an array (or list) in a certain order
- when sorting numerical data it is called the **numerical ordering**
- if we are after the sorted order of string sor characters – it is called **alphabetical ordering**

Sorting Algorithms

12	4	-2	1	20	0	8	3

Sorting Algorithms

-2	0	1	3	4	8	12	20

Sorting Algorithms

comparison based *sorting algorithms*
(bubble sort, merge sort
or quicksort)

if nums[i] < nums[j]:
swap items

non-comparison based
sorting algorithms
(bucket sort or radix sort)

Sorting Algorithms

For sorting **N** items: we have to make $\mathbf{log_2\ N!}$ comparisons

With *Stirling-formula* it can be reduced to **NlogN**

- so the **Ω(N logN)** time complexity is the lower bound for **comparison based** sorting algorithms
- ok but we can achieve **O(N)** running time as far as sorting is concerned such as bucket sort or radix sort

THESE ARE NOT COMPARISON BASED ALGORITHMS !!!

Sorting Algorithms

We can classify comparison based sorting algorithms based on their running time (how fast they are)

$O(N^2)$ quadratic running time sorting algorithms

(*bubble sort, insertion sort and selection sort*)

$O(N\log N)$ linearithmic running time sorting algorithms

(*merge sort* and *quicksort*)

$O(N)$ linear running time sorting algorithms

(*bucket sort* and *radix sort*)

Sorting Algorithms

1.) **IN-PLACE**: a sorting algorithm is called *in-place* if it does not need any additional memory – it needs **O(1)** additional memory beyond the items being sorted

→ does not need to allocate extra memory for the sorting algorith

→ quicksort, insertion sort, selection sort are in-place algorithms but merge sort is not

WE PREFER IN-PLACE ALGORITHMS BECAUSE THEY ARE MEMORY EFFICIENT

Sorting Algorithms

2.) **RECURSIVE**: a sorting algorithm may be implemented with recursion (the divide-and-conquer approaches) or without recursion

→ most of the sorting algorithms are not recursive

(bubble sort, insertion sort, selection sort ...)

→ quicksort and merge sort are recursively implemented sorted algorithms

Sorting Algorithms

3.) **STABLE SORTING**: a stable sorting algorithm maintains the relative order of items with equal values (keys)

4	12	-3	12	16	*before sorting*
-3	4	12	12	16	*after sorting*

(insetion sort and merge sort are stable sorting algorithms but quicksort on the other hand is unstable)

Sorting Algorithms

It is crucial to use stable sorting approaches if we sort by multiple columns in a dataset (or database)

EMPLOYEE DATABASE	
NAME	**COMPANY**
Bill	Microsoft
Adam	Google
Emily	Google
Kevin	HP
Michael	Google
Daniel	British Patrol (BP)

Sorting Algorithms

It is crucial to use stable sorting approaches if we sort by multiple columns in a dataset (or database)

EMPLOYEE DATABASE	
NAME	**COMPANY**
Adam	Google
Bill	Microsoft
Daniel	British Patrol (BP)
Emily	Google
Kevin	HP
Michael	Google

Sorting Algorithms

It is crucial to use stable sorting approaches if we sort by multiple columns in a dataset (or database)

we are using ***stable sorting*** *approach (!!!)*

EMPLOYEE DATABASE	
NAME	**COMPANY**
Daniel	British Patrol
Adam	Google
Emily	Google
Michael	Google
Kevin	HP
Bill	Microsoft

Sorting Algorithms

It is crucial to use stable sorting approaches if we sort by multiple columns in a dataset (or database)

EMPLOYEE DATABASE	
NAME	**COMPANY**
Bill	Microsoft
Adam	Google
Emily	Google
Kevin	HP
Michael	Google
Daniel	British Patrol (BP)

Sorting Algorithms

It is crucial to use stable sorting approaches if we sort by multiple columns in a dataset (or database)

EMPLOYEE DATABASE	
NAME	**COMPANY**
Adam	Google
Bill	Microsoft
Daniel	British Patrol (BP)
Emily	Google
Kevin	HP
Michael	Google

Sorting Algorithms

It is crucial to use stable sorting approaches if we sort by multiple columns in a dataset (or database)

we are using ***unstable sorting*** *approach (!!!)*

EMPLOYEE DATABASE	
NAME	**COMPANY**
Daniel	British Patrol
Emily	Google
Michael	Google
Adam	Google
Kevin	HP
Bill	Microsoft

Adaptive Sorting Algorithms

(Algorithms and Data Structures)

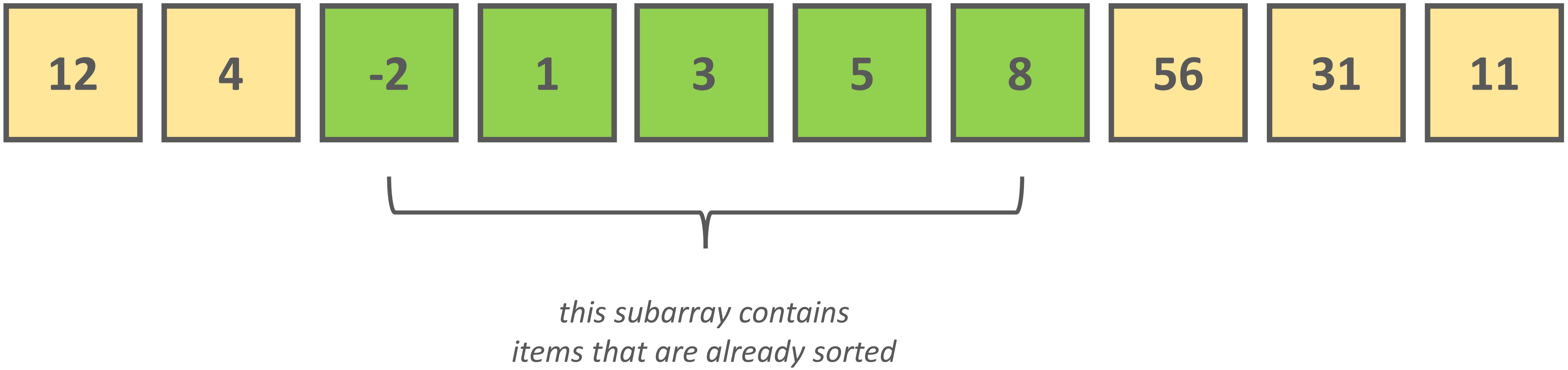
Adaptive Sorting Algorithms

- **adaptive algorithms** change their behavior based on information available at **run-time**
- adaptive sorting approach takes advantage of existing **local order** in its input
- sometimes the subset of the original array is sorted by default – in these cases sorting algorithms will be faster
- most of the times we just have to modify existing sorting algorithms to end up with adaptive approaches

Adaptive Sorting Algorithms

12	4	-2	1	3	5	8	56	31	11

Adaptive Sorting Algorithms



Adaptive Sorting Algorithms

- comparison based sorting algorithms can not do bettern than **O(NlogN)** linearithmic running time
- but what if there are *local sorted regions* in the input?
- in these cases even **O(N)** linear running time can be achieved
- **IMPORTANT**: nearly sorted arrays are quite common in practise
- Heapsort and merge sort approaches do not take advantage of presorted sequences
- **BUT INSERTION SORT AND SHELL SORT ARE ADAPTIVE ALGORITHMS**

Bogo Sort Algorithm

(Algorithms and Data Structures)

Bogo Sort Algorithm

- **bogo sort** is also known as shotgun sort or permutation sort
- the algorithm keeps generating permutations of the input until it finds the sorted order
- it is a particularly inefficient sorting method – there are **N!** permutations for **N** items
- this is why the running time complexity is **O(N!)** factorial

Bogo Sort Algorithm

There are **2** variants:

1.) DETERMINISTIC ALGORITHM

The algorithm enumerates all possible permutations until it finds the sorted order

2.) RANDOMIZED ALGORITHM

The algorithm *randomly* permutates the input until it finds the sorted order – still has **O(N!)** running time

Bogo Sort Algorithm

12	4	-2	1	3

Bogo Sort Algorithm

4	-2	1	12	3

Bogo Sort Algorithm

3	12	4	-2	1

Bogo Sort Algorithm

-2	12	4	1	3

Bogo Sort Algorithm

-2	1	3	4	12

Bogo Sort Algorithm

- why to consider maybe the slowest sorting algorithm possible?
- it is indeed inefficient - for **classical computers**
- if we try to solve the same problem with **quantum computers** then it is the fastest approach possible with **O(1)** running time
- because of quantum entanglement we can „search” for every possible permutations simultaneously

Bubble Sort Algorithm

(Algorithms and Data Structures)

Bubble Sort Algorithm

- **bubble sort** repeatedely steps through the list to be sorted - compares each pair of adjacent items and swaps them if they are in the wrong order
- it is too slow and impractical for most problems even when compared to insertion sort
- bubble sort has worst-case and average-case complexity both $\mathbf{O(N^2)}$
- this is why it is not a practical sorting algorithm
- it is not efficient in the case of a reverse-ordered collection as well

Bubble Sort Algorithm

- in **computer graphics** bubble sort is popular for its capability to detect a very small error (like swap of just two elements) in almost-sorted arrays and fix it
- in these cases bubble sort may run in **O(N)** linear complexity
- it is used in a polygon filling algorithm where bounding lines are sorted by their **x** coordinate at a specific scan line (a line parallel to **x** axis) and with incrementing **y** their order changes (two elements are swapped) only at intersections of two lines
- bubble sort is a **stable** sorting algorithm

Bubble Sort Algorithm

12	4	-2	1	3	5	8	56	31	11

Bubble Sort Algorithm

12	4	-2	1	3	5	8	56	31	11

Bubble Sort Algorithm

12	4	-2	1	3	5	8	56	31	11

Bubble Sort Algorithm

4	12	-2	1	3	5	8	56	31	11

Bubble Sort Algorithm

4	12	-2	1	3	5	8	56	31	11

Bubble Sort Algorithm

4	12	-2	1	3	5	8	56	31	11

Bubble Sort Algorithm

4	-2	12	1	3	5	8	56	31	11

Bubble Sort Algorithm

4	-2	12	1	3	5	8	56	31	11

Bubble Sort Algorithm

4	-2	12	1	3	5	8	56	31	11

Bubble Sort Algorithm

4	-2	1	12	3	5	8	56	31	11

Bubble Sort Algorithm

4	-2	1	12	3	5	8	56	31	11

Bubble Sort Algorithm

4	-2	1	12	3	5	8	56	31	11

Bubble Sort Algorithm

4	-2	1	3	12	5	8	56	31	11

Bubble Sort Algorithm

4	-2	1	3	12	5	8	56	31	11

Bubble Sort Algorithm

4	-2	1	3	12	5	8	56	31	11

Bubble Sort Algorithm

4	-2	1	3	5	12	8	56	31	11

Bubble Sort Algorithm

4	-2	1	3	5	12	8	56	31	11

Bubble Sort Algorithm

4	-2	1	3	5	12	8	56	31	11

Bubble Sort Algorithm

4	-2	1	3	5	8	12	56	31	11

Bubble Sort Algorithm

4	-2	1	3	5	8	12	56	31	11

Bubble Sort Algorithm

4	-2	1	3	5	8	12	56	31	11

Bubble Sort Algorithm

4	-2	1	3	5	8	12	56	31	11

Bubble Sort Algorithm

4	-2	1	3	5	8	12	56	31	11

Bubble Sort Algorithm

4	-2	1	3	5	8	12	31	56	11

Bubble Sort Algorithm

4	-2	1	3	5	8	12	31	56	11

Bubble Sort Algorithm

4	-2	1	3	5	8	12	31	56	11

Bubble Sort Algorithm

4	-2	1	3	5	8	12	31	11	56

Bubble Sort Algorithm

4	-2	1	3	5	8	12	31	11	56

Bubble Sort Algorithm

4	-2	1	3	5	8	12	31	11	56

Bubble Sort Algorithm

4	-2	1	3	5	8	12	31	11	56

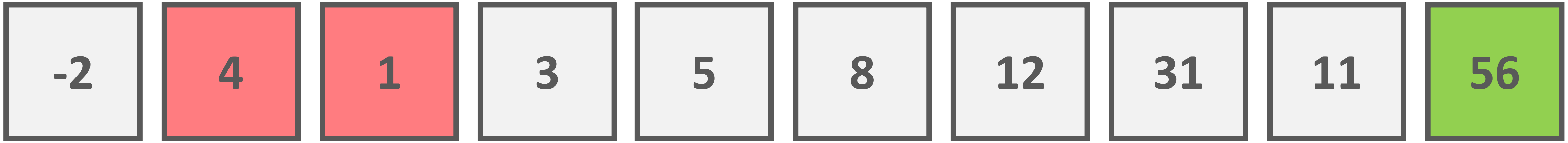
Bubble Sort Algorithm

-2	4	1	3	5	8	12	31	11	56

Bubble Sort Algorithm

-2	4	1	3	5	8	12	31	11	56

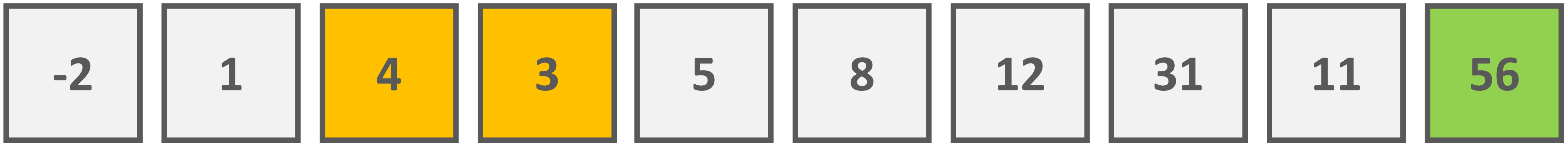
Bubble Sort Algorithm



Bubble Sort Algorithm

-2	1	4	3	5	8	12	31	11	56

Bubble Sort Algorithm



Bubble Sort Algorithm

-2	1	4	3	5	8	12	31	11	56

Bubble Sort Algorithm

-2	1	3	4	5	8	12	31	11	56

Bubble Sort Algorithm

-2	1	3	4	5	8	12	31	11	56

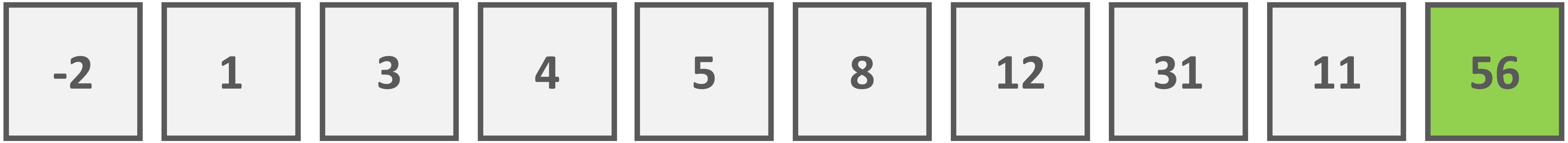
Bubble Sort Algorithm

-2	1	3	4	5	8	12	31	11	56

Bubble Sort Algorithm

-2	1	3	4	5	8	12	31	11	56

Bubble Sort Algorithm



Bubble Sort Algorithm

-2	1	3	4	5	8	12	31	11	56

Bubble Sort Algorithm

-2	1	3	4	5	8	12	31	11	56

Bubble Sort Algorithm

-2	1	3	4	5	8	12	31	11	56

Bubble Sort Algorithm

-2	1	3	4	5	8	12	31	11	56

Bubble Sort Algorithm

-2	1	3	4	5	8	12	31	11	56

Bubble Sort Algorithm

-2	1	3	4	5	8	12	31	11	56

Bubble Sort Algorithm

-2	1	3	4	5	8	12	11	31	56

Bubble Sort Algorithm

-2	1	3	4	5	8	12	11	31	56

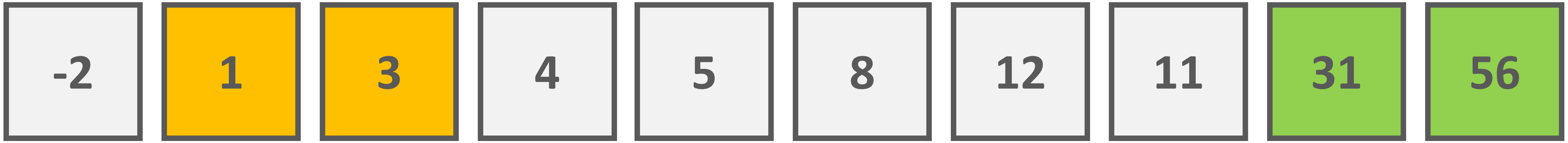
Bubble Sort Algorithm

-2	1	3	4	5	8	12	11	31	56

Bubble Sort Algorithm

-2	1	3	4	5	8	12	11	31	56

Bubble Sort Algorithm



Bubble Sort Algorithm

-2	1	3	4	5	8	12	11	31	56

Bubble Sort Algorithm

-2	1	3	4	5	8	12	11	31	56

Bubble Sort Algorithm

-2	1	3	4	5	8	12	11	31	56

Bubble Sort Algorithm

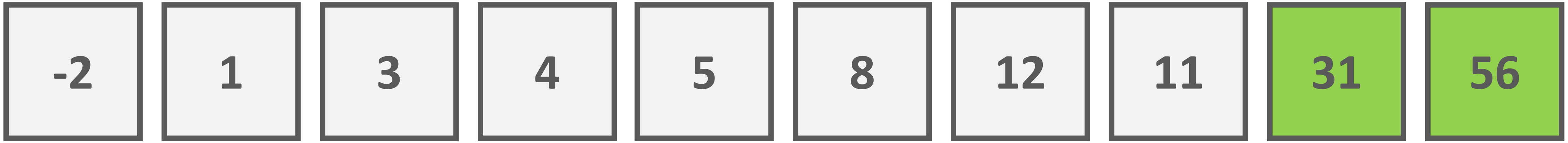
-2	1	3	4	5	8	12	11	31	56

Bubble Sort Algorithm

-2	1	3	4	5	8	12	11	31	56

Bubble Sort Algorithm

Bubble Sort Algorithm



Bubble Sort Algorithm

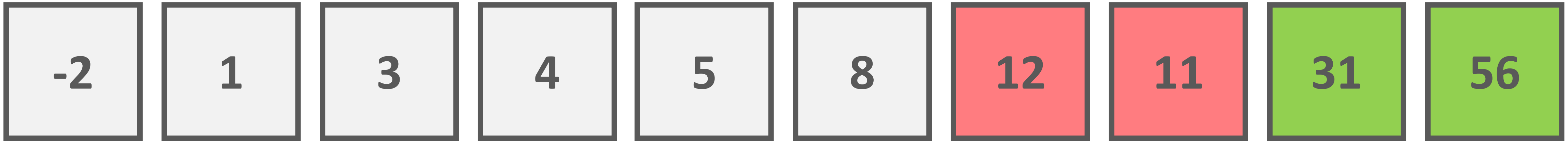
Bubble Sort Algorithm

-2	1	3	4	5	8	12	11	31	56

Bubble Sort Algorithm

-2	1	3	4	5	8	12	11	31	56

Bubble Sort Algorithm



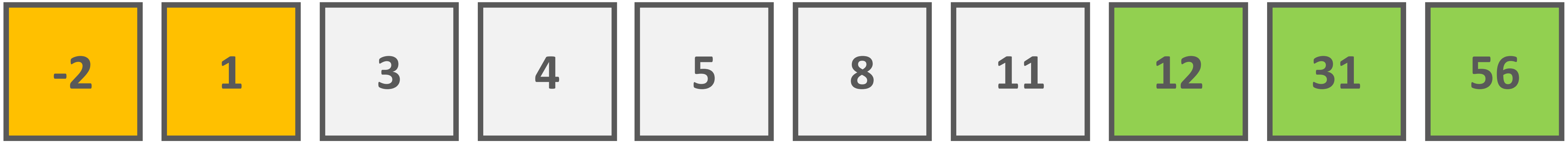
Bubble Sort Algorithm

-2	1	3	4	5	8	11	12	31	56

Bubble Sort Algorithm

-2	1	3	4	5	8	11	12	31	56

Bubble Sort Algorithm



Bubble Sort Algorithm

-2	1	3	4	5	8	11	12	31	56

Bubble Sort Algorithm

-2	1	3	4	5	8	11	12	31	56

Bubble Sort Algorithm

Bubble Sort Algorithm

-2	1	3	4	5	8	11	12	31	56

Bubble Sort Algorithm

-2	1	3	4	5	8	11	12	31	56

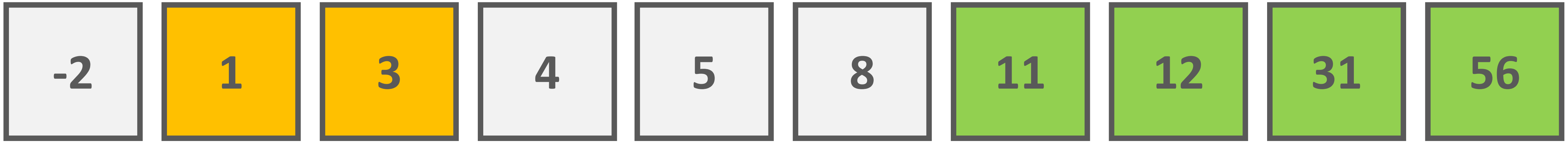
Bubble Sort Algorithm

-2	1	3	4	5	8	11	12	31	56

Bubble Sort Algorithm

-2	1	3	4	5	8	11	12	31	56

Bubble Sort Algorithm



Bubble Sort Algorithm

-2	1	3	4	5	8	11	12	31	56

Bubble Sort Algorithm

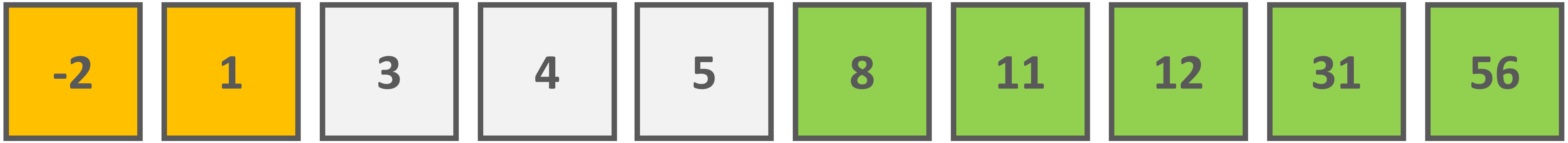
Bubble Sort Algorithm

-2	1	3	4	5	8	11	12	31	56

Bubble Sort Algorithm

-2	1	3	4	5	8	11	12	31	56

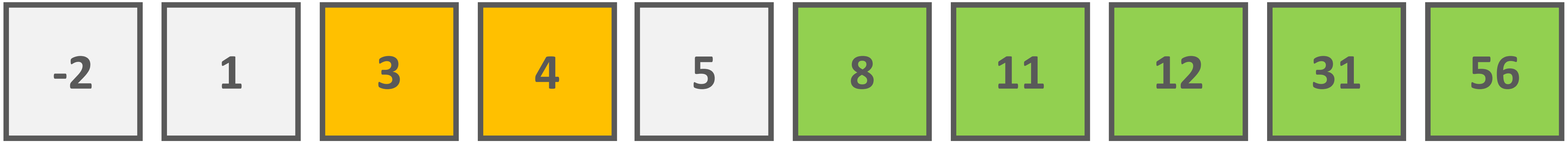
Bubble Sort Algorithm



Bubble Sort Algorithm

-2	1	3	4	5	8	11	12	31	56

Bubble Sort Algorithm



Bubble Sort Algorithm

-2	1	3	4	5	8	11	12	31	56

Bubble Sort Algorithm

-2	1	3	4	5	8	11	12	31	56

Bubble Sort Algorithm

Bubble Sort Algorithm

Bubble Sort Algorithm

Bubble Sort Algorithm

-2	1	3	4	5	8	11	12	31	56

Bubble Sort Algorithm

-2	1	3	4	5	8	11	12	31	56

Bubble Sort Algorithm

-2	1	3	4	5	8	11	12	31	56

Bubble Sort Algorithm

-2	1	3	4	5	8	11	12	31	56

Bubble Sort Algorithm

Bubble Sort Algorithm

-2	1	3	4	5	8	11	12	31	56

Bubble Sort Algorithm

-2	1	3	4	5	8	11	12	31	56

Bubble Sort Algorithm

-2	1	3	4	5	8	11	12	31	56

Selection Sort Algorithm

(Algorithms and Data Structures)

Selection Sort

- selection sort is another **O(N²)** quadratic running time sorting algorithm
- it is noted for its simplicity and it has performance advantages over the more complicated algorithms
- particularly important and useful when auxiliary memory is limited
- the algorithm divides the original array into **2** parts: items already sorted and the items that are not yet sorted

Selection Sort

- the main idea is linear search: we can find the smallest (largest) item in **O(N)** linear running time complexity
- then swap the item with the leftmost item in the array – that is not yet sorted of course
- we have to make linear search for **N-1** items this is why the final running time complexity is **$O(N^2)$**
- it is **in-place** so it does not need additional memory
- selection sort is **not a stable** sorting algorithm
- selection sort always outperforms bubble sort

Selection Sort

- selection sort and insertion sort are rather slow approaches – but they are faster with small arrays (**5-10** items)
- this is why the fast sorting approaches use selection sort and insertion sort when the number of items **< 10**
- **it makes less writes than insertion sort** – it is crucial when writes are significantly more expensive than reads
- important with **EEPROM** and **flash memory** where every write lessens the lifespan of the memory

Selection Sort

-2	1	13	5	8	-5

Selection Sort

-2	1	13	5	8	-5

Selection Sort

-2	1	13	5	8	-5

Selection Sort

Selection Sort

Selection Sort

-2	1	13	5	8	-5

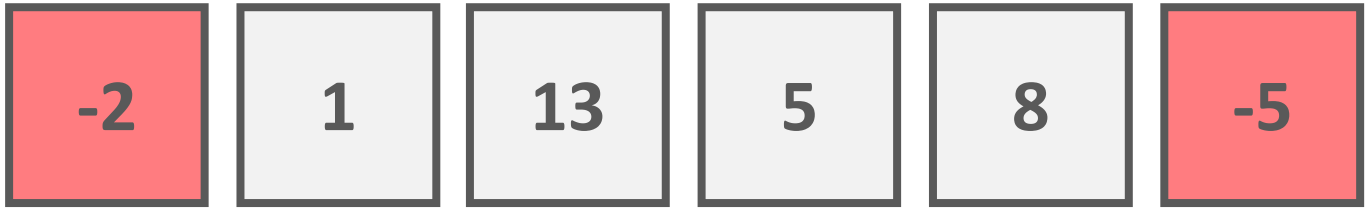
Selection Sort

-2	1	13	5	8	-5

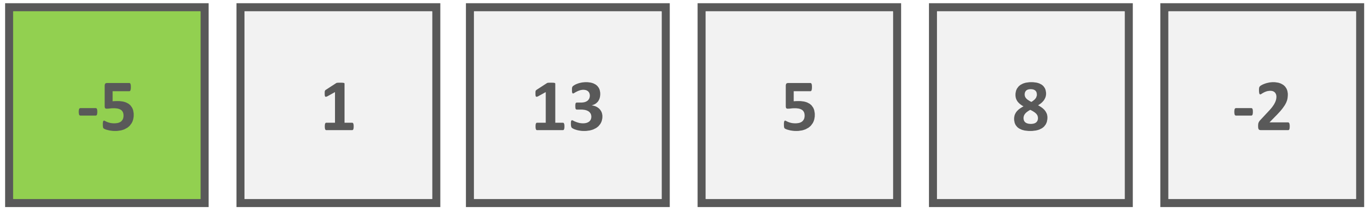
Selection Sort

-2	1	13	5	8	-5

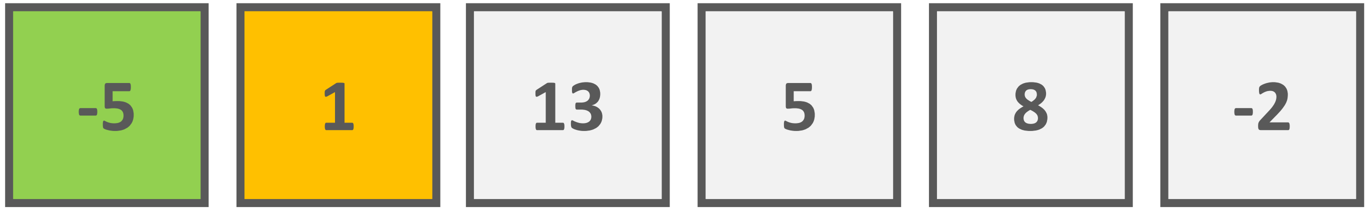
Selection Sort



Selection Sort



Selection Sort

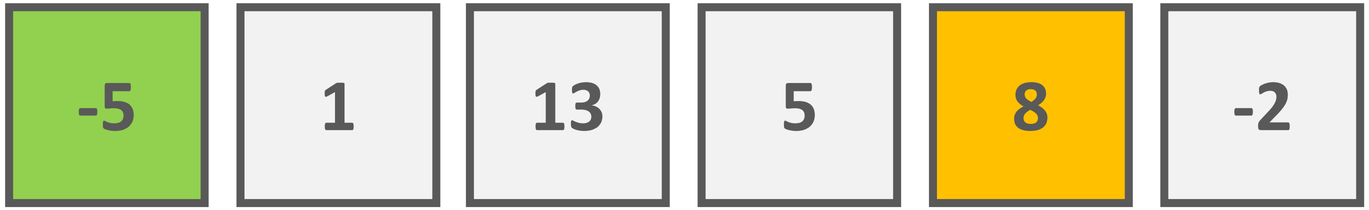


Selection Sort

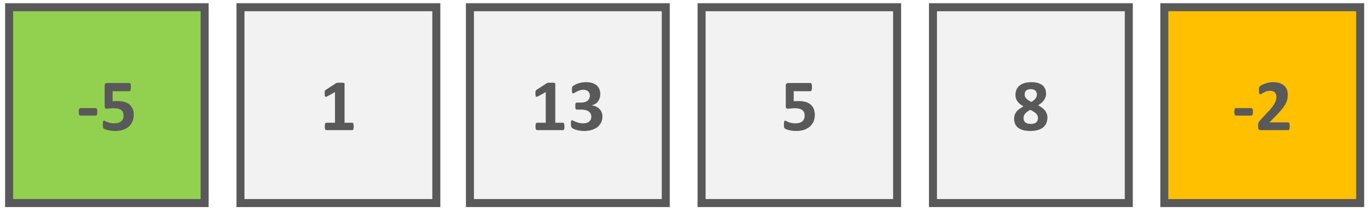
-5	1	13	5	8	-2

Selection Sort

Selection Sort



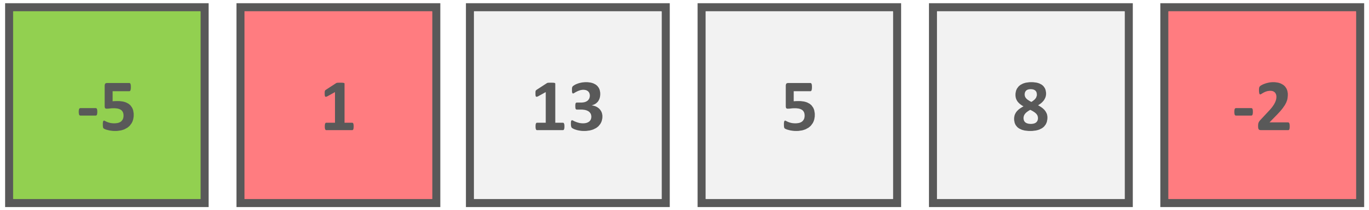
Selection Sort



Selection Sort

-5	1	13	5	8	-2

Selection Sort



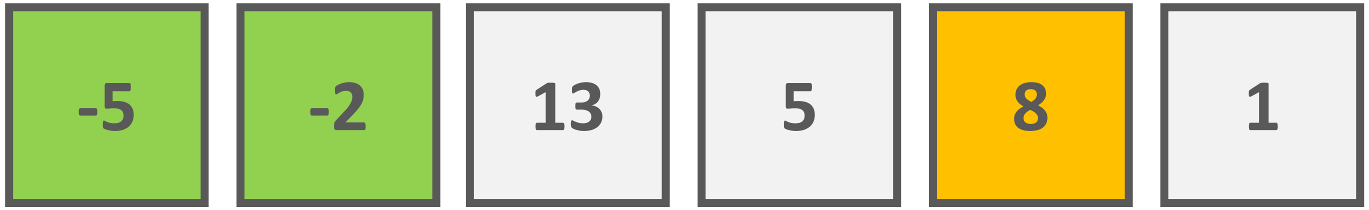
Selection Sort

-5	-2	13	5	8	1

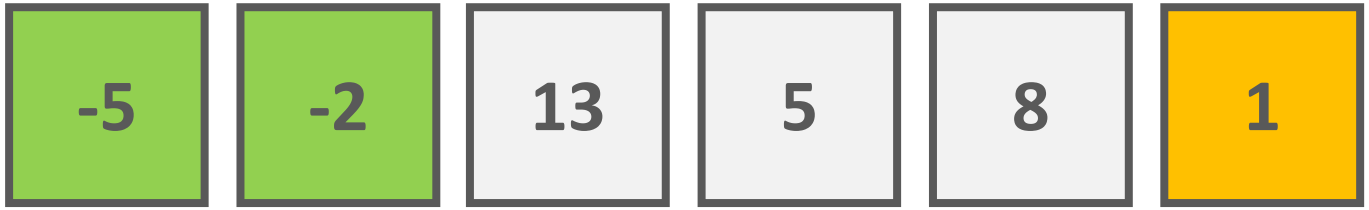
Selection Sort

Selection Sort

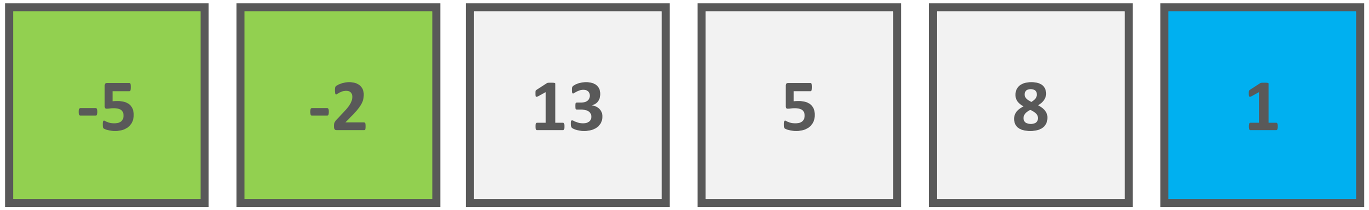
Selection Sort



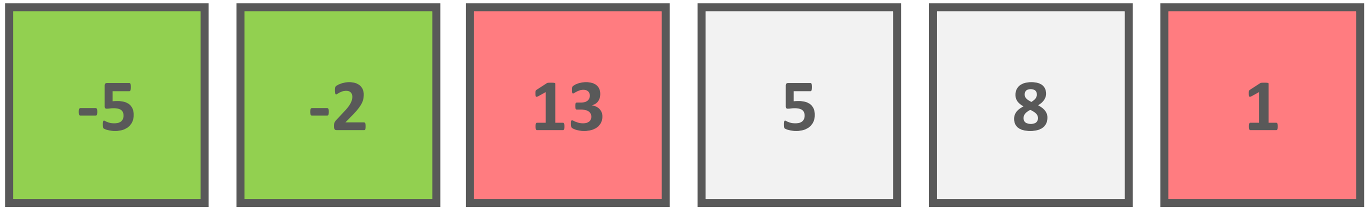
Selection Sort



Selection Sort



Selection Sort



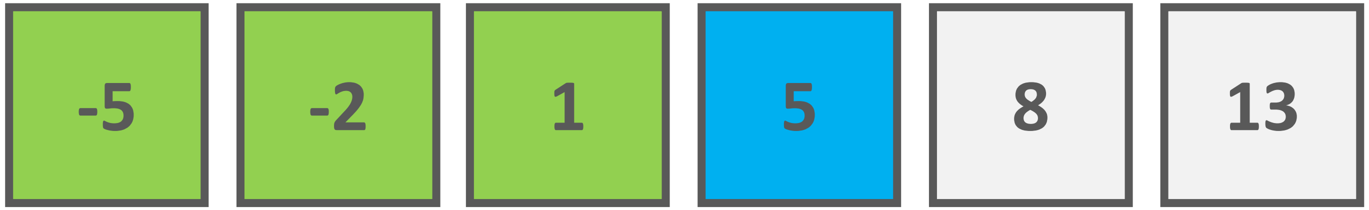
Selection Sort

Selection Sort

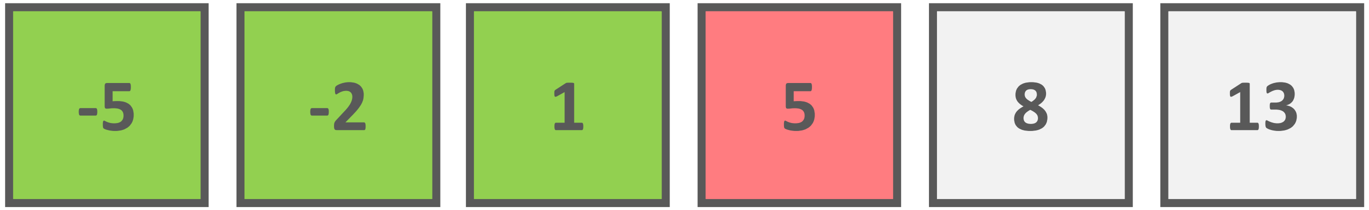
Selection Sort

Selection Sort

Selection Sort



Selection Sort



Selection Sort

-5	-2	1	5	8	13

Selection Sort

Selection Sort

Selection Sort

Selection Sort

Selection Sort

Selection Sort

Selection Sort

Insertion Sort

(Algorithms and Data Structures)

Insertion Sort

- insertion sort is another **O(N^2)** quadratic running time algorithm
- on large datasets it is very inefficient - but on arrays with **10-20** items it is quite good
- a huge advantage is that it is easy to implement it
- it is more efficient than other quadratic running time sorting procedures such as bubble sort or selection sort
- it is an **adaptive algorithm** – it speeds up when array is already substantially sorted
- it is **stable** so preserves the order of the items with equal keys

Insertion Sort

- insertion sort is an **in-place algorithm** – does not need any additional memory
- it is an **online algorithm** – it can sort an array as it receives the items for example downloading data from web
- hybrid algorithms uses insertion sort if the subarray is small enough: insertion sort is faster for small subarrays than quicksort !!!
- variant of insertion sort is **shell sort**

Insertion Sort

- sometimes selection sort is better: they are very similar algorithms
- insertion sort requires more writes because the inner loop can require shifting large sections of the sorted portion of the array
- in general insertion sort will write to the array **$O(N^2)$** times while selection sort will write only **O(N)** times
- for this reason selection sort may be preferable in cases where writing to memory is significantly more expensive than reading (such as with flash memory)

Insertion Sort

12	4	-2	11	3	43	2	56	31	1

Insertion Sort

12	4	-2	11	3	43	2	56	31	1

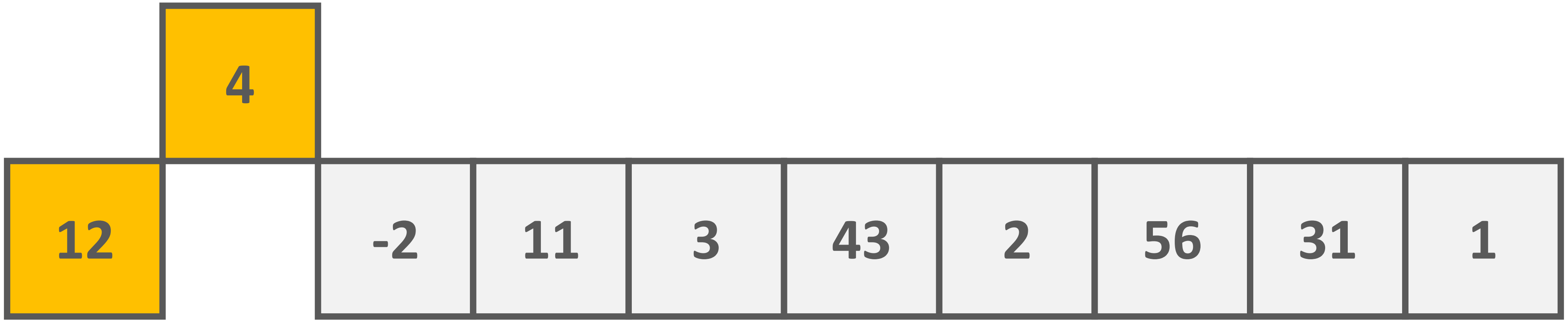
Insertion Sort

12	4	-2	11	3	43	2	56	31	1

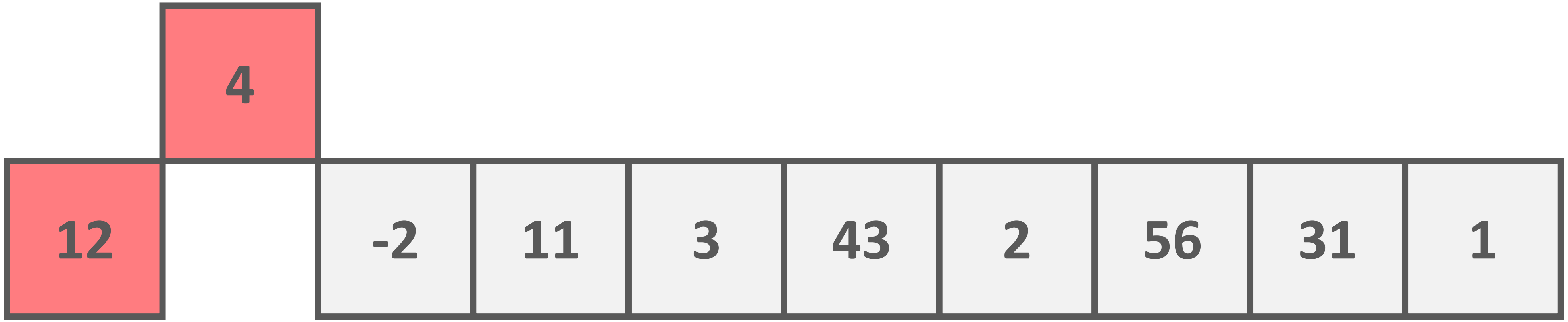
Insertion Sort

12	4	-2	11	3	43	2	56	31	1

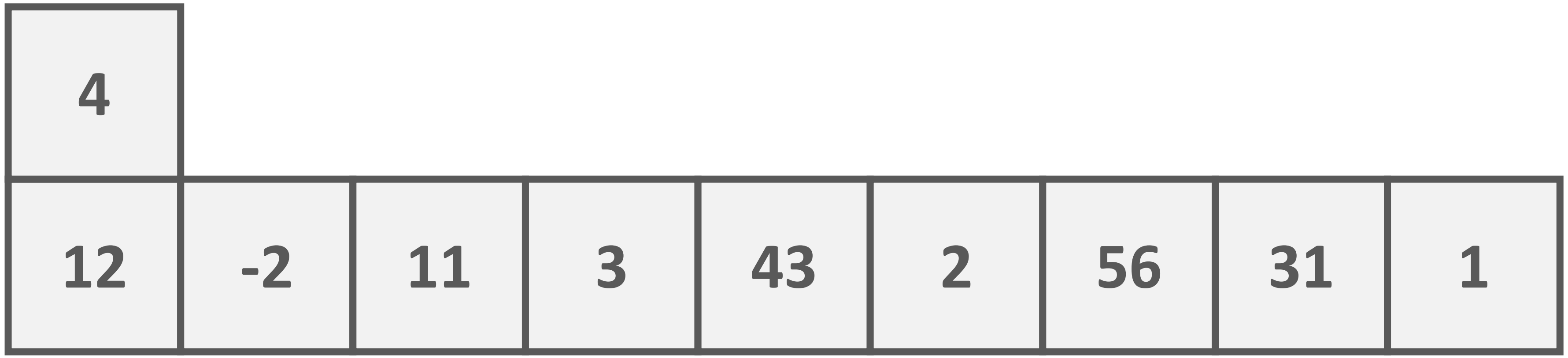
Insertion Sort



Insertion Sort



Insertion Sort



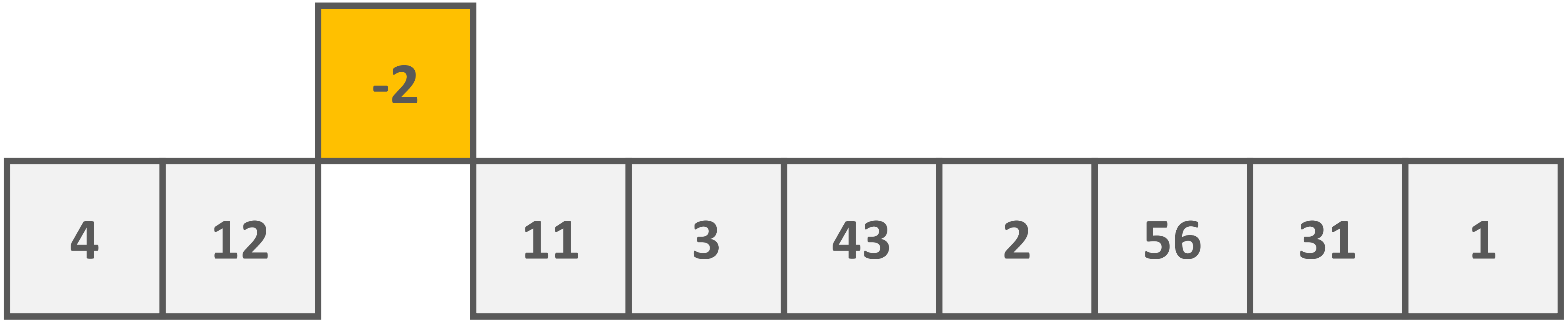
Insertion Sort

4	12	-2	11	3	43	2	56	31	1

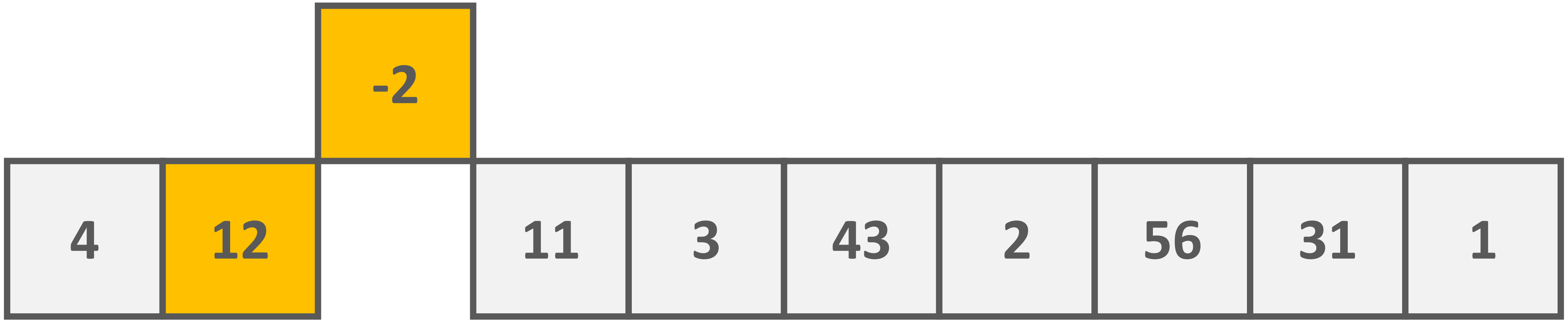
Insertion Sort

4	12	-2	11	3	43	2	56	31	1

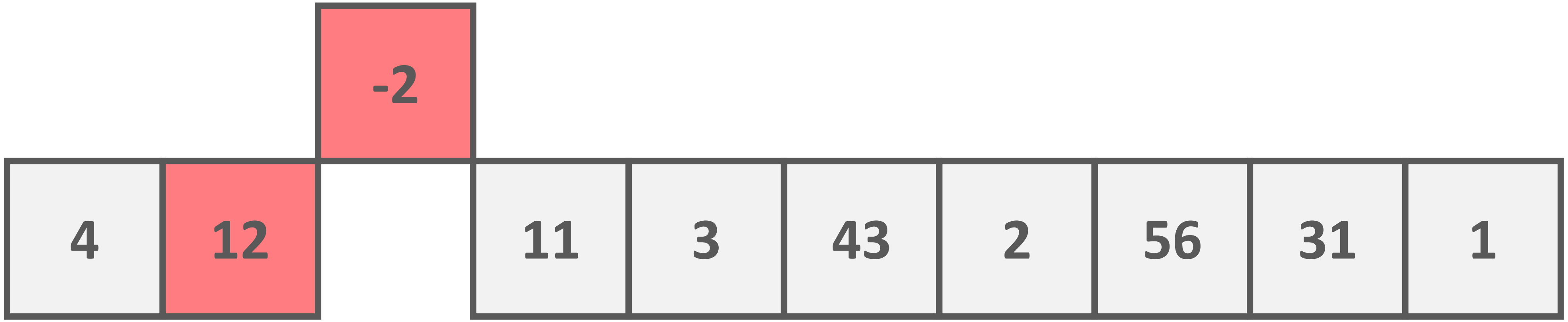
Insertion Sort



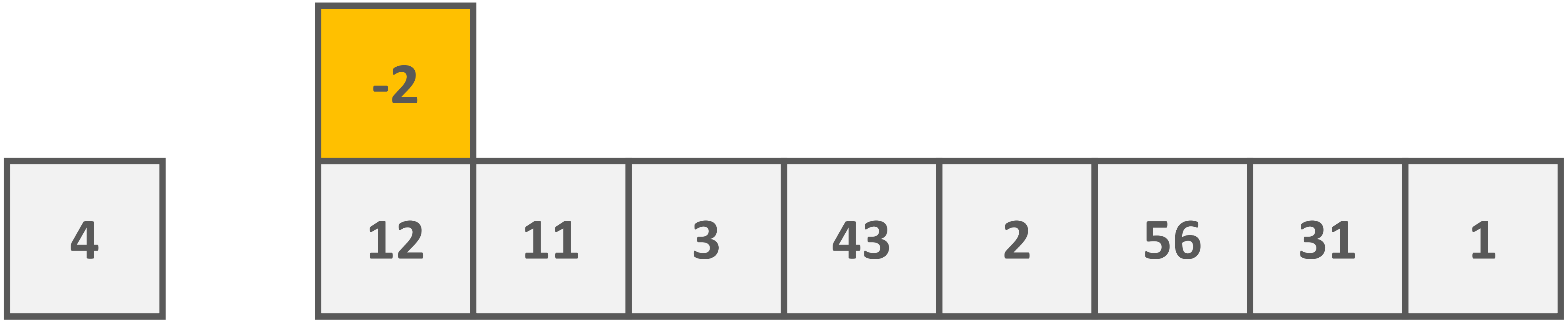
Insertion Sort



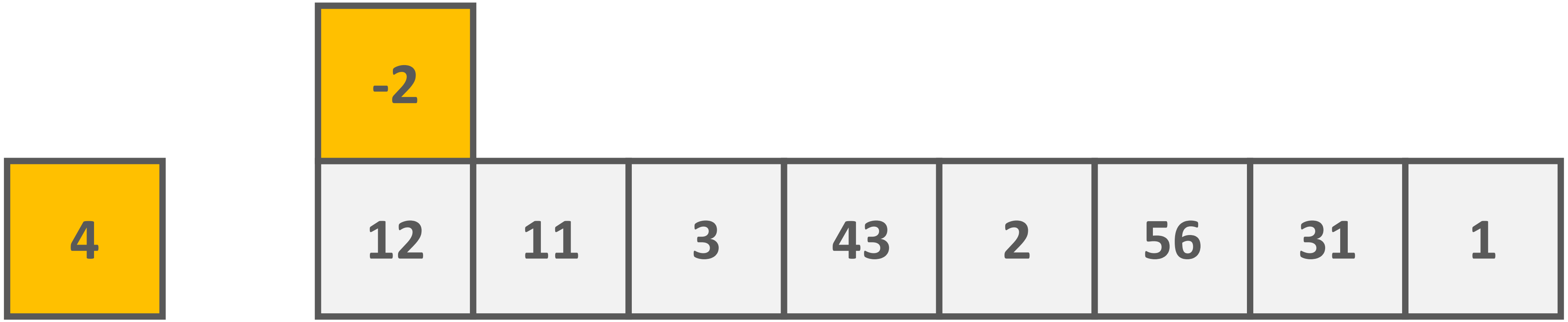
Insertion Sort



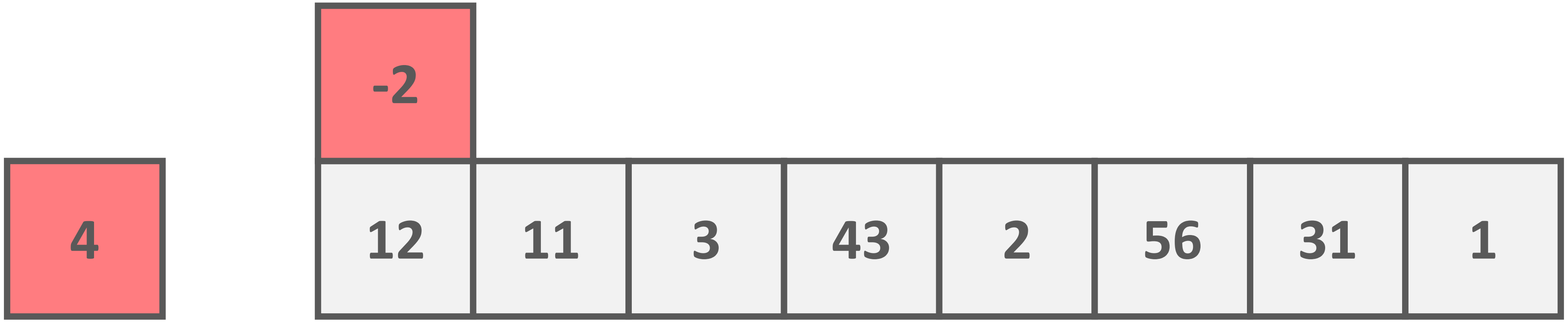
Insertion Sort



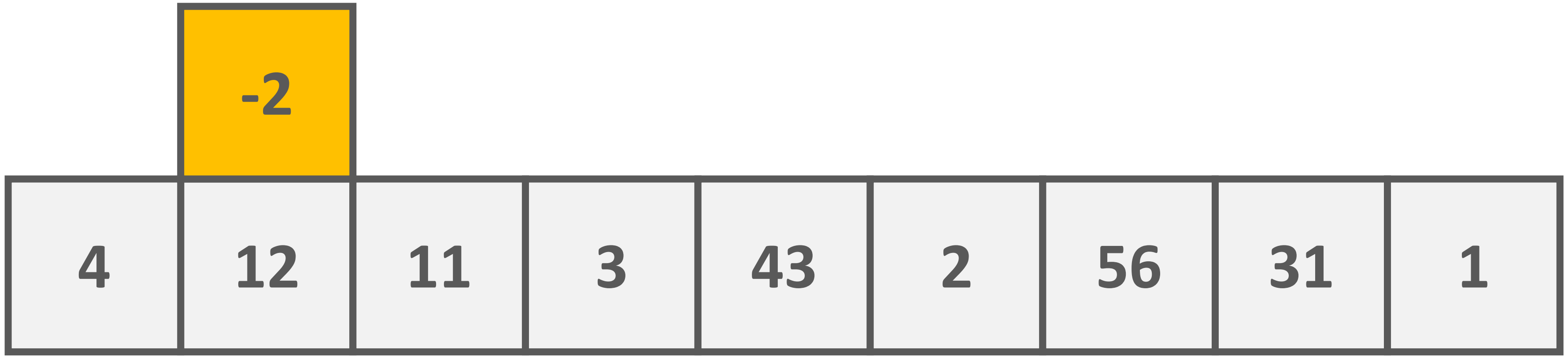
Insertion Sort



Insertion Sort



Insertion Sort



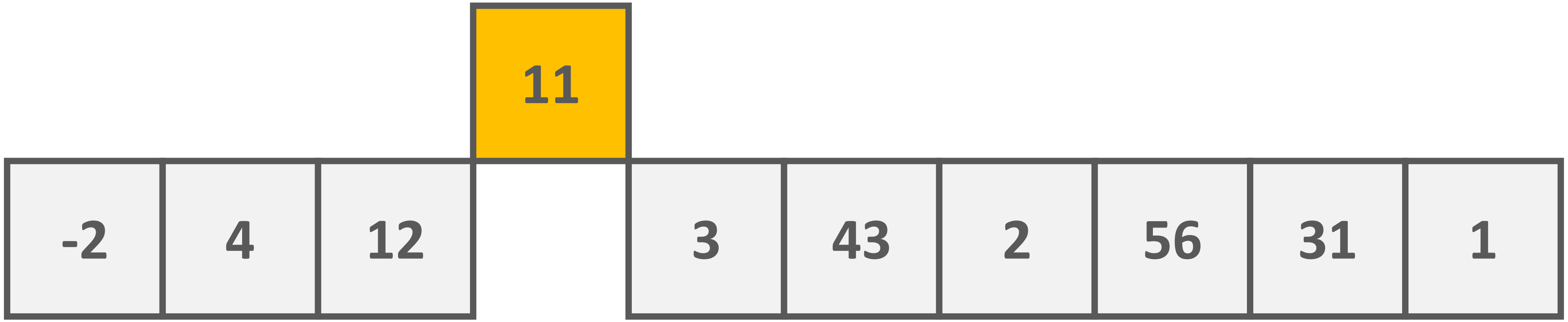
Insertion Sort

-2	4	12	11	3	43	2	56	31	1

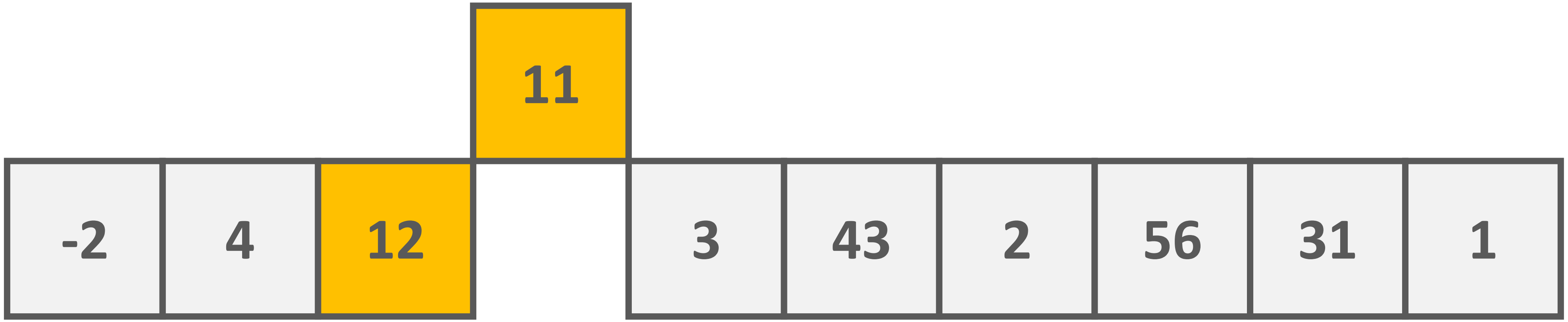
Insertion Sort

-2	4	12	11	3	43	2	56	31	1

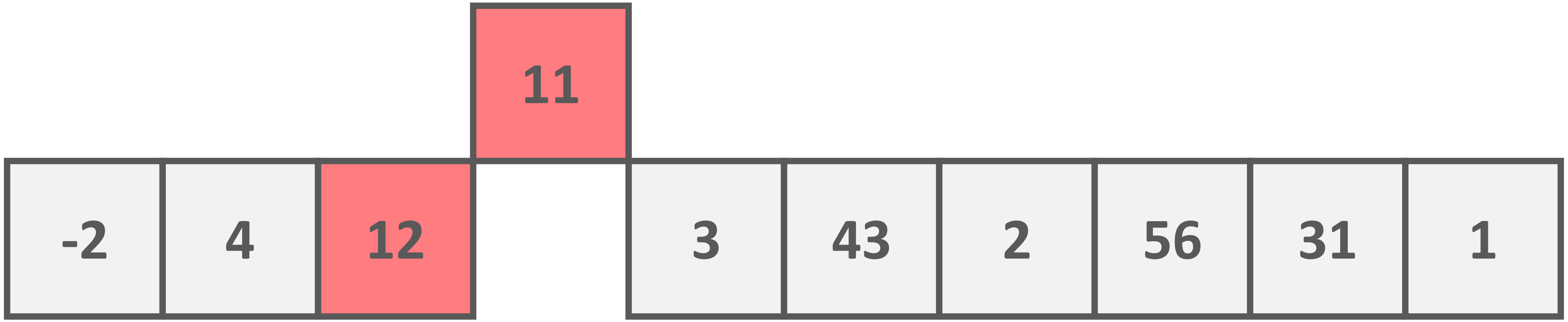
Insertion Sort



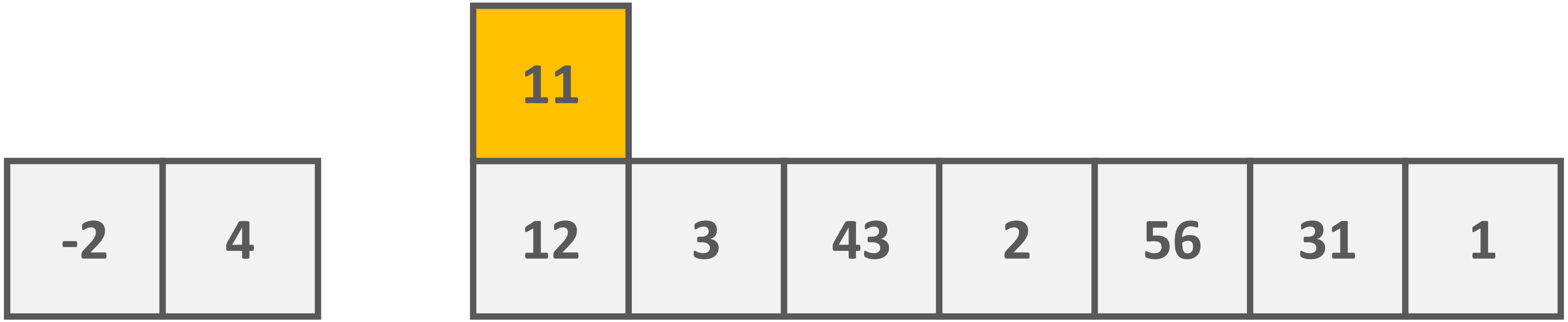
Insertion Sort



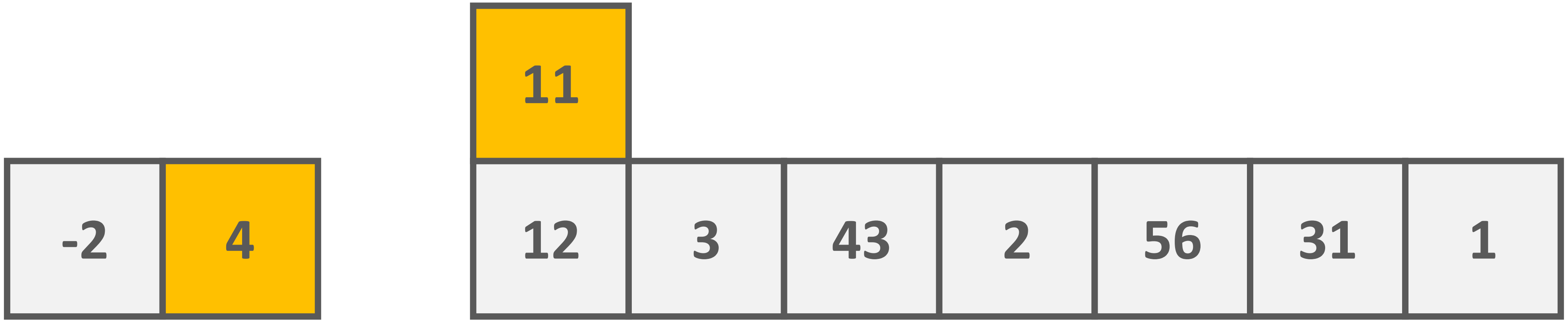
Insertion Sort



Insertion Sort



Insertion Sort



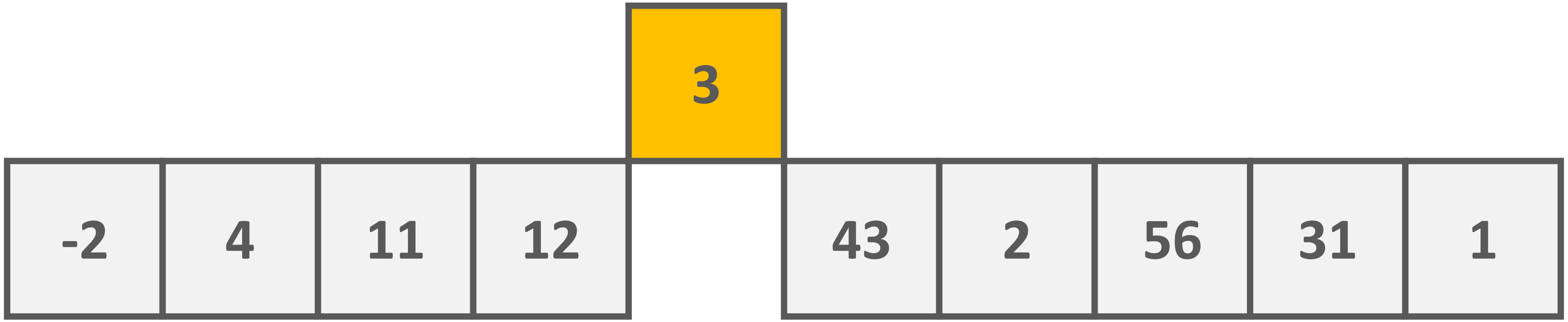
Insertion Sort

-2	4	11	12	3	43	2	56	31	1

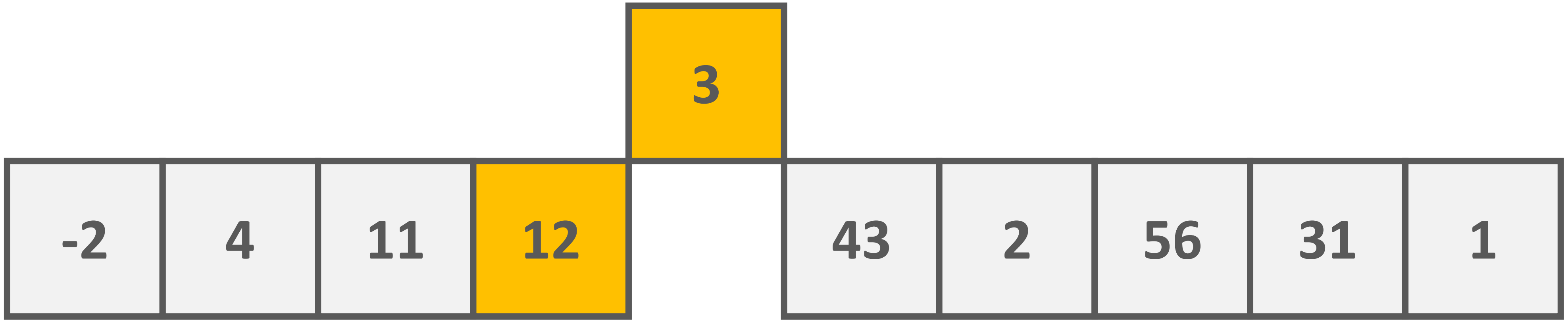
Insertion Sort

-2	4	11	12	3	43	2	56	31	1

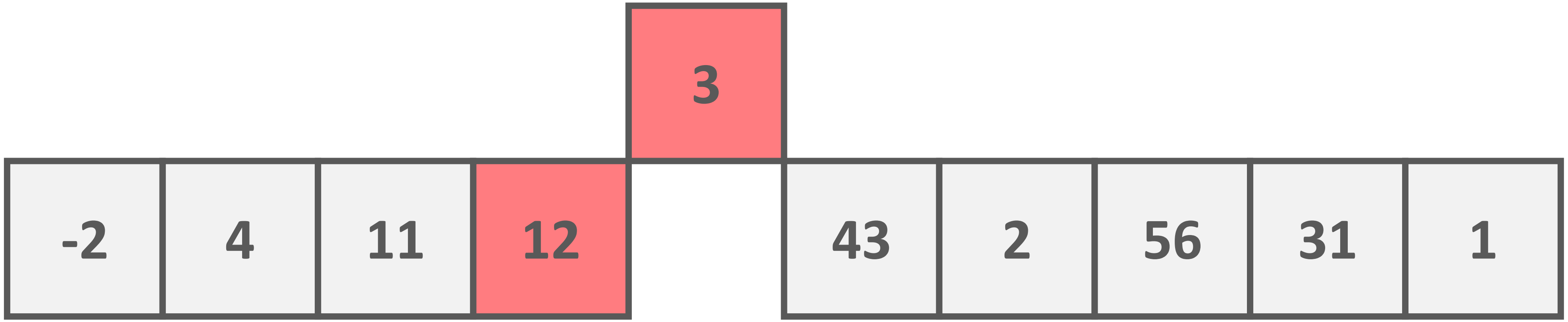
Insertion Sort



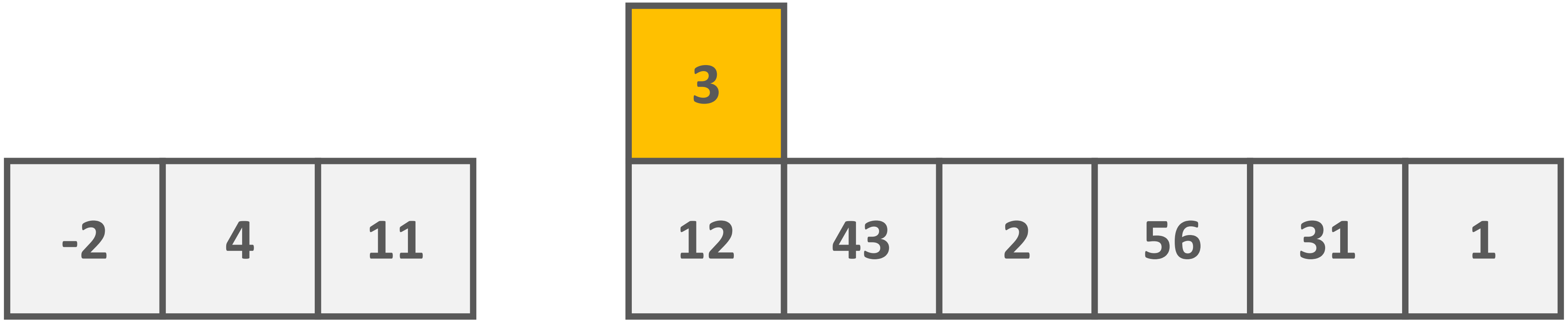
Insertion Sort



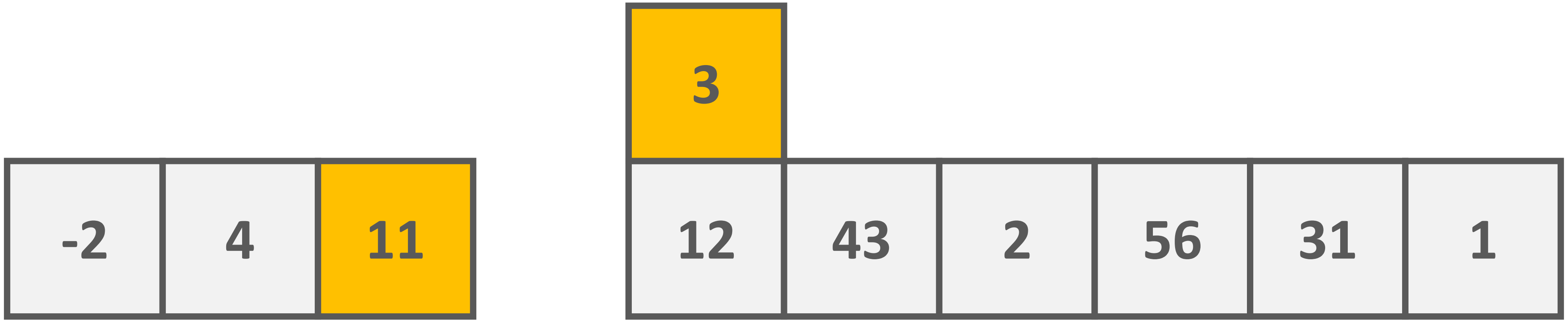
Insertion Sort



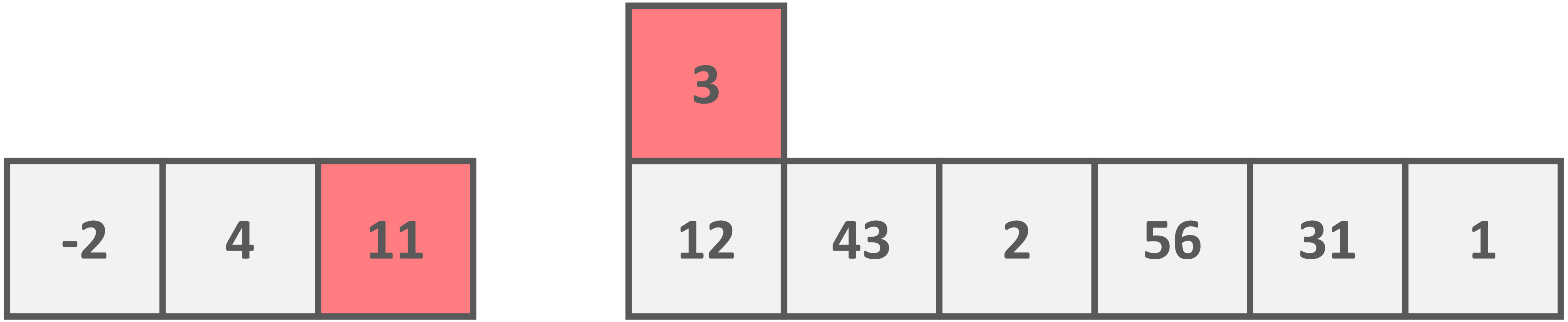
Insertion Sort



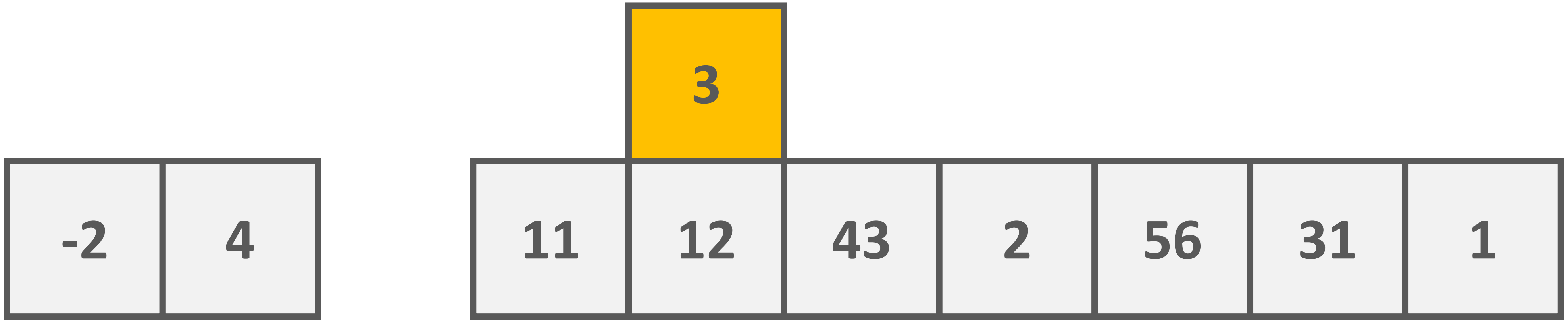
Insertion Sort



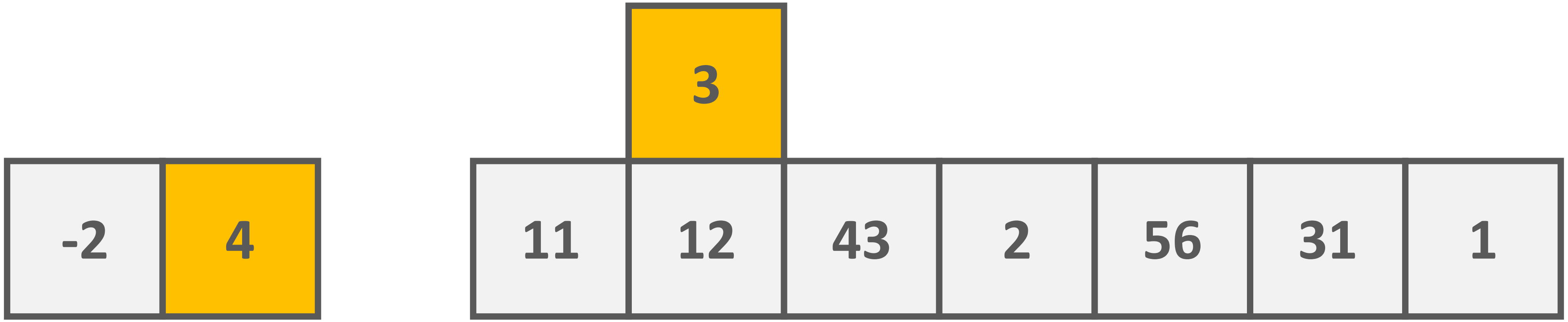
Insertion Sort



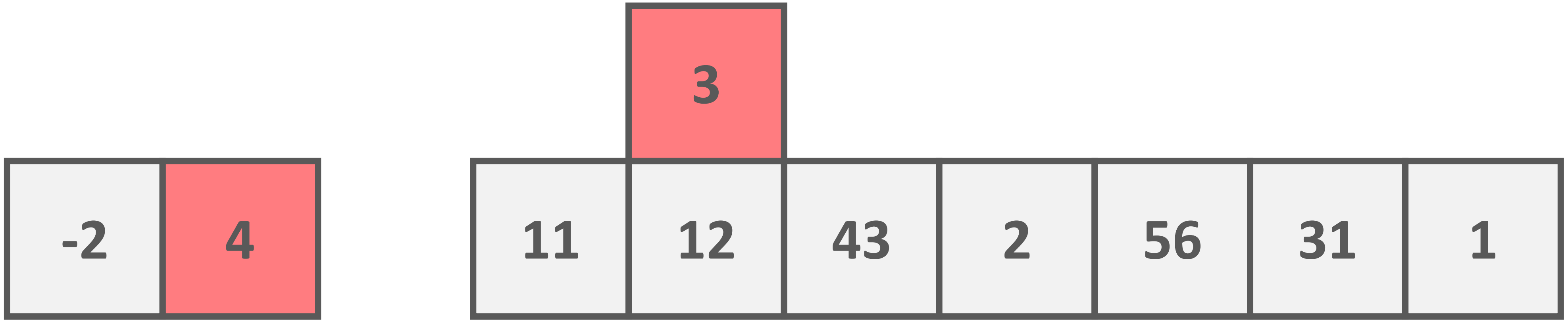
Insertion Sort



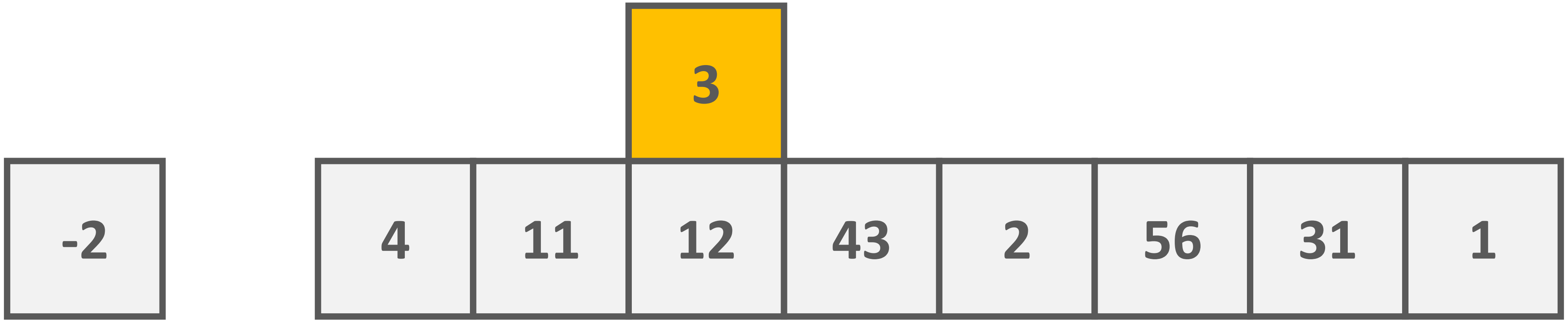
Insertion Sort



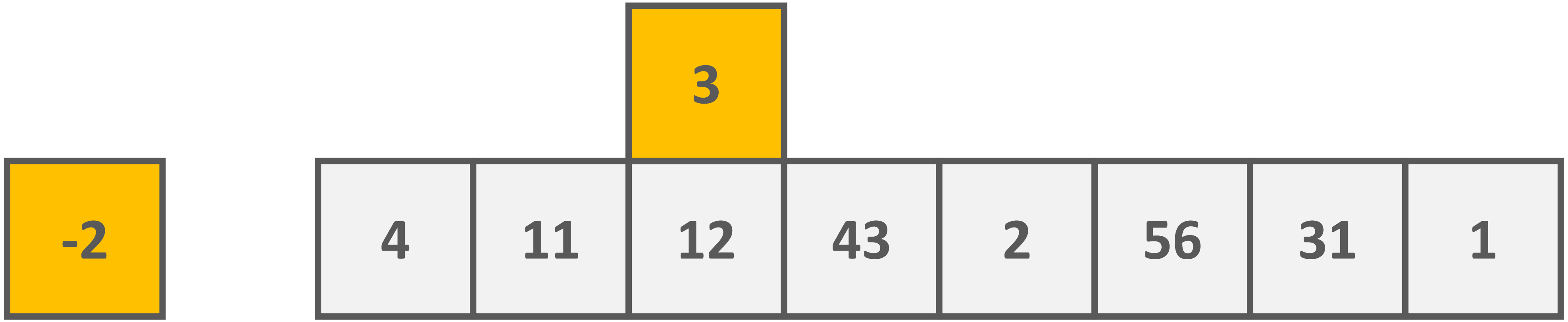
Insertion Sort



Insertion Sort



Insertion Sort



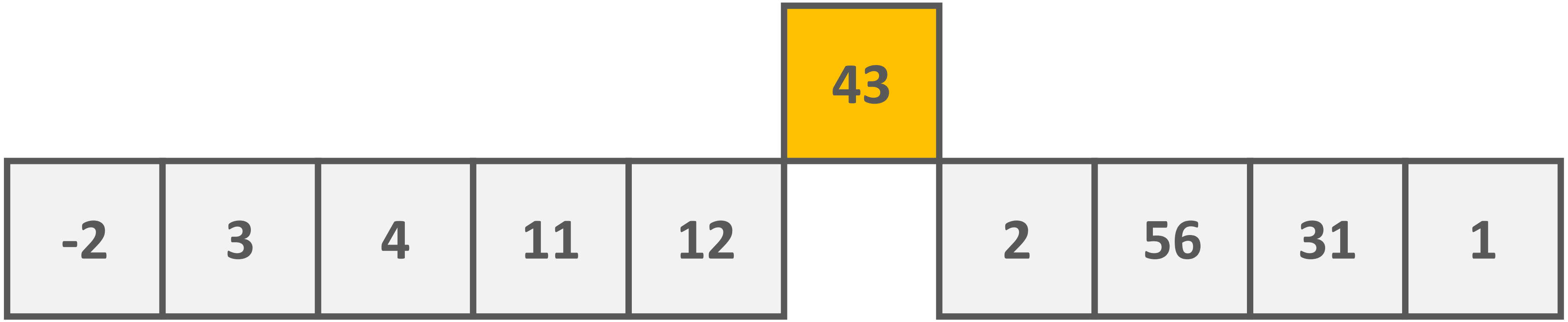
Insertion Sort

-2	3	4	11	12	43	2	56	31	1

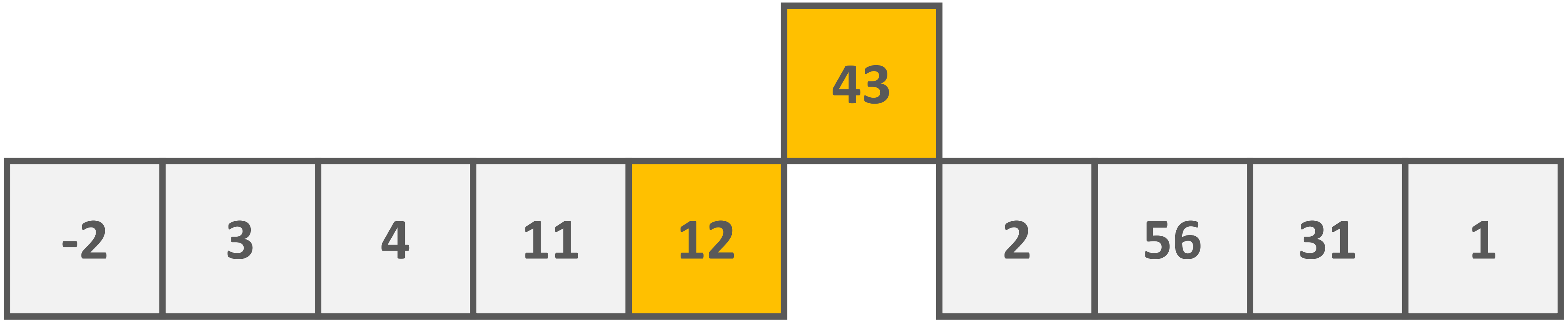
Insertion Sort

-2	3	4	11	12	43	2	56	31	1

Insertion Sort



Insertion Sort



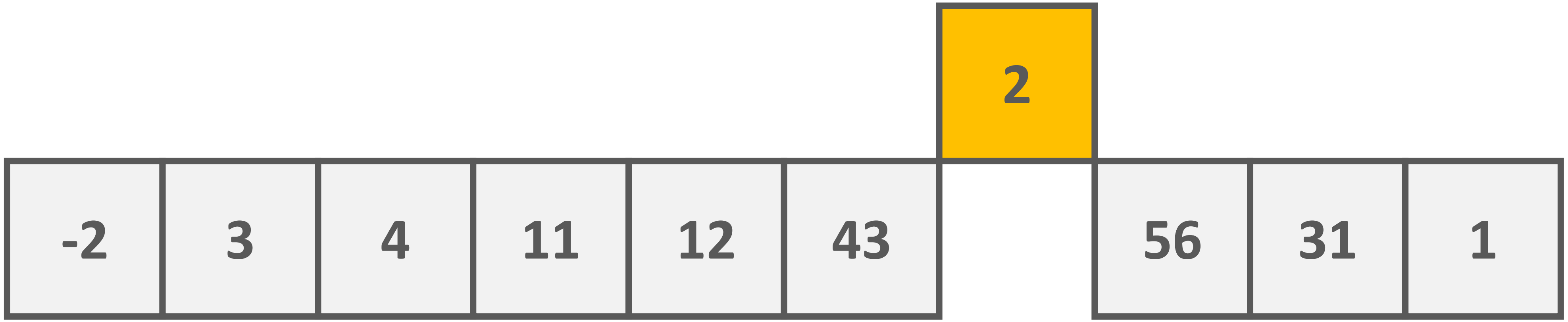
Insertion Sort

-2	3	4	11	12	43	2	56	31	1

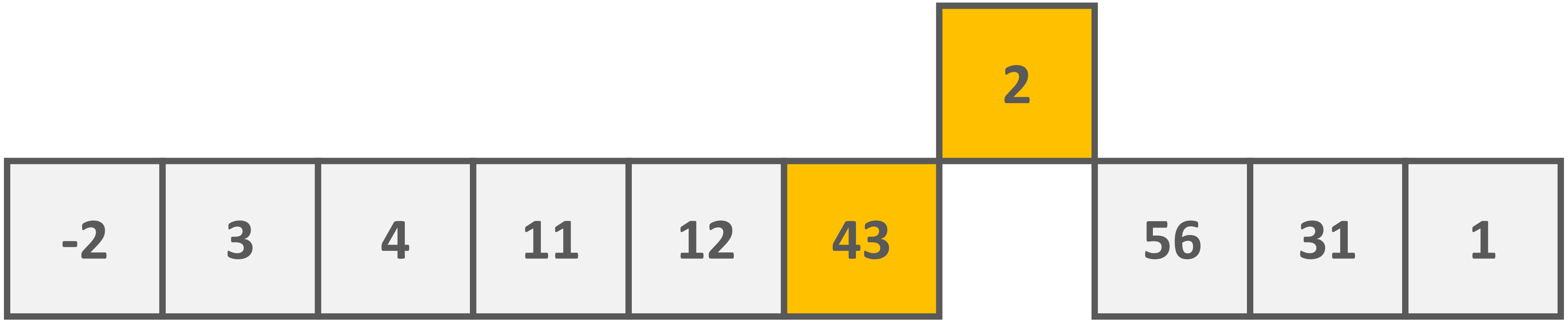
Insertion Sort

-2	3	4	11	12	43	2	56	31	1

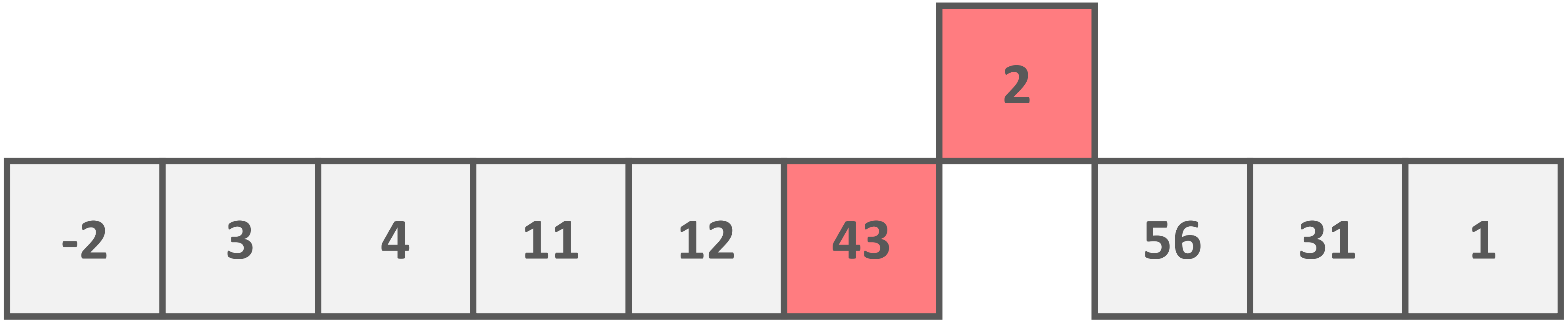
Insertion Sort



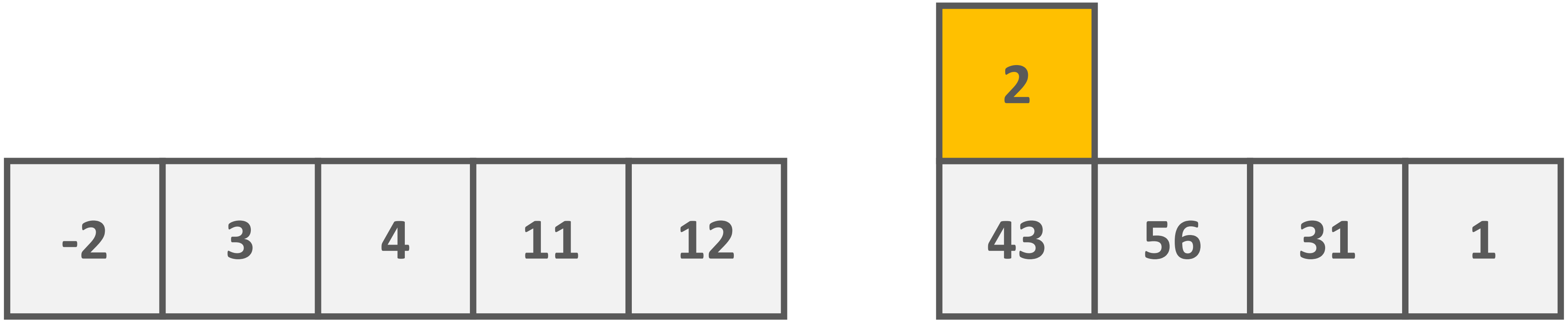
Insertion Sort



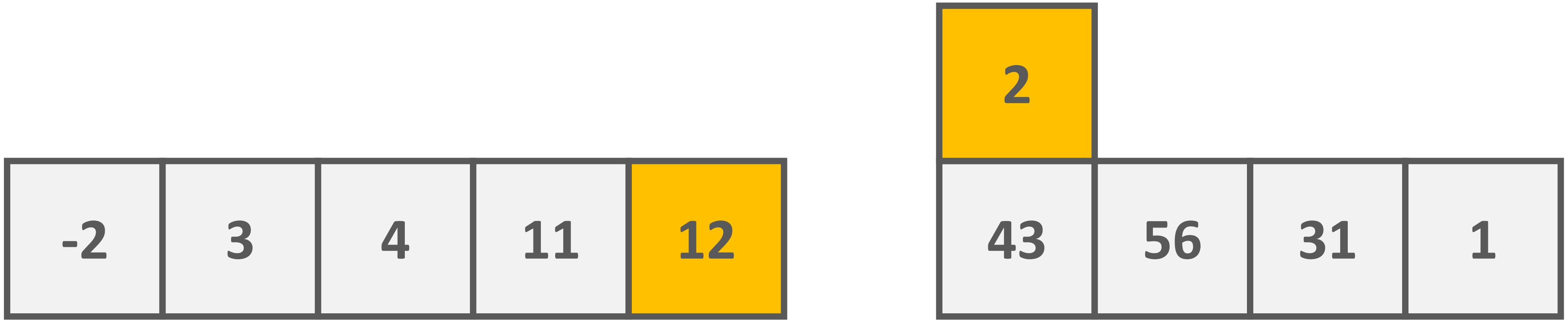
Insertion Sort



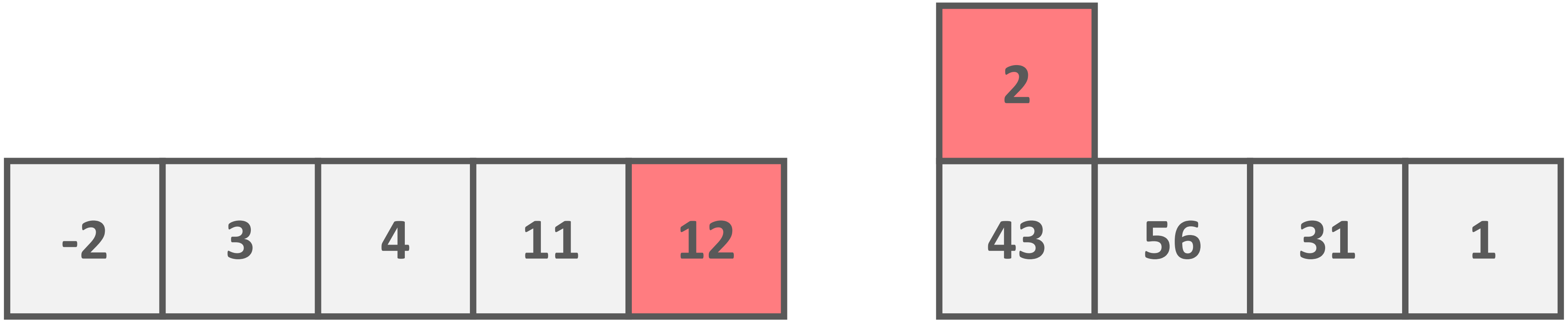
Insertion Sort



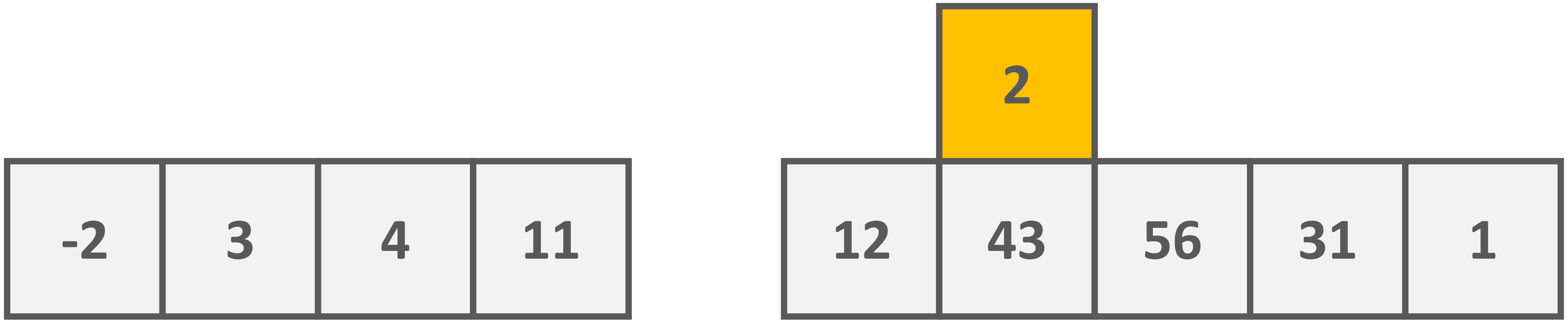
Insertion Sort



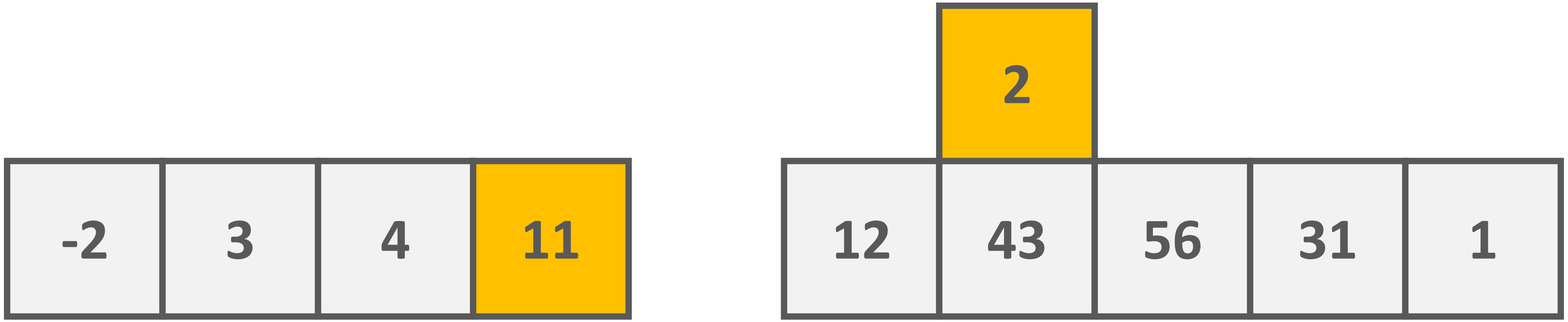
Insertion Sort



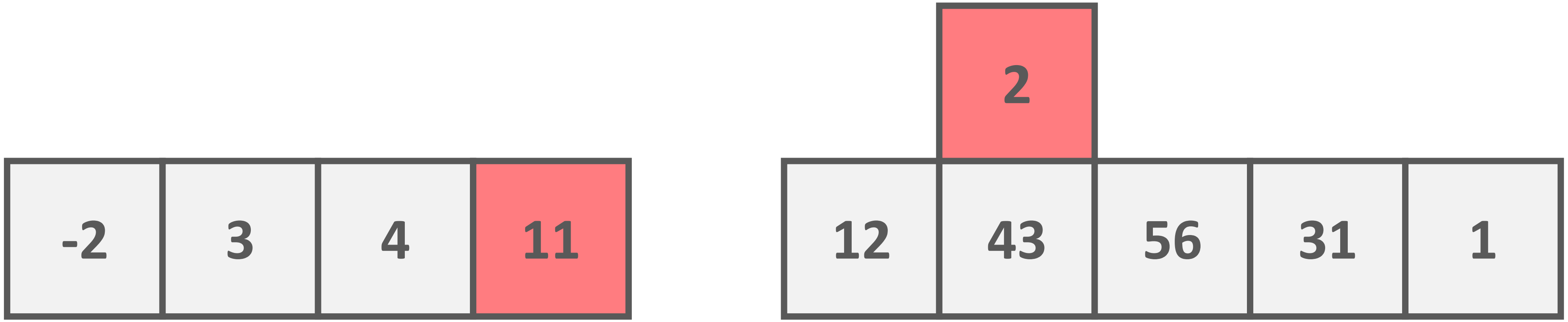
Insertion Sort



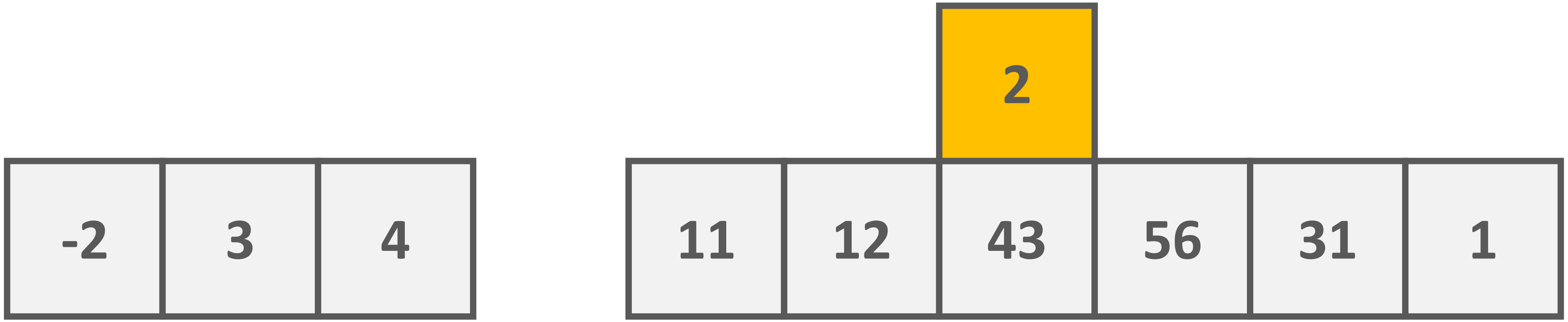
Insertion Sort



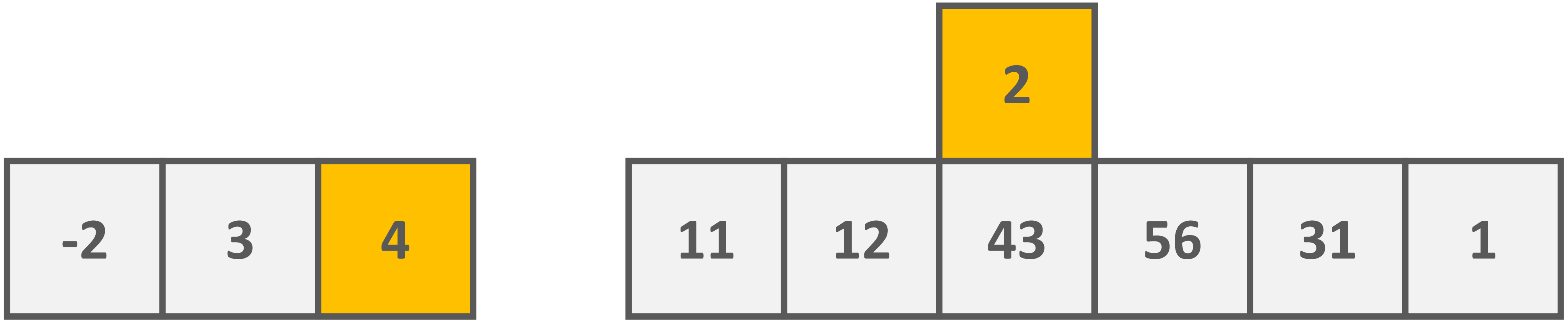
Insertion Sort



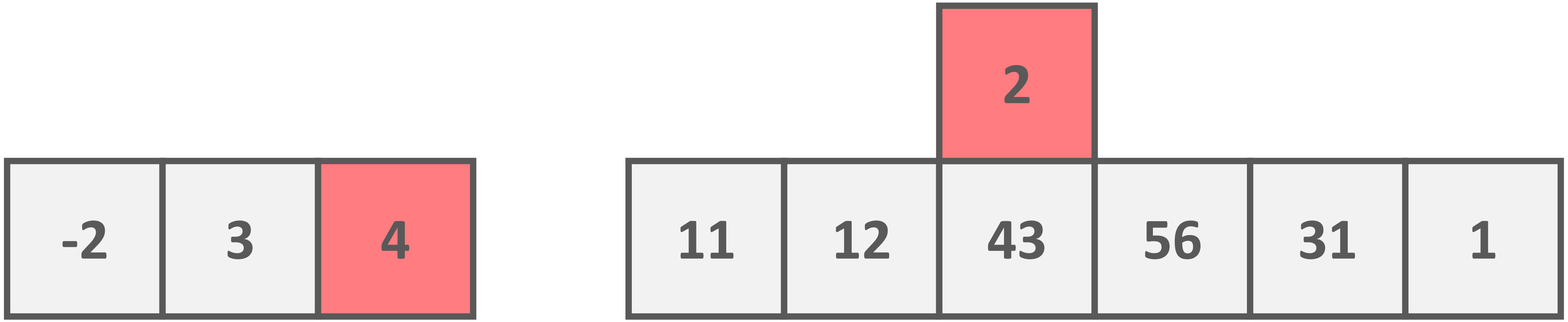
Insertion Sort



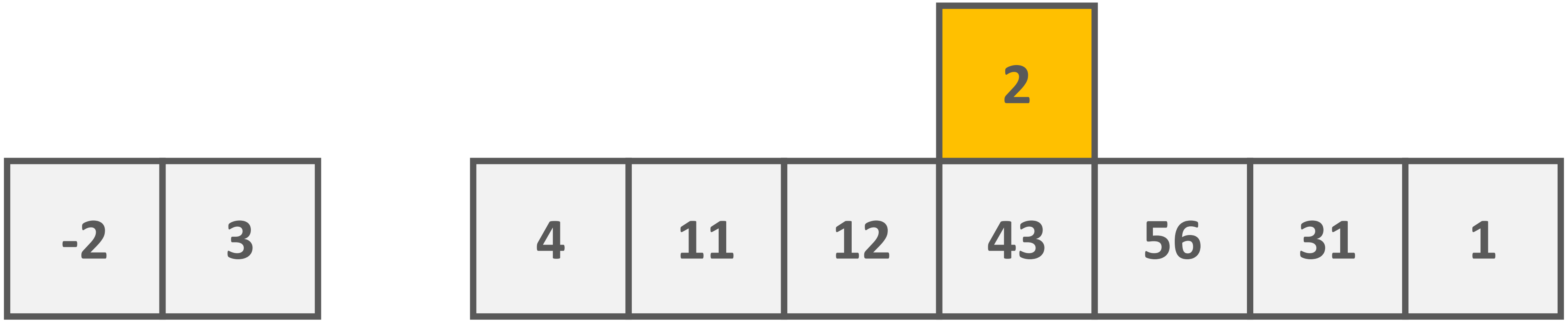
Insertion Sort



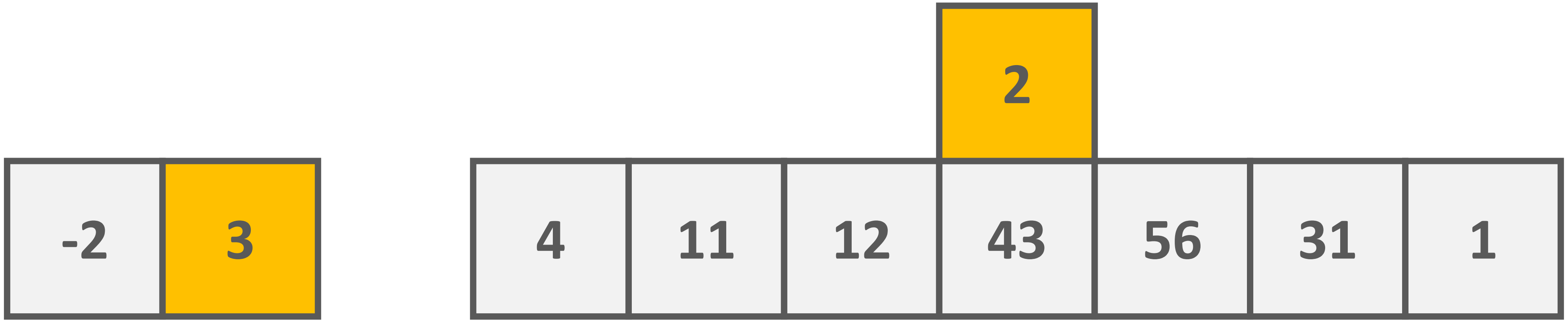
Insertion Sort



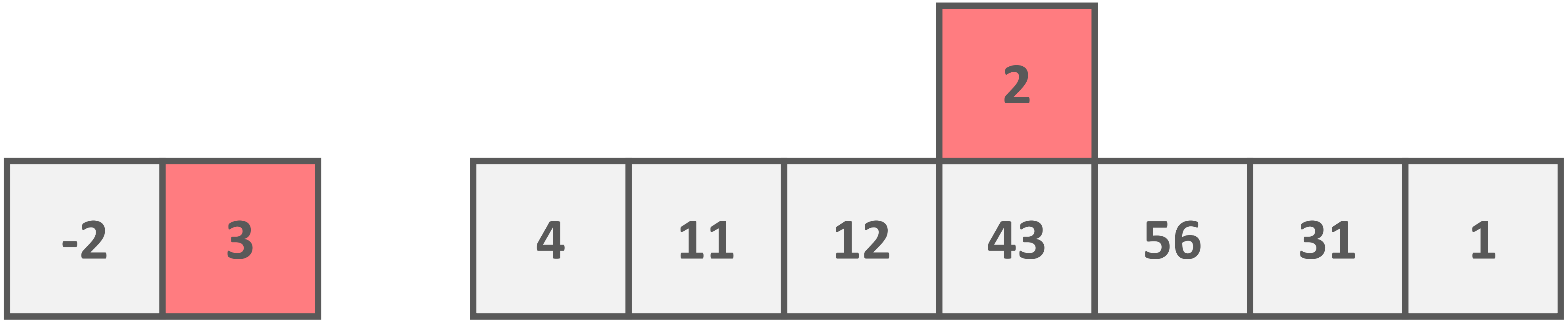
Insertion Sort



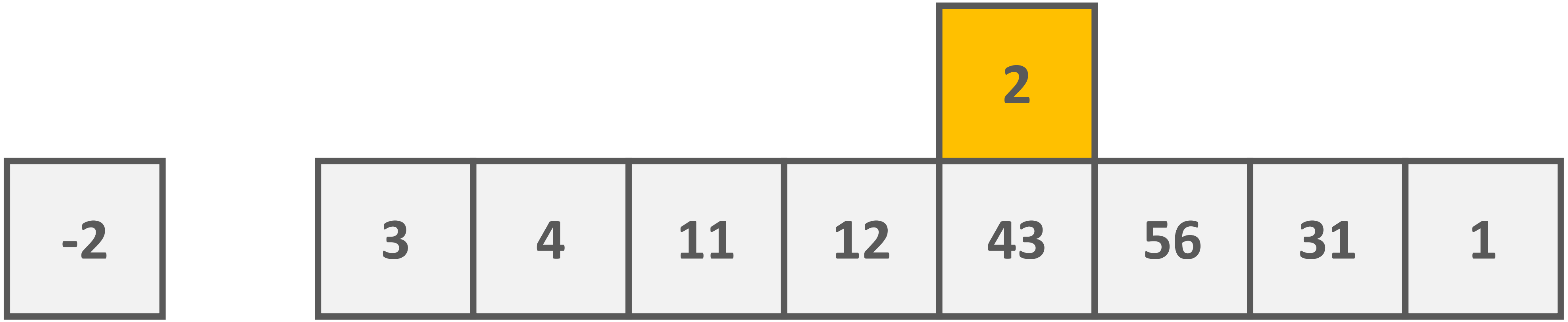
Insertion Sort



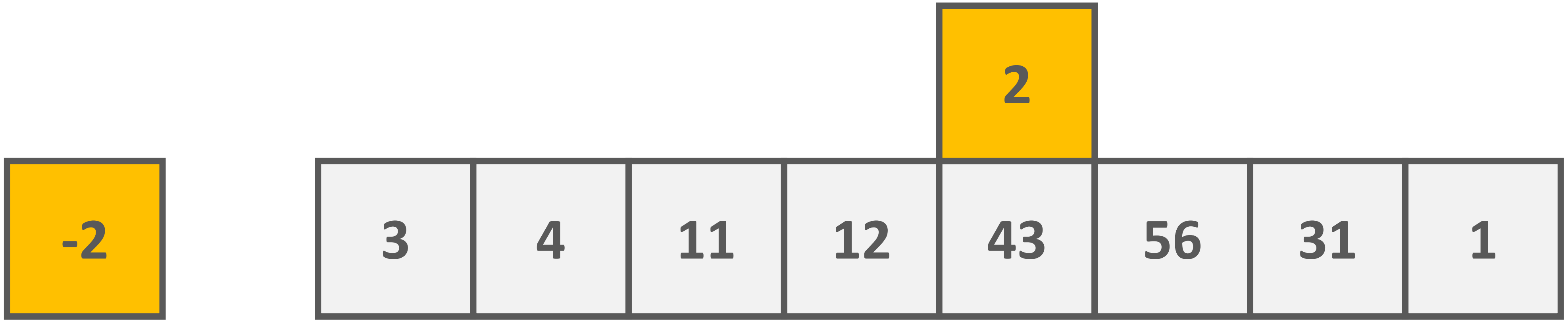
Insertion Sort



Insertion Sort



Insertion Sort



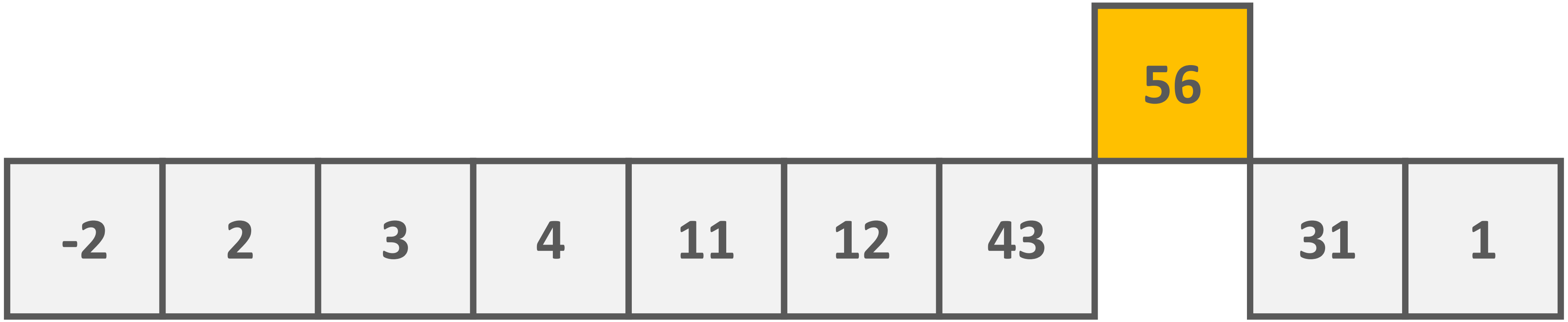
Insertion Sort

-2	2	3	4	11	12	43	56	31	1

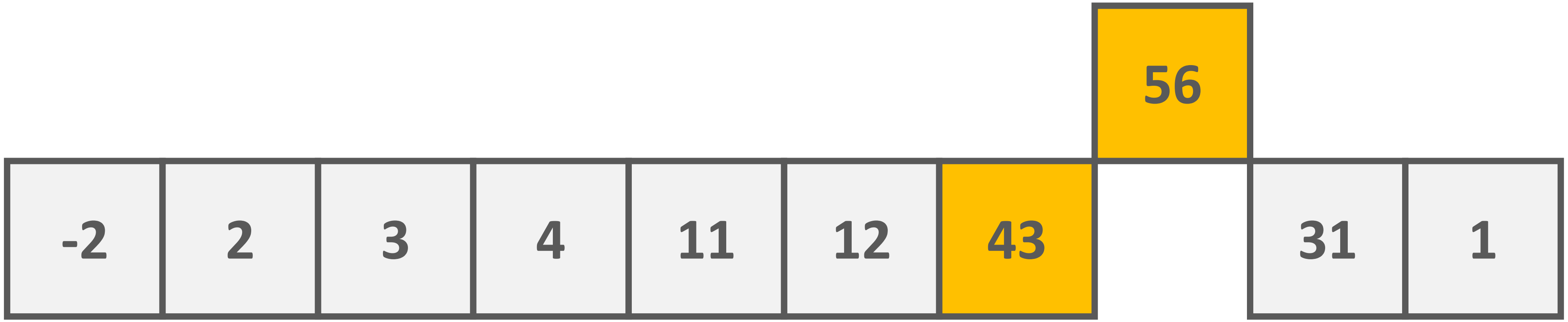
Insertion Sort

-2	2	3	4	11	12	43	56	31	1

Insertion Sort



Insertion Sort



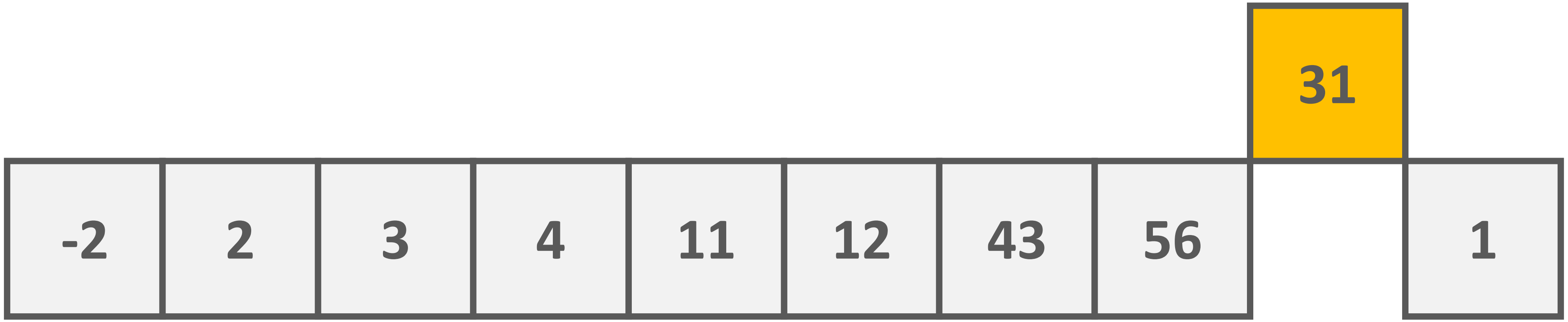
Insertion Sort

-2	2	3	4	11	12	43	56	31	1

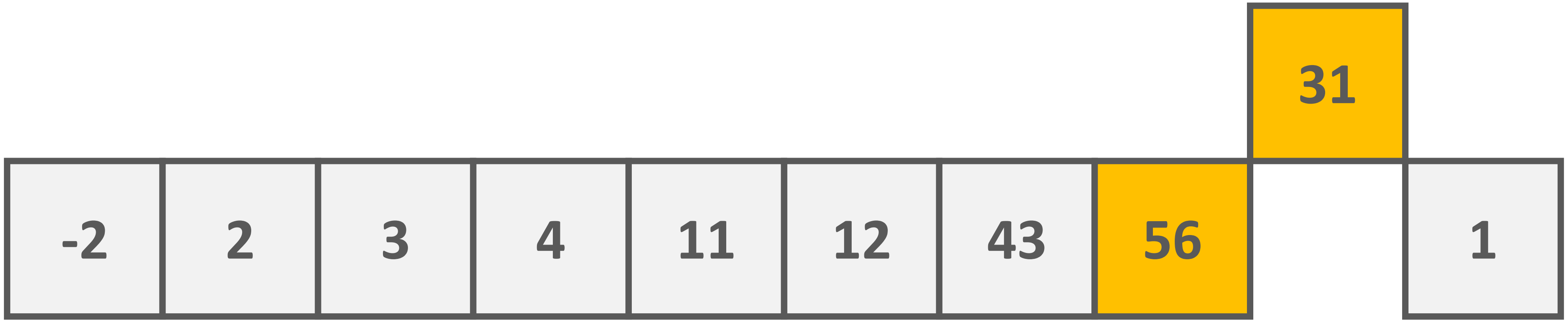
Insertion Sort

-2	2	3	4	11	12	43	56	31	1

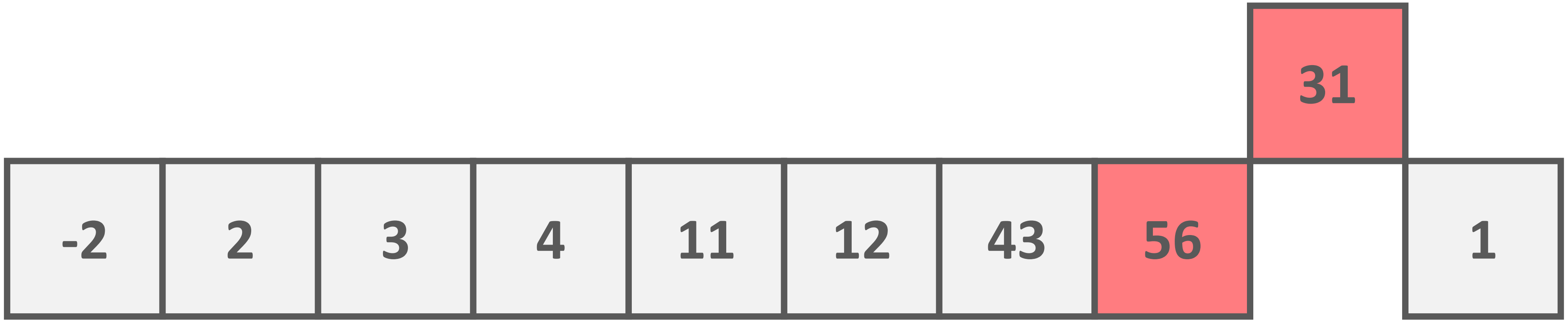
Insertion Sort



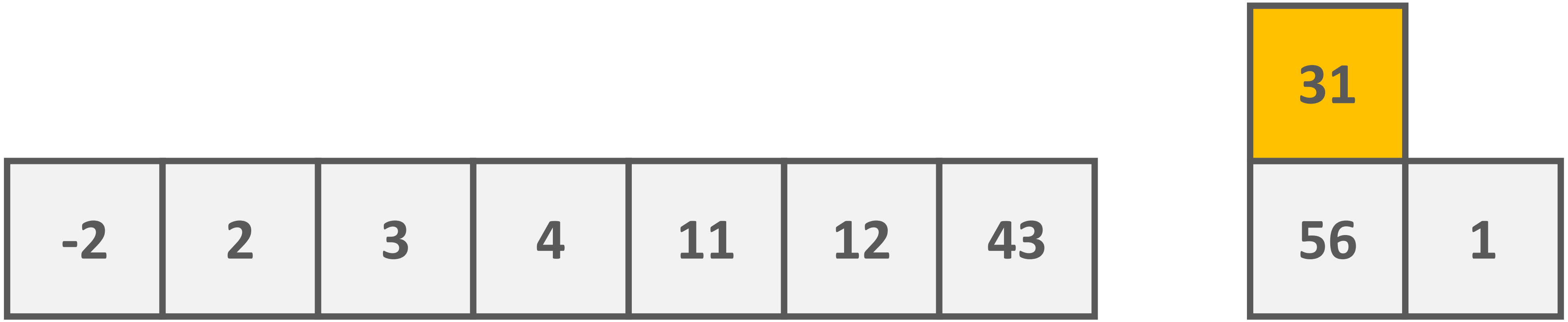
Insertion Sort



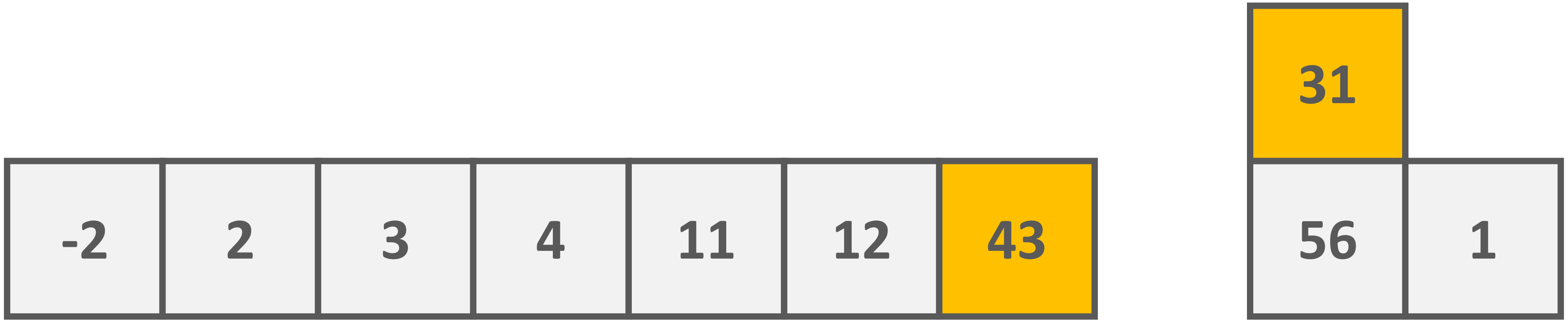
Insertion Sort



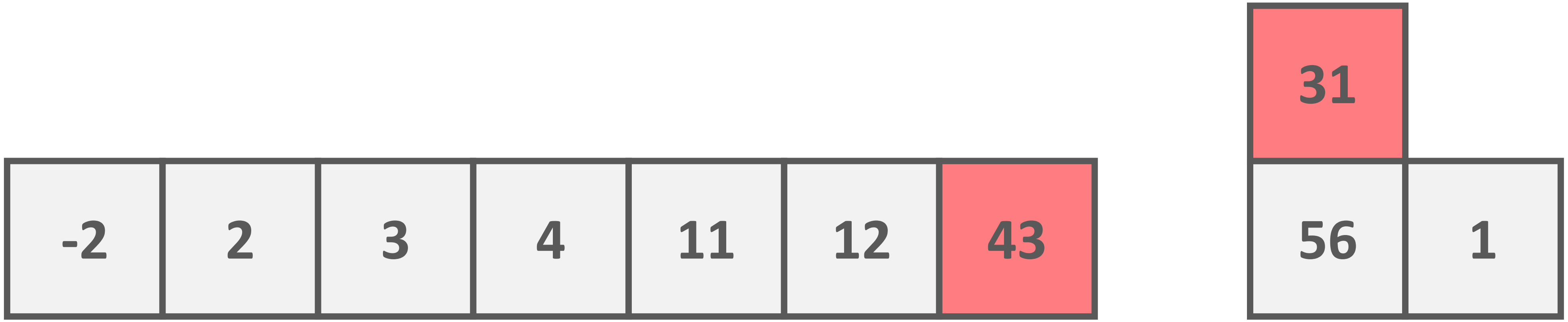
Insertion Sort



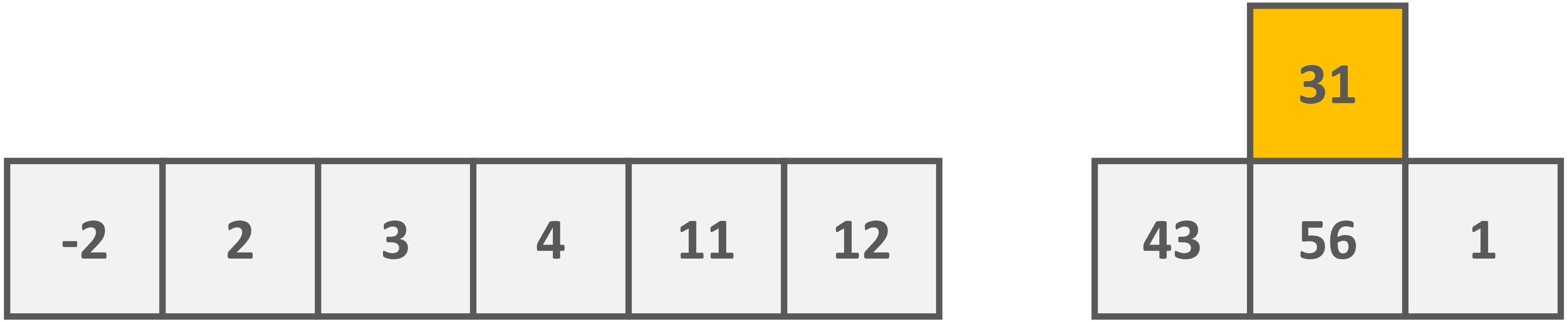
Insertion Sort



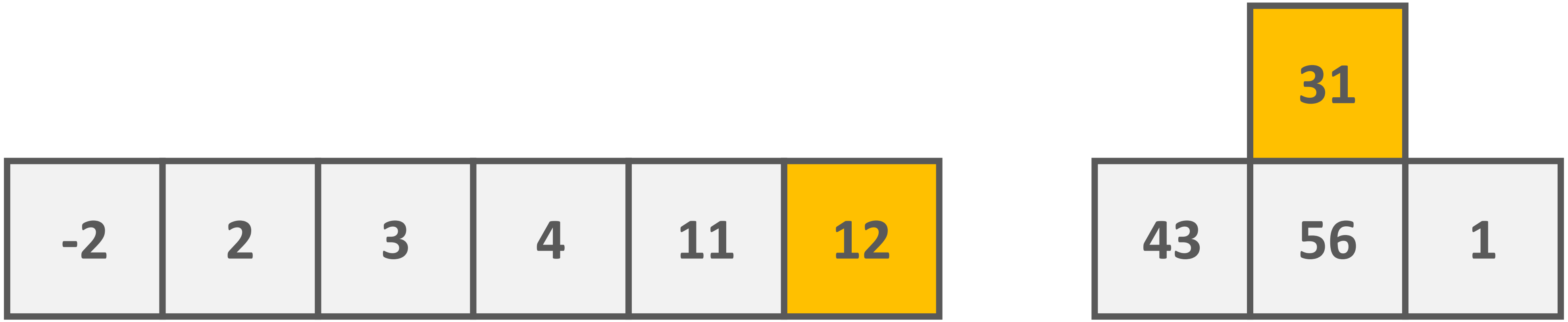
Insertion Sort



Insertion Sort



Insertion Sort



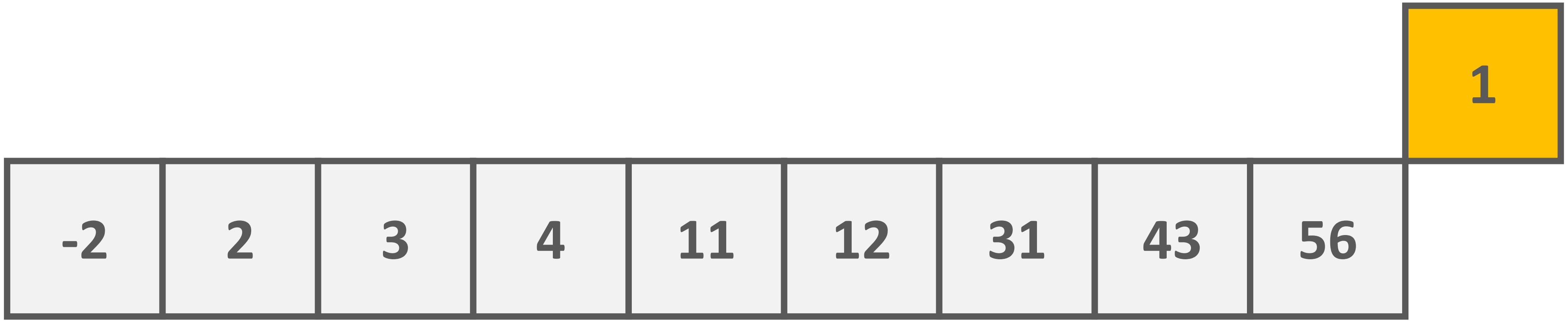
Insertion Sort

-2	2	3	4	11	12	31	43	56	1

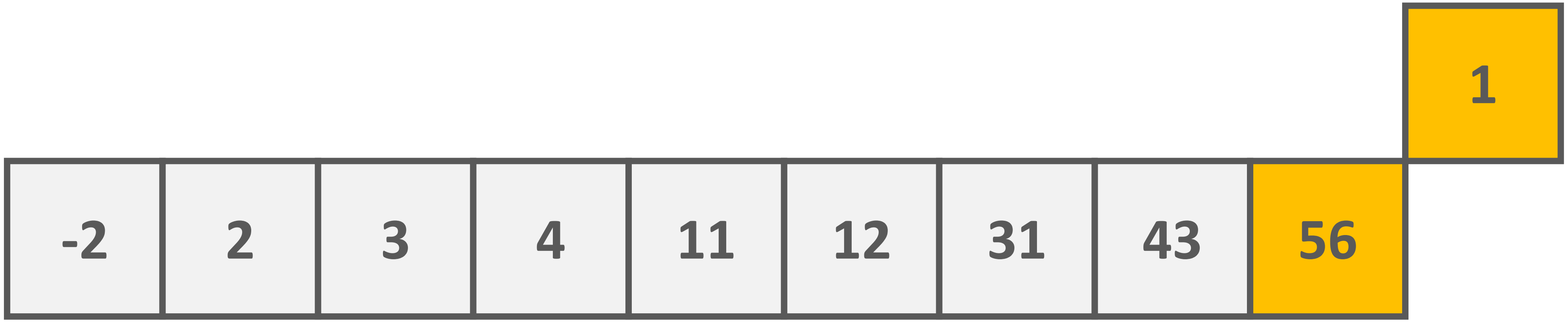
Insertion Sort

-2	2	3	4	11	12	31	43	56	1

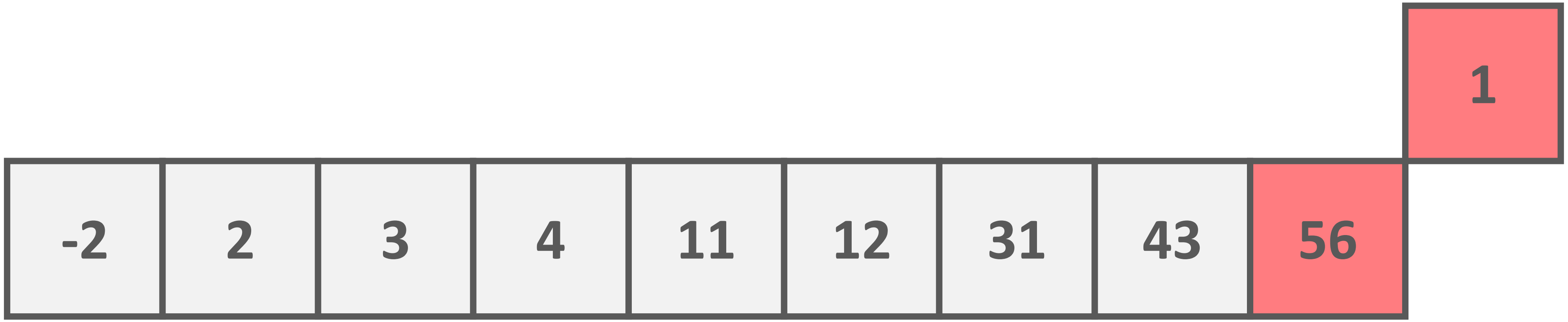
Insertion Sort



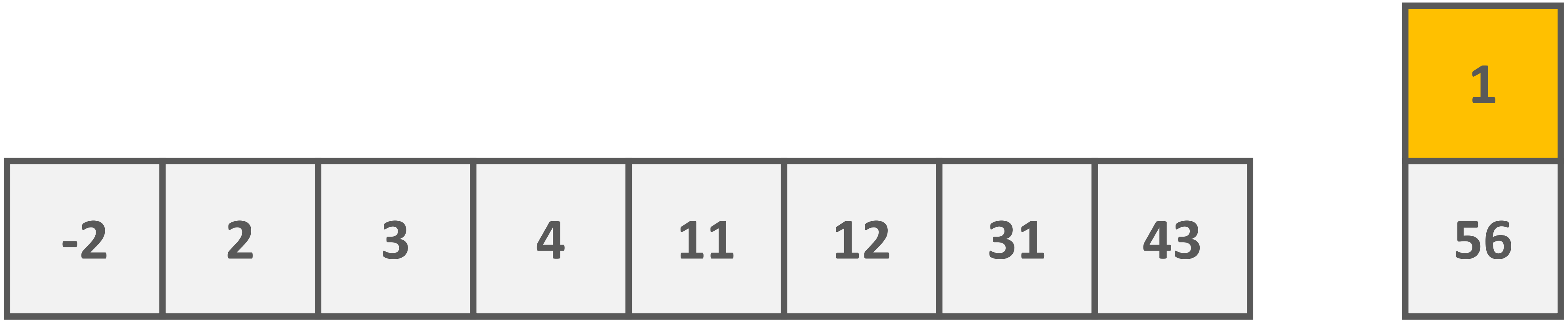
Insertion Sort



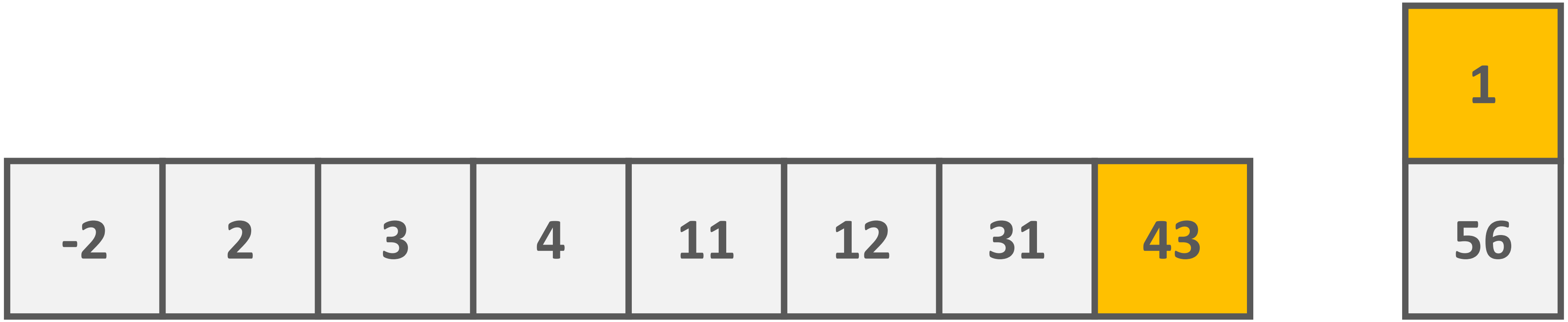
Insertion Sort



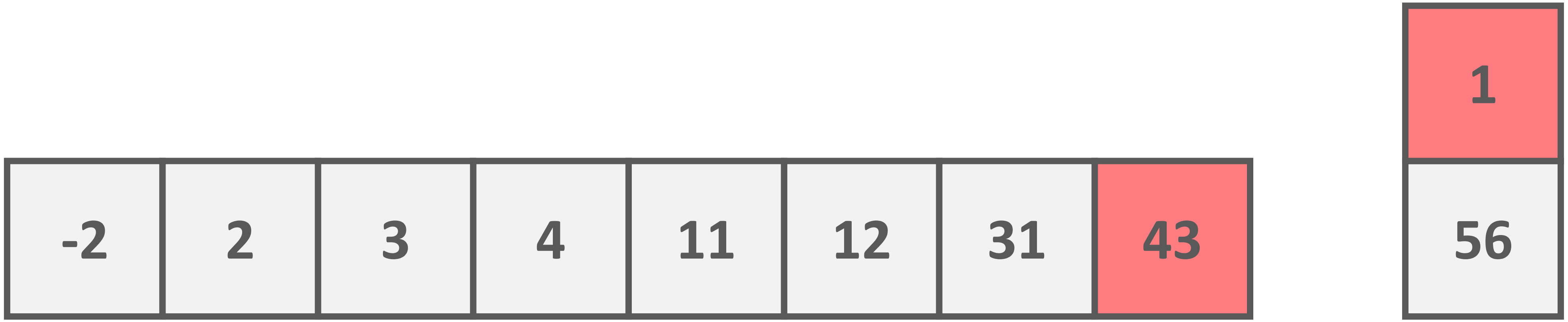
Insertion Sort



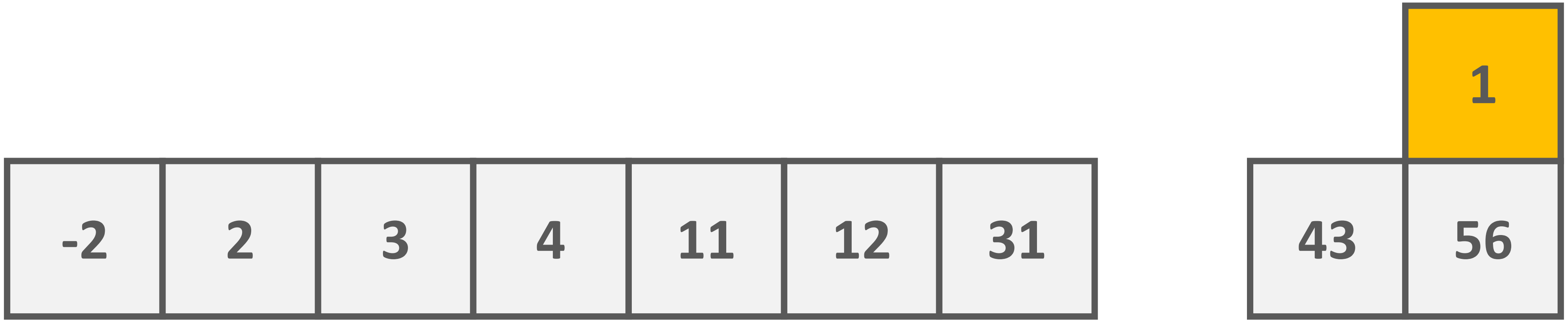
Insertion Sort



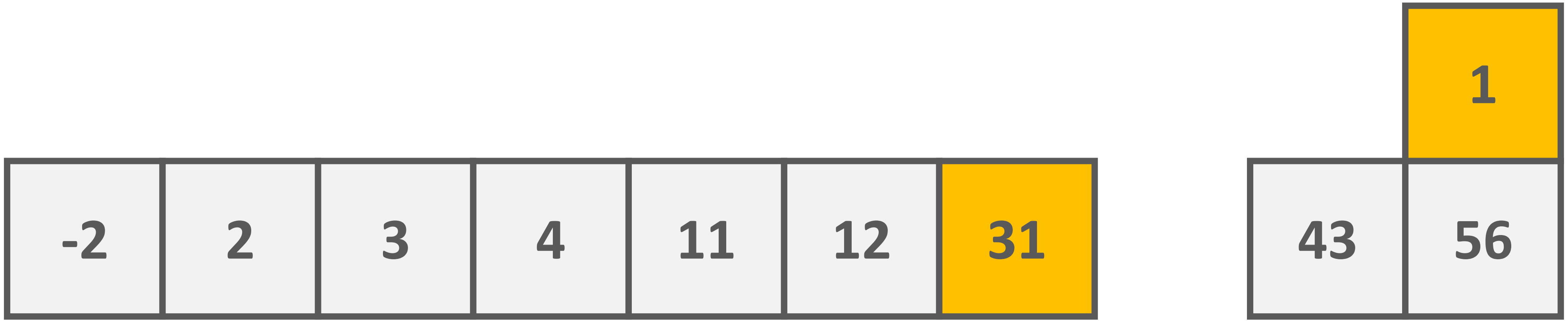
Insertion Sort



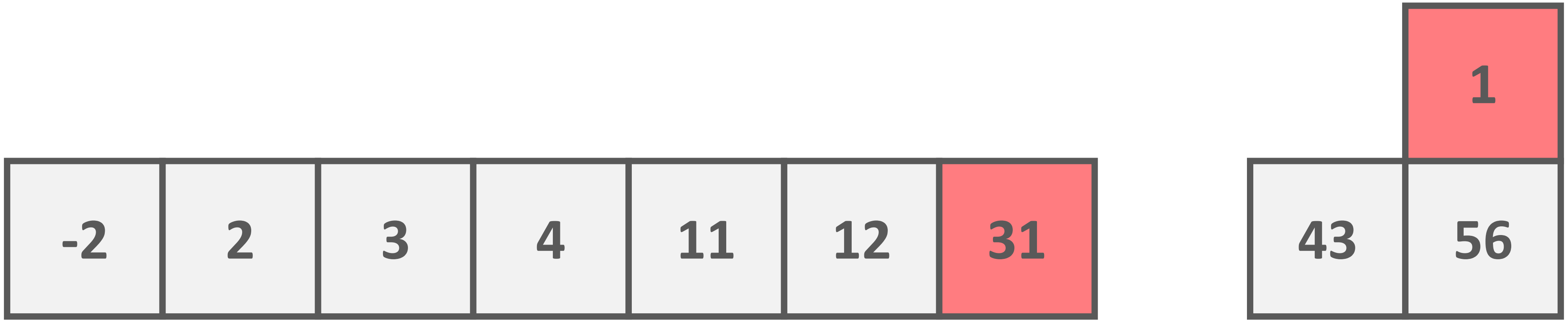
Insertion Sort



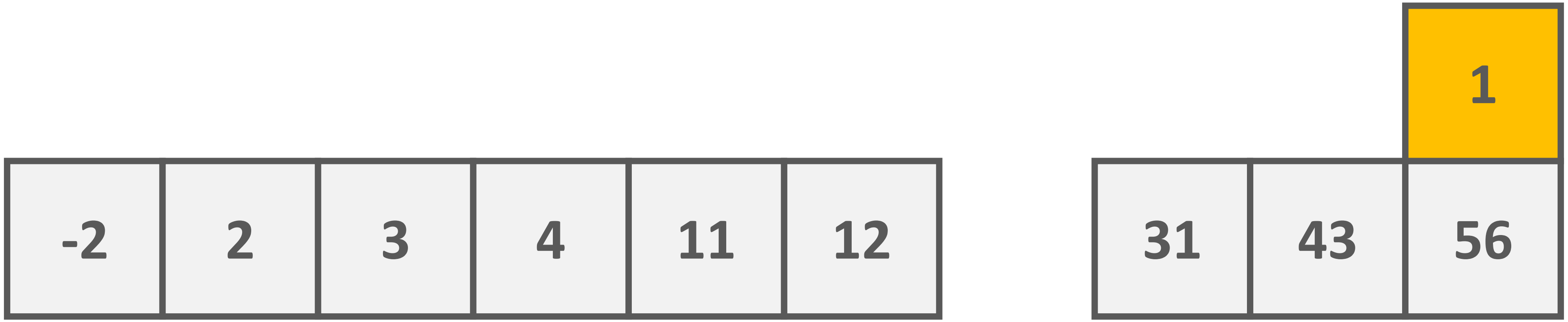
Insertion Sort



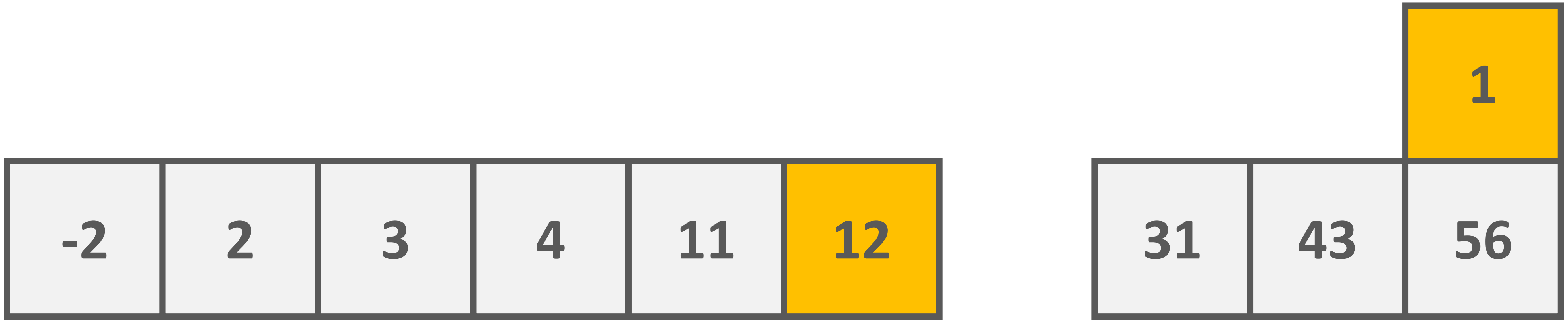
Insertion Sort



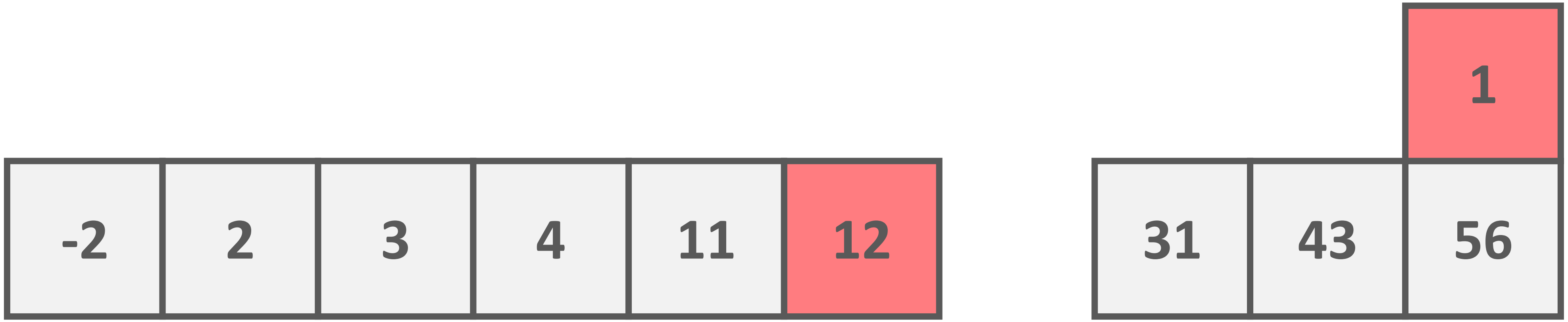
Insertion Sort



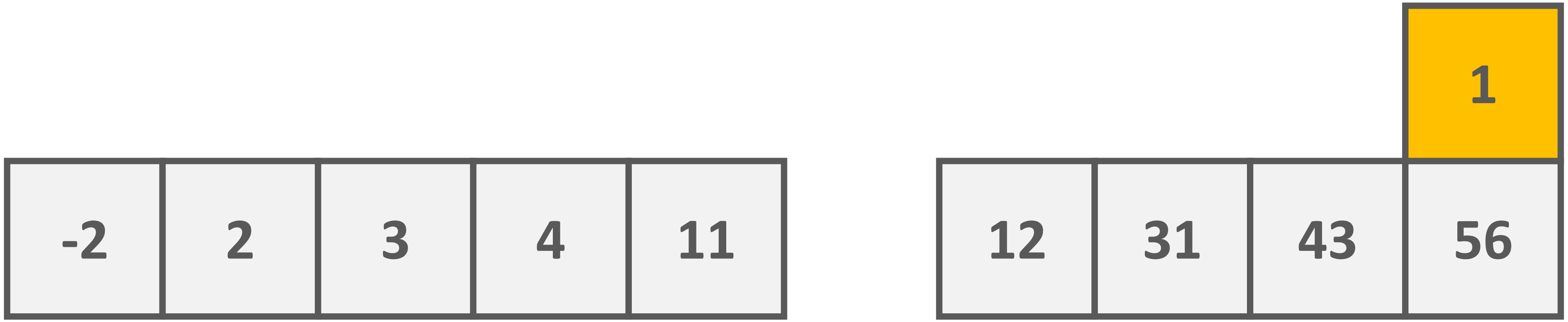
Insertion Sort



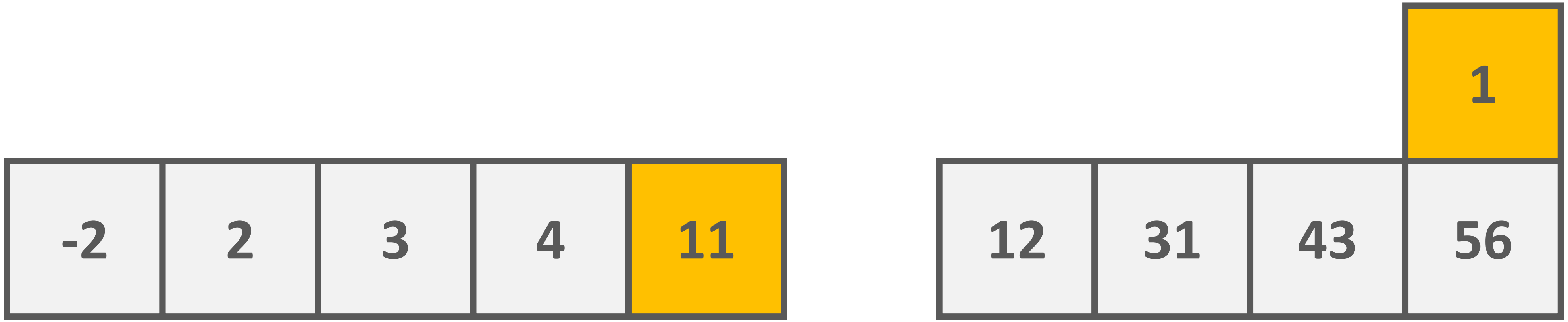
Insertion Sort



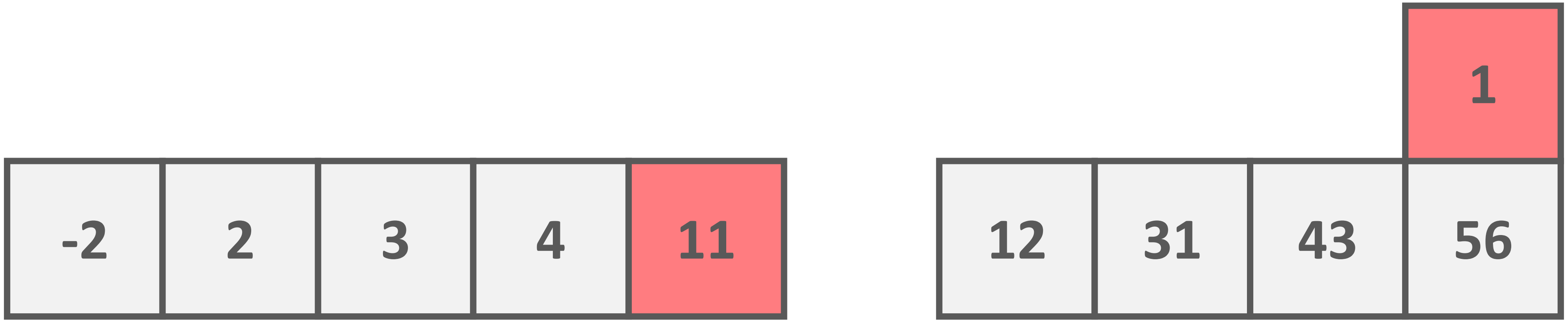
Insertion Sort



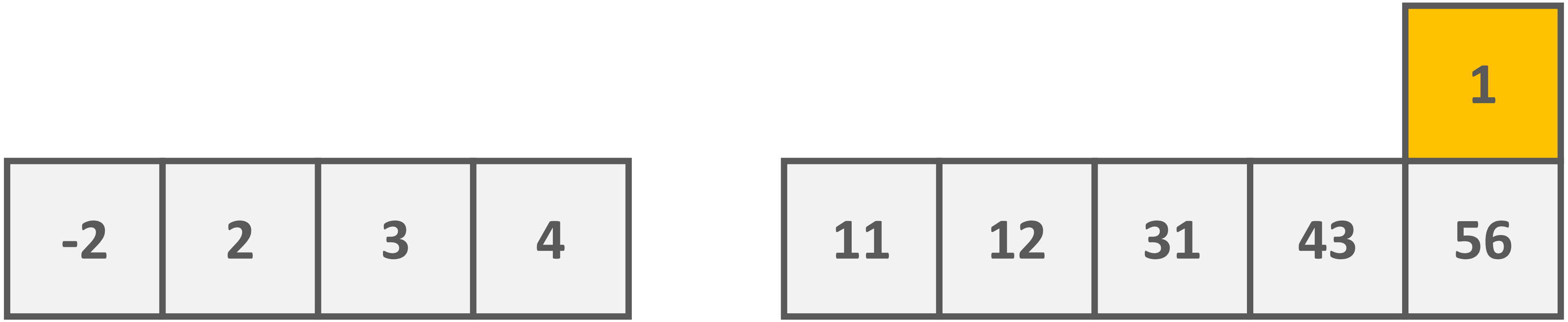
Insertion Sort



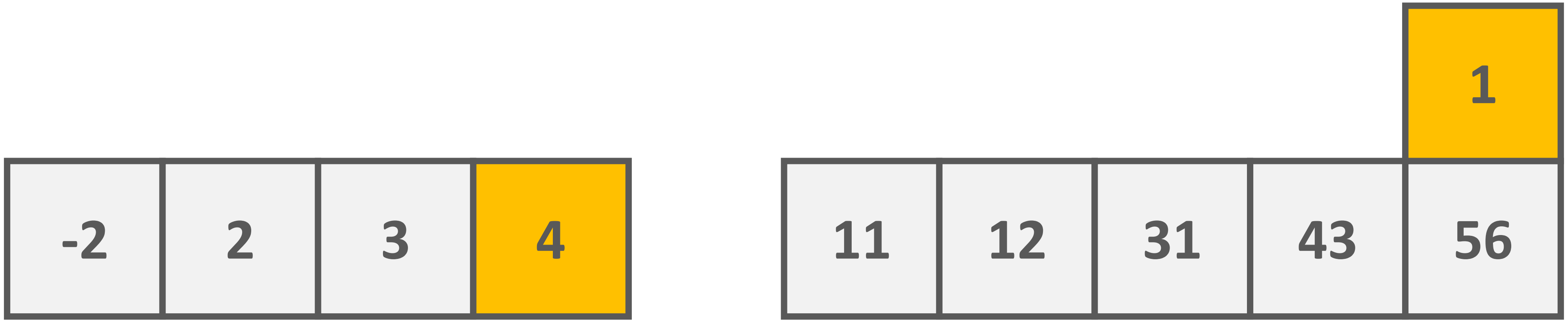
Insertion Sort



Insertion Sort

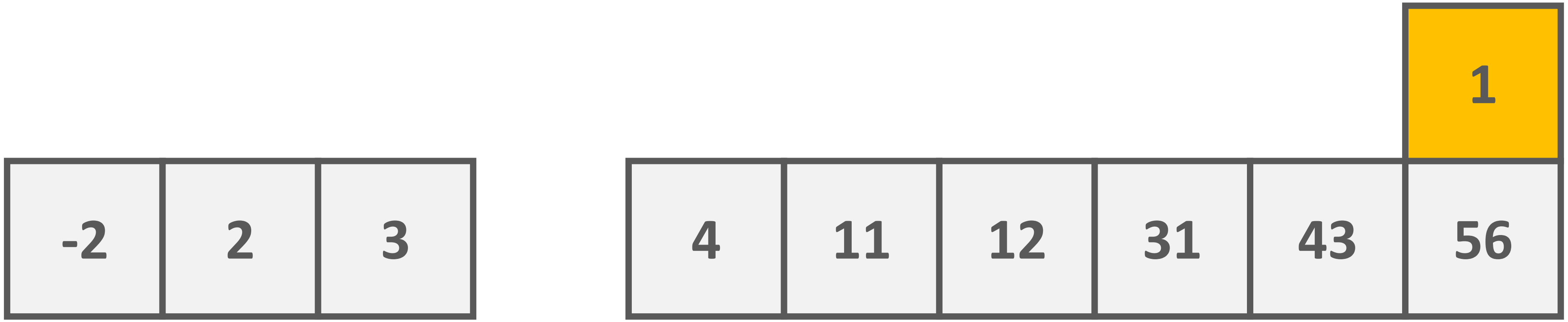


Insertion Sort

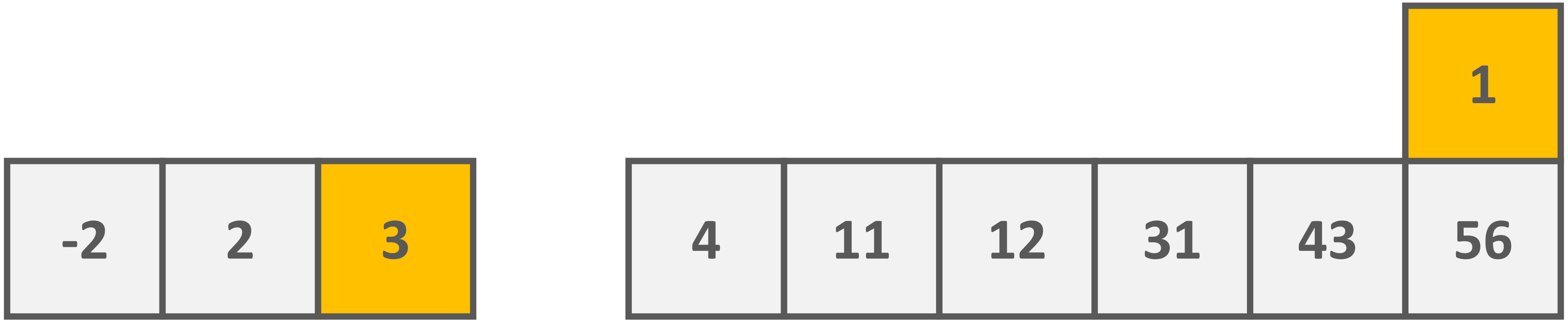


Insertion Sort

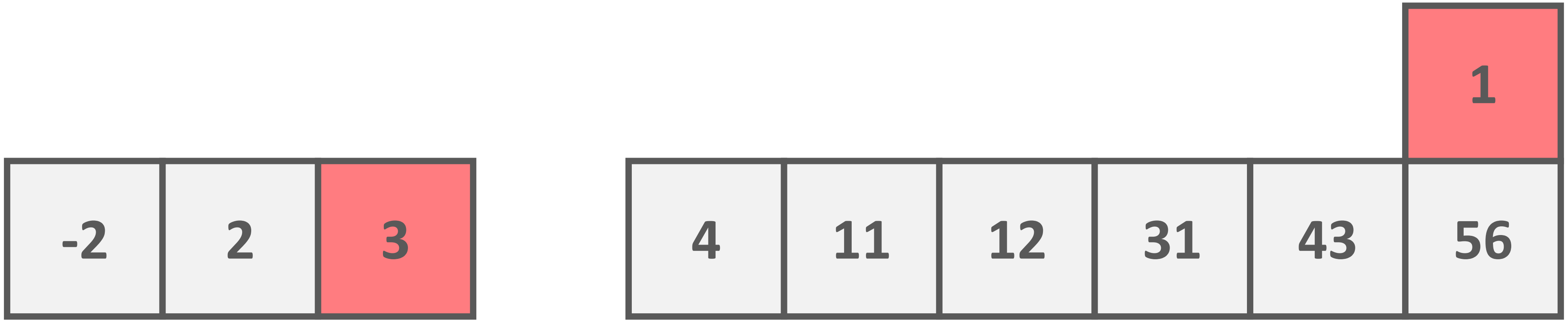
Insertion Sort



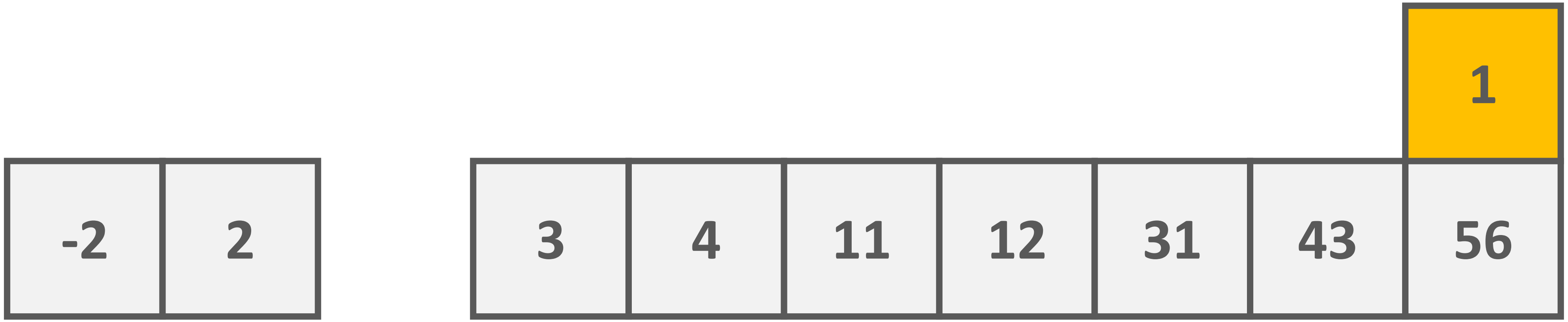
Insertion Sort



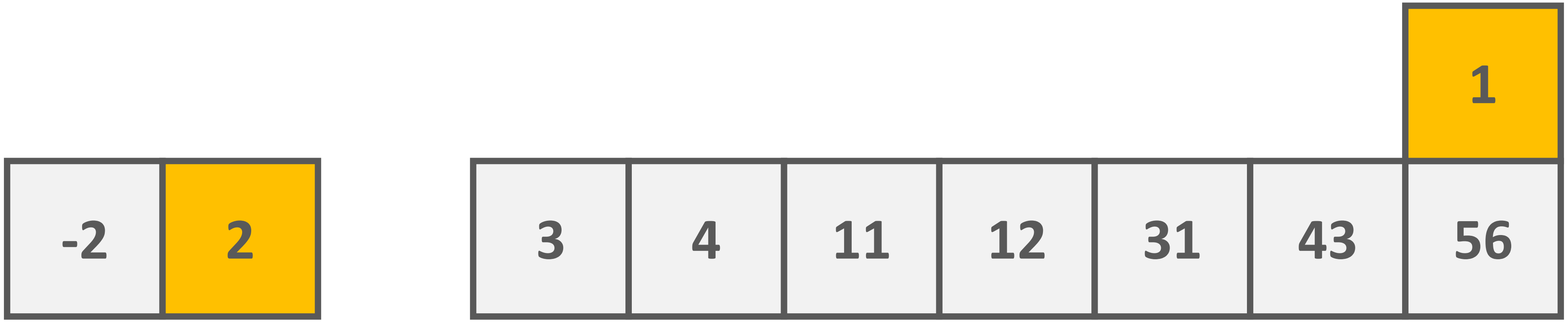
Insertion Sort



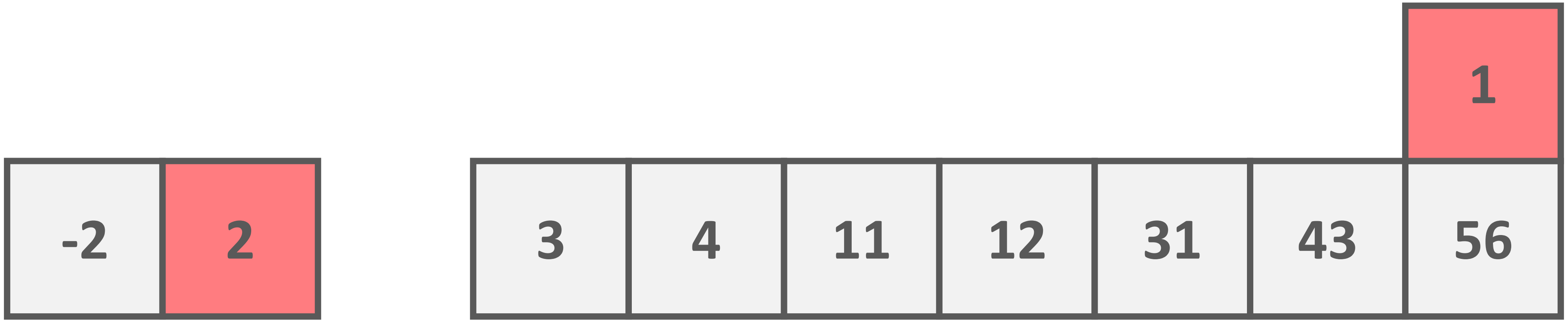
Insertion Sort



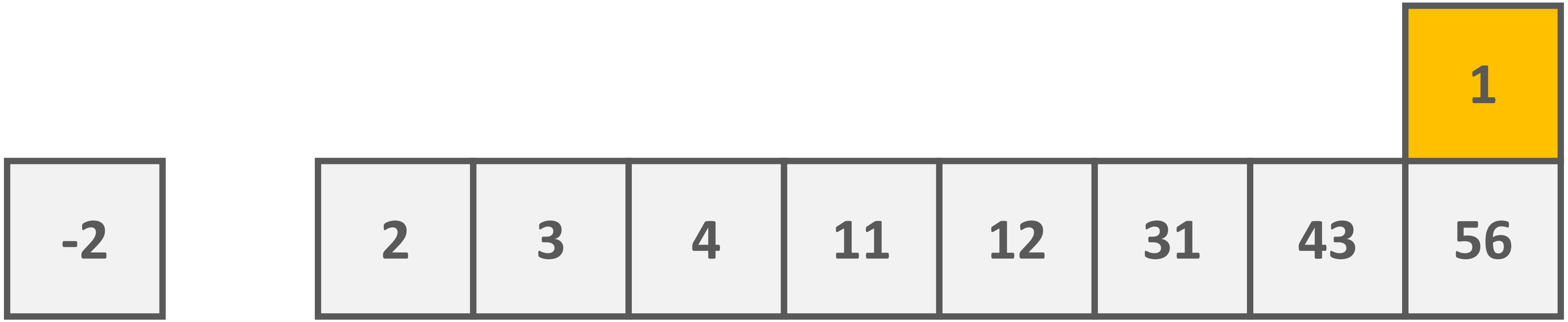
Insertion Sort



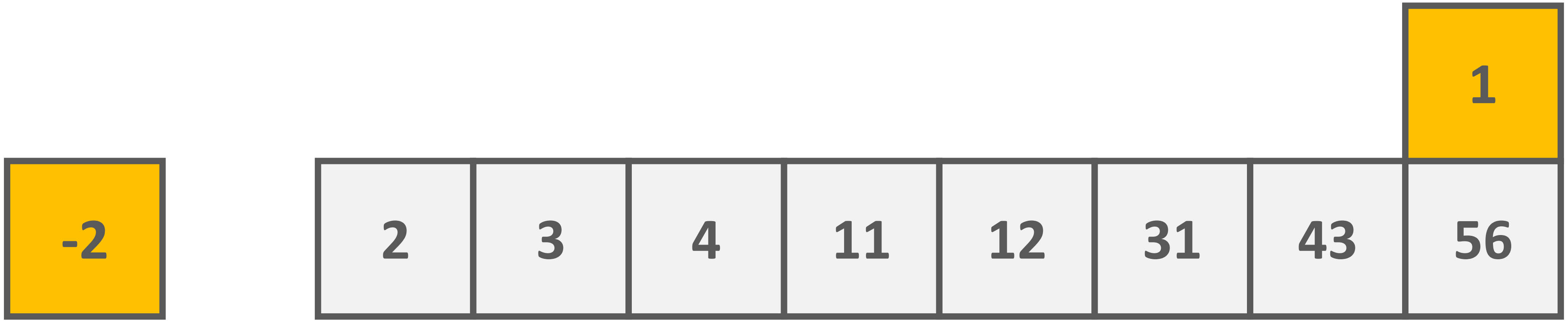
Insertion Sort



Insertion Sort



Insertion Sort



Insertion Sort

-2	1	2	3	4	11	12	31	43	56

Insertion Sort

-2	1	2	3	4	11	12	31	43	56

Shell Sort

(Algorithms and Data Structures)

Shell Sort

- it is the generalization of the insertion sort
- main problem of insertion sort is that sometimes we have to make **lots of shift operations** (swaps)
- this feature is not so good – thats why shell sort came to be as an enhanced insertion sort
- the method starts by sorting pairs of elements far apart from each other
- then progressively reducing the gap between elements to be compared
- starting with far apart elements can move some out-of-place elements into position faster than a simple nearest neighbor exchange

Shell Sort

- shell sort is heavily dependent on the gap sequence it uses
- consider every **k**-th element in the array
- such a subarray is said to be **k**-sorted
- we use insertion sort as a subprocedure – the only difference is that we start sorting items far away from each other
- this rearrangement allows elements to move long distances in the original list reducing large amounts of disorder quickly

Shell Sort

- shell sort is unstable – it changes the relative order of elements with equal value
- because it relies heavily on insertion sort – it is also an **adaptive algorithm** so runs faster on partially sorted input
- not so popular algorithm nowadays

Shell Sort

12	4	-2	11	1	65	-5	17	31	10	43	2	56	29	1

Shell Sort

12	4	-2	11	1	65	-5	17	31	10	43	2	56	29	1

Shell Sort

1	4	-2	11	12	65	-5	17	31	10	43	2	56	29	1

Shell Sort

1	4	-2	11	12	65	-5	17	31	10	43	2	56	29	1

Shell Sort

1	4	-2	11	12	65	-5	17	31	10	43	2	56	29	1

Shell Sort

1	4	-2	11	12	10	-5	17	31	29	43	2	56	65	1

Shell Sort

1	4	-2	11	12	10	-5	17	31	29	43	2	56	65	1

Shell Sort

1	4	-2	11	12	10	-5	17	31	29	43	2	56	65	1

Shell Sort

1	4	-2	11	12	10	-5	17	31	29	1	2	56	65	43

Shell Sort

1	4	-2	11	12	10	-5	17	31	29	1	2	56	65	43

Shell Sort

1	4	-2	11	12	10	-5	17	31	29	1	2	56	65	43

Shell Sort

1	4	-2	2	12	10	-5	11	31	29	1	17	56	65	43

Shell Sort

1	4	-2	2	12	10	-5	11	31	29	1	17	56	65	43

Shell Sort

1	4	-2	2	12	10	-5	11	31	29	1	17	56	65	43

Shell Sort

-5	4	-2	1	12	10	2	11	31	29	1	17	56	65	43

Shell Sort

1	4	-2	2	12	10	-5	11	31	29	1	17	56	65	43

Shell Sort

1	4	-2	2	12	10	-5	11	31	29	1	17	56	65	43

Shell Sort

1	1	-2	2	4	10	-5	11	31	29	12	17	56	65	43

Shell Sort

1	1	-2	2	4	10	-5	11	31	29	12	17	56	65	43

Shell Sort

1	1	-2	2	4	10	-5	11	31	29	12	17	56	65	43

Shell Sort

1	1	-2	2	4	10	-5	11	17	29	12	31	56	65	43

Shell Sort

1	1	-2	2	4	10	-5	11	17	29	12	31	56	65	43

Shell Sort

1	1	-2	2	4	10	-5	11	17	29	12	31	56	65	43

Shell Sort

-5	1	-2	2	1	10	4	11	12	29	17	31	43	65	56

Shell Sort

-5	1	-2	2	1	10	4	11	12	29	17	31	43	65	56

Shell Sort

-5	1	-2	2	1	10	4	11	12	29	17	31	43	65	56

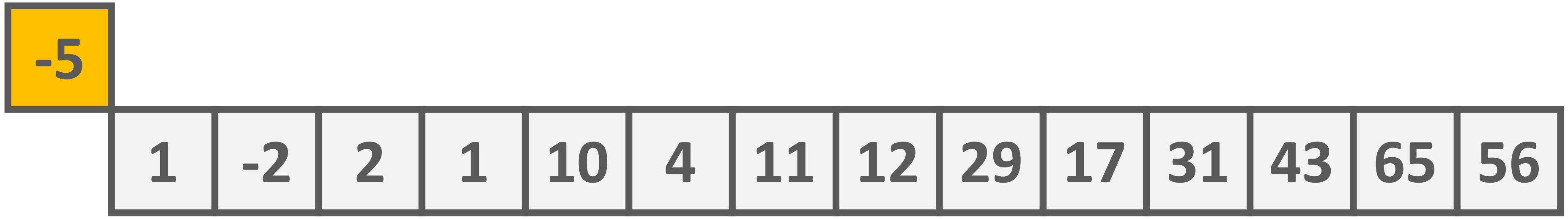
Shell Sort

-5	1	-2	2	1	10	4	11	12	29	17	31	43	65	56

Shell Sort

-5	1	-2	2	1	10	4	11	12	29	17	31	43	65	56

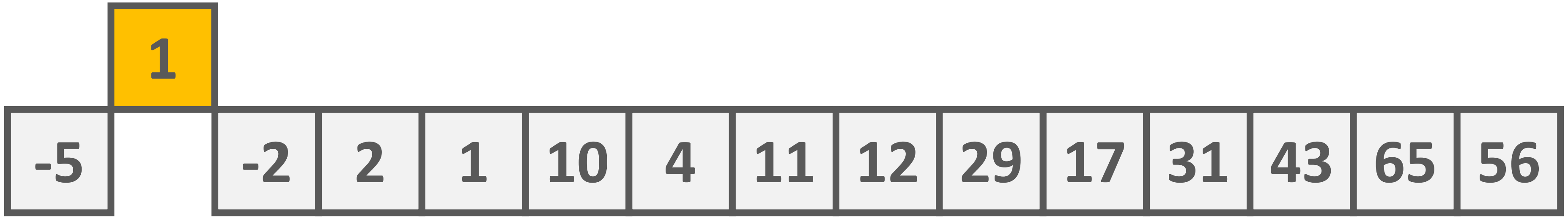
Shell Sort



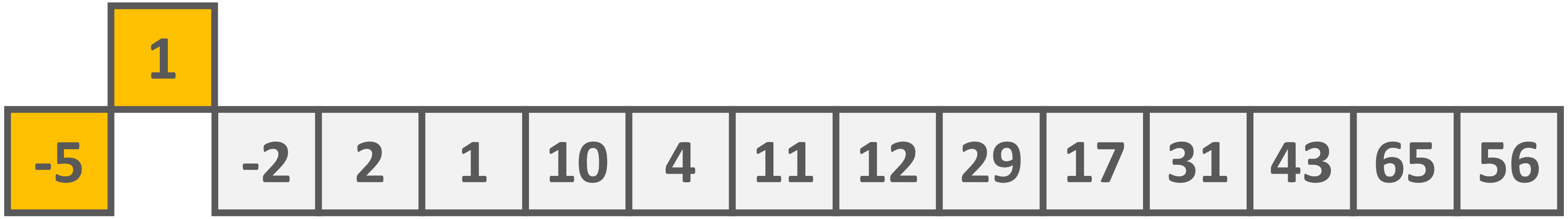
Shell Sort

-5	1	-2	2	1	10	4	11	12	29	17	31	43	65	56

Shell Sort



Shell Sort



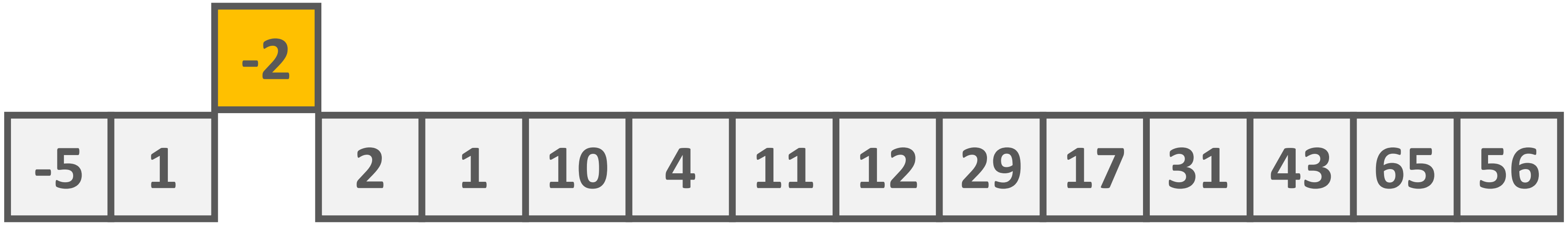
Shell Sort

-5	1	-2	2	1	10	4	11	12	29	17	31	43	65	56

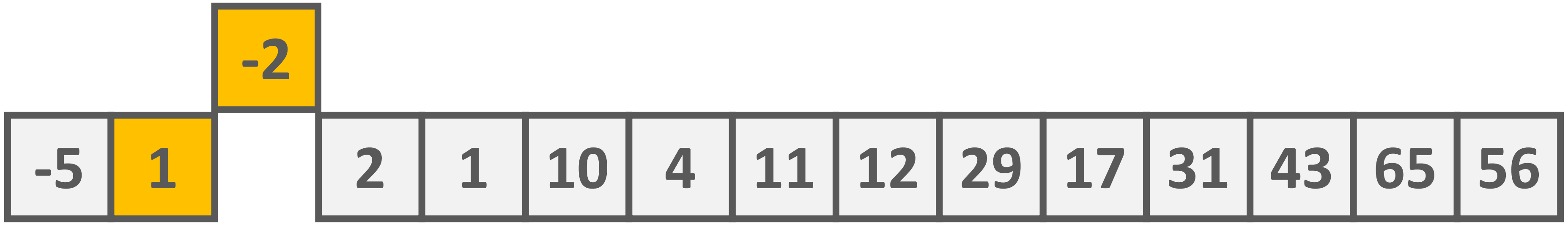
Shell Sort

-5	1	-2	2	1	10	4	11	12	29	17	31	43	65	56

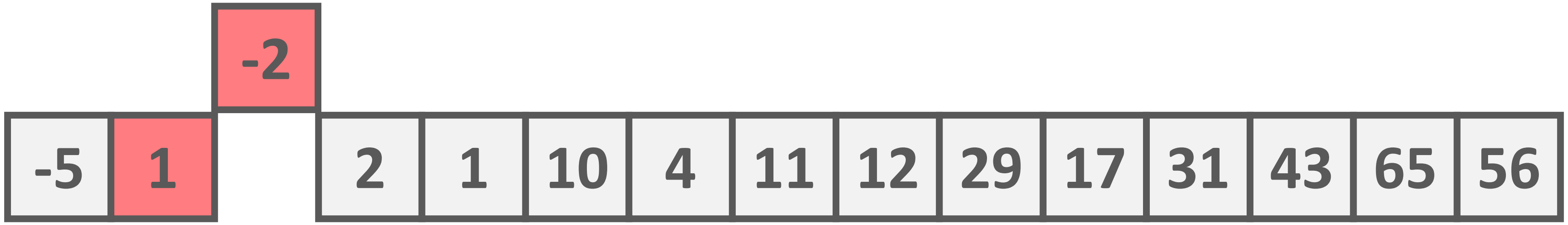
Shell Sort



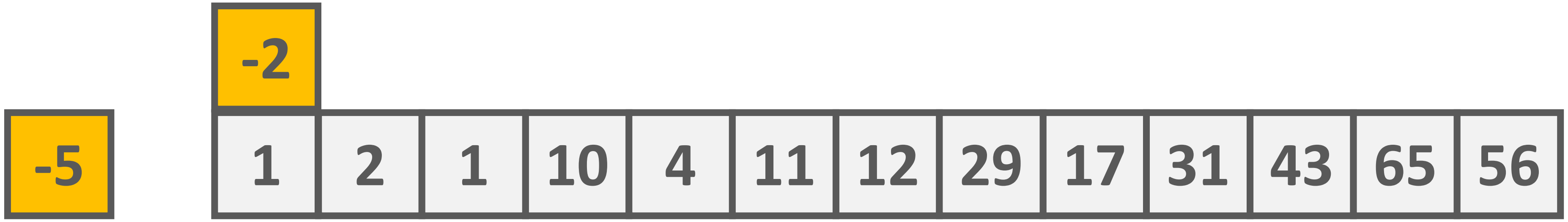
Shell Sort



Shell Sort



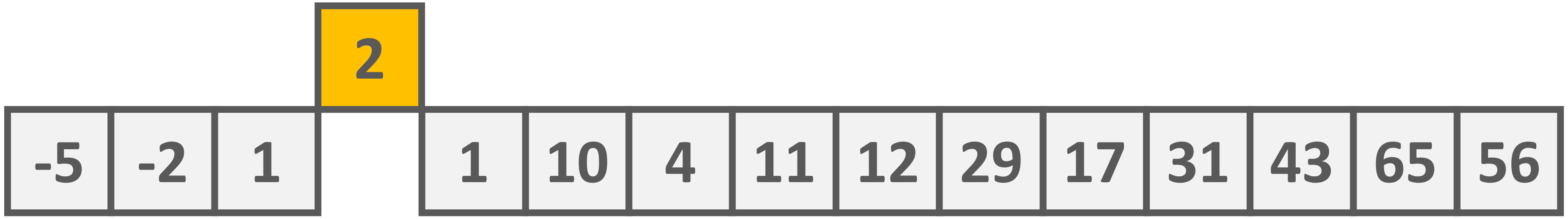
Shell Sort



Shell Sort

-5	-2	1	2	1	10	4	11	12	29	17	31	43	65	56

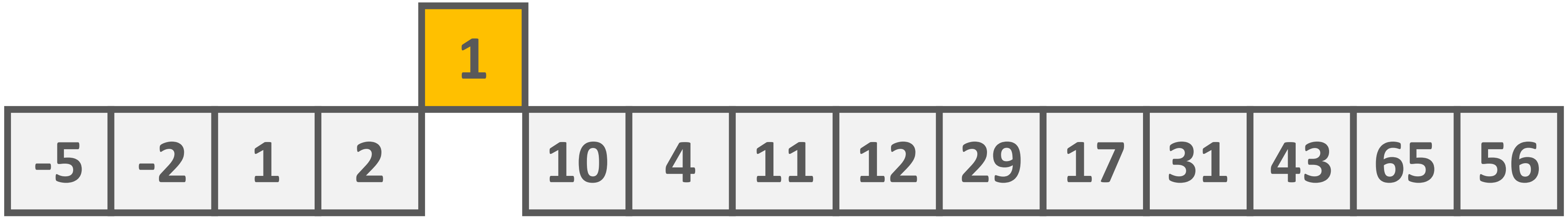
Shell Sort



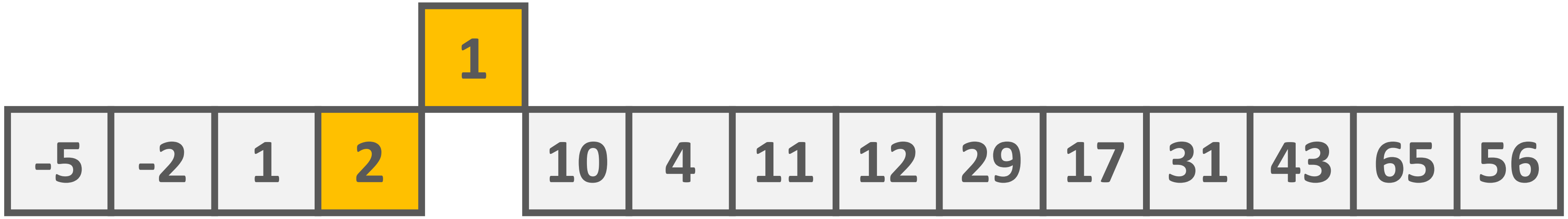
Shell Sort

-5	-2	1	2	1	10	4	11	12	29	17	31	43	65	56

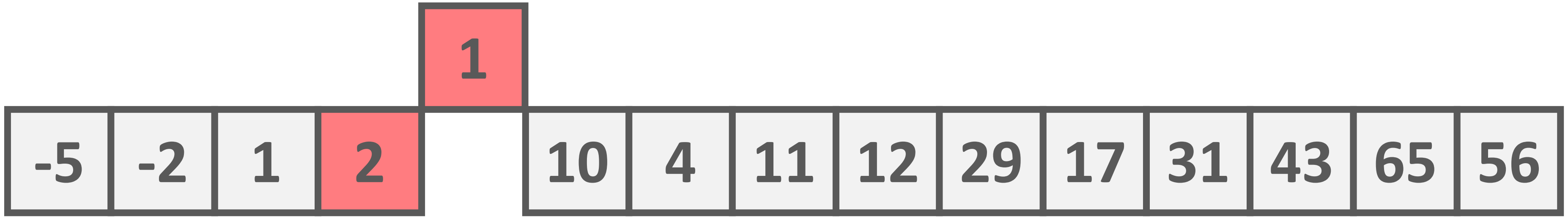
Shell Sort



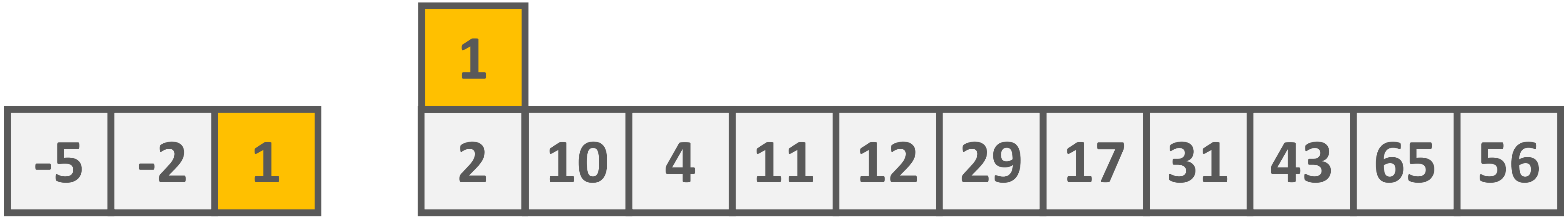
Shell Sort



Shell Sort



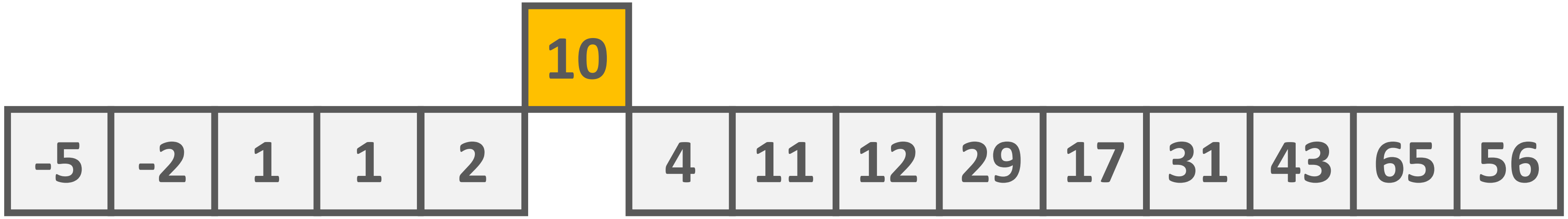
Shell Sort



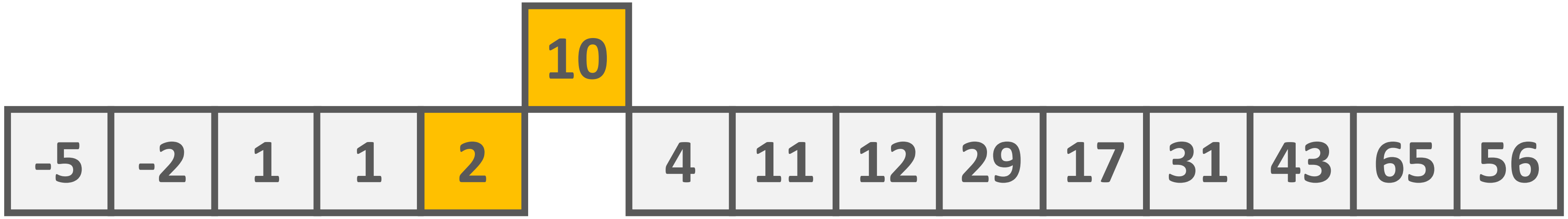
Shell Sort

-5	-2	1	1	2	10	4	11	12	29	17	31	43	65	56

Shell Sort



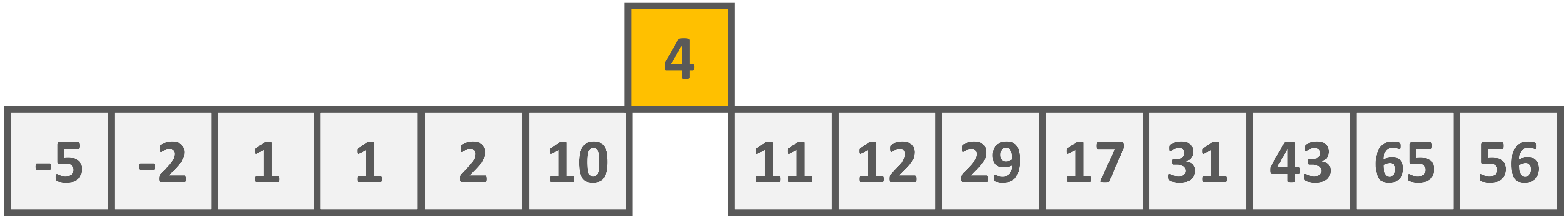
Shell Sort



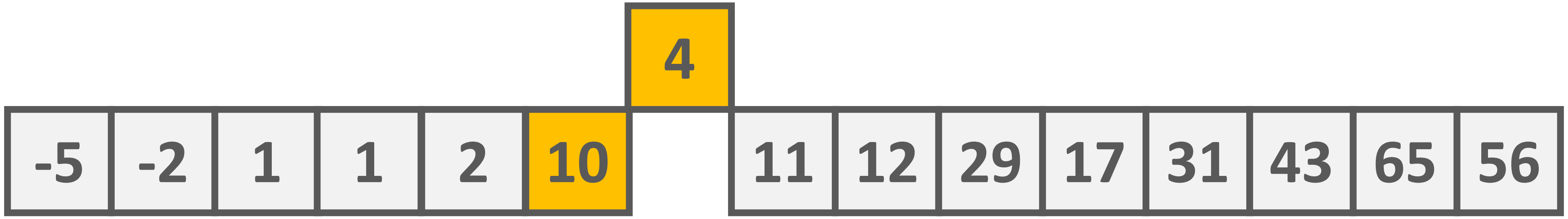
Shell Sort

-5	-2	1	1	2	10	4	11	12	29	17	31	43	65	56

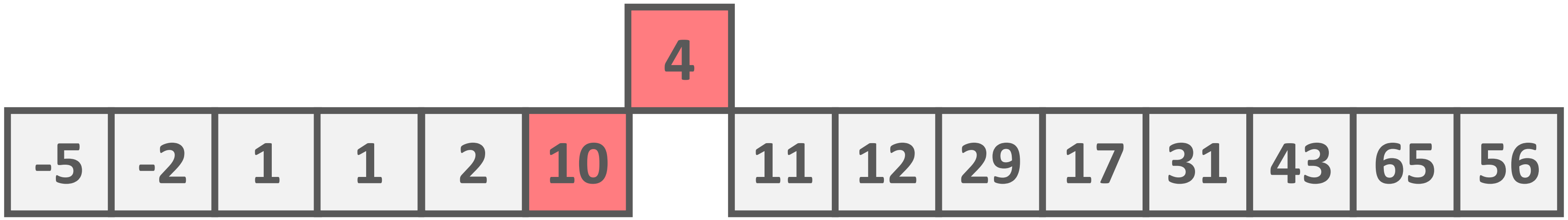
Shell Sort



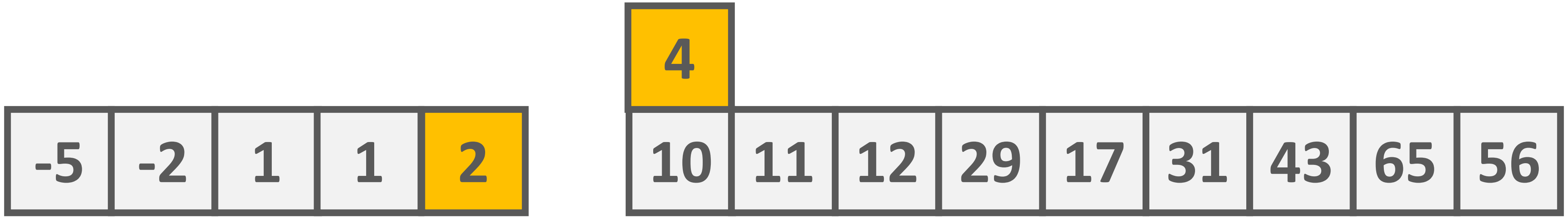
Shell Sort



Shell Sort



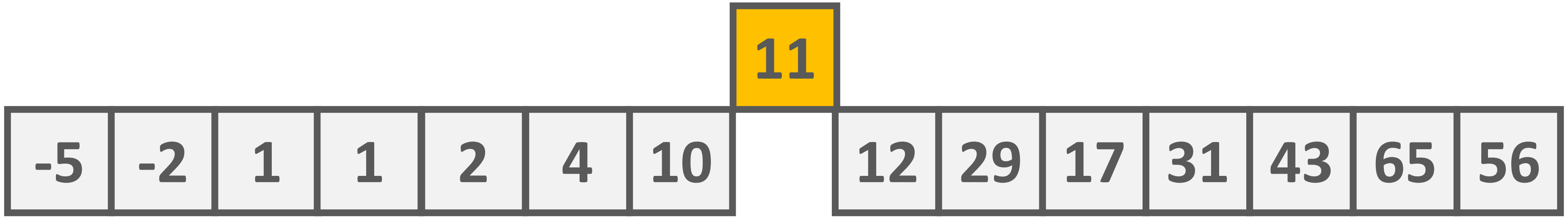
Shell Sort



Shell Sort

-5	-2	1	1	2	4	10	11	12	29	17	31	43	65	56

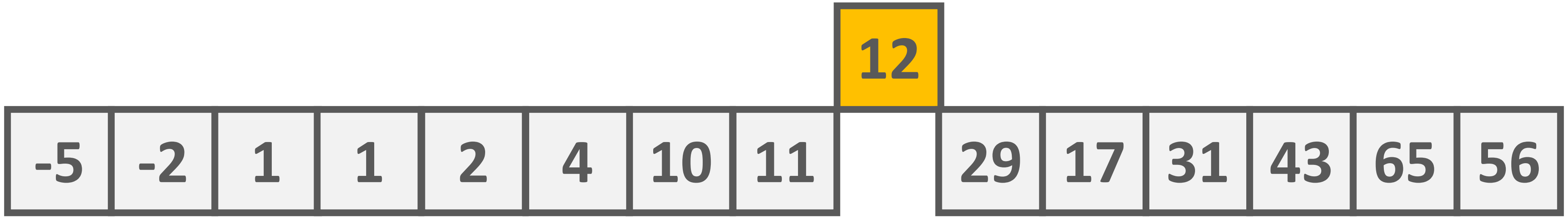
Shell Sort



Shell Sort

-5	-2	1	1	2	4	10	11	12	29	17	31	43	65	56

Shell Sort



Shell Sort

-5	-2	1	1	2	4	10	11	12	29	17	31	43	65	56

Shell Sort

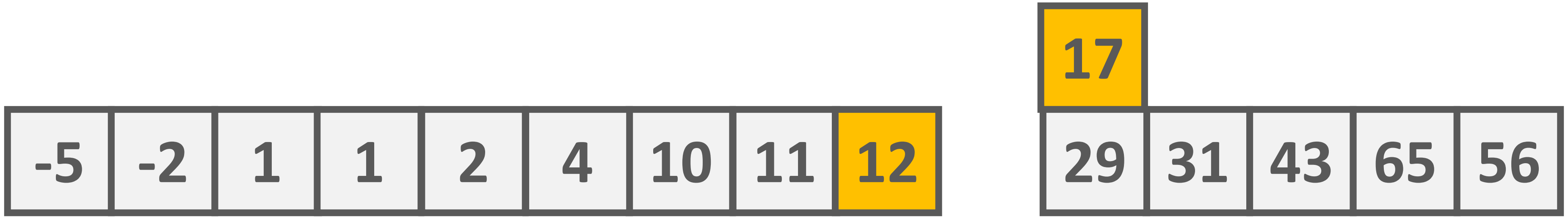
Shell Sort

-5	-2	1	1	2	4	10	11	12	29	17	31	43	65	56

Shell Sort

Shell Sort

Shell Sort



Shell Sort

-5	-2	1	1	2	4	10	11	12	17	29	31	43	65	56

Shell Sort

Shell Sort

-5	-2	1	1	2	4	10	11	12	17	29	31	43	65	56

Shell Sort

Shell Sort

-5	-2	1	1	2	4	10	11	12	17	29	31	43	65	56

Shell Sort

Shell Sort

-5	-2	1	1	2	4	10	11	12	17	29	31	43	65	56

Shell Sort

56

-5	-2	1	1	2	4	10	11	12	17	29	31	43	65

Shell Sort

Shell Sort

-5	-2	1	1	2	4	10	11	12	17	29	31	43

56
65

Shell Sort

-5	-2	1	1	2	4	10	11	12	17	29	31	43	56	65

Quicksort Algorithm

(Algorithms and Data Structures)

Quicksort

- **quicksort** was developed by **Tony Hoare** in **1959** – the same person who invented quickselect algorithm
- it is a **divide and conquer** algorithm – divides the problem into smaller and smaller subproblems
- it is an efficient sorting algorithm that has **O(NlogN)** average-case running time complexity
- a well implemented quicksort can outperform heapsort and merge sort algorithms
- **quicksort** is a comparison based sorting algorithm
- it is an in-place algorithm but **not stable**

Quicksort

- the efficient implementation of quicksort is **NOT** stable – does not keep the relative order of items with equal value
- it is in-place so does not need any additional memory
- on average it has **O(NlogN)** running time
- but the worst case running time is **$O(N^2)$** quadratic
- **quicksort** is widely used in programming languages
- for primitive types (ints, floats) quicksort is used
- for reference types (objects) merge sort is used
- Python relies heavily on **timsort**

Quicksort

Quicksort algorithm has **2** phases

1.) PARTITION PHASE

The algorithms generates a pivot item and partition the array. The pivot is the item in the middle:

- smaller items are on the left side of the pivot
- larger items are on the right side of the pivot

2.) RECURSION PHASE

We found the left and right subarrays during partition. We call the quicksort function recursively on both subarrays.

Quicksort

Quicksort algorithm has **2** phases

1.) PARTITION PHASE

The algorithms generates a pivot item and partition the array. The pivot is the item in the middle:

- smaller items are on the left side of the pivot
- larger items are on the right side of the pivot

How to generate the **pivot item**? There are **2** main approaches

1.) we can use the middle item of the array as the pivot

2.) we can generate a random item

Quicksort

1.) THE PARTITION PHASE

The partition method is just for partitioning the array according to the **pivot**

→ choose a pivot value at **random**: we generate a random number in the range **[first_index, last_index]**

→ re-arrange the array in a way that all elements less than pivot are on left side of pivot and others on right.

~ partition returns with the final position (index) of the pivot element

THE PIVOT IS ALWAYS IN ITS FINAL POSITION IN THE SORTED ORDER

Quicksort

1.) THE PARTITION PHASE

7	-2	5	8	1	6

Quicksort

1.) THE PARTITION PHASE

7	-2	5	8	1	6

Quicksort

1.) THE PARTITION PHASE

1	-2	5	8	7	6

→ choose a pivot value at **random**: we generate a random number in the range **[first_index, last_index]**

→ re-arrange the array in a way that all elements less than pivot are on left side of pivot and others on right.

Quicksort

1.) THE PARTITION PHASE

1	-2	5	8	7	6

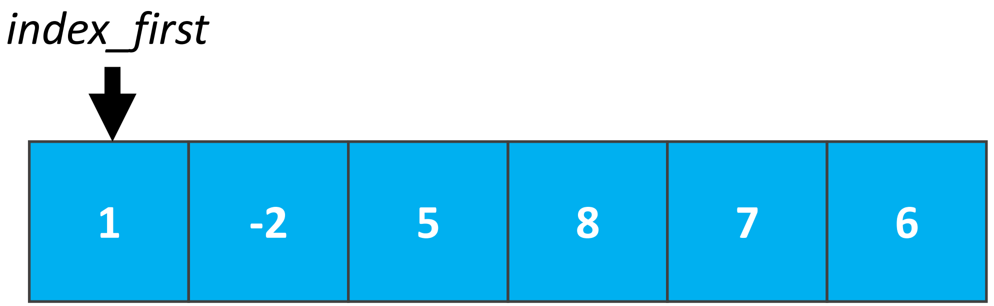
We are done, **we return the index of the pivot**! Of course in the course of the algorithm, we may have to make several partition procedure

Main idea behind quicksort: we use the same approach on both subarrays

LEFT SIDE – we use the excact same approach but of course on smaller and smaller arrays

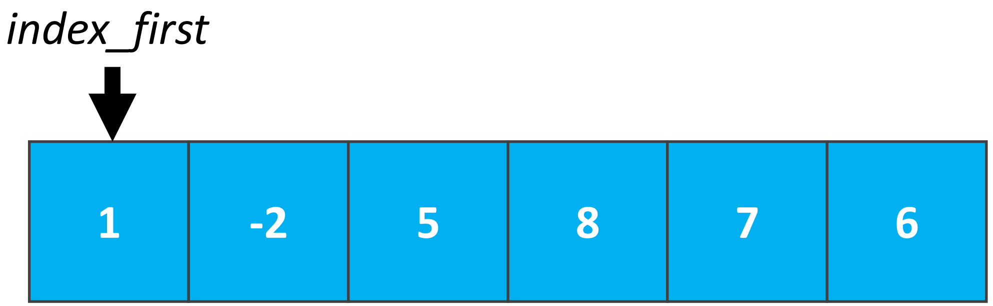
RIGHT SIDE - we use the excact same approach but of course on smaller and smaller arrays

Quickselect



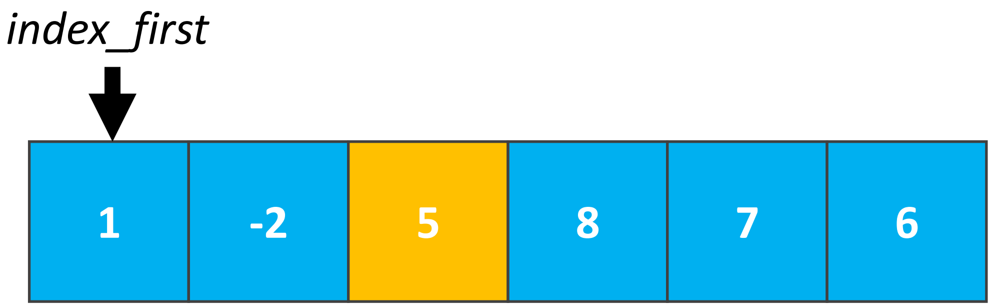

```
partition(index_first, index_last)

    pivot = random(index_first, index_last)
    swap(index_last, pivot)

    for i = index_first upto index_last
        if nums[i] < nums[index_last]
                swap(i, index_first)
                index_first+=1

    swap(index_first, index_last)

    return index_first
```

Quickselect



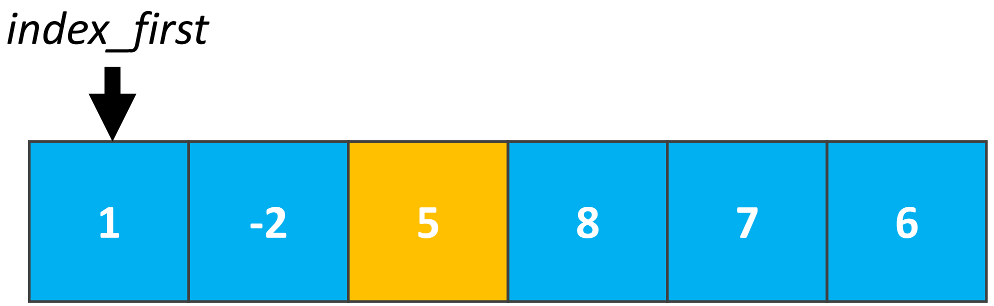

```
partition(index_first, index_last)

    pivot = random(index_first, index_last)
    swap(index_last, pivot)

    for i = index_first upto index_last
        if nums[i] < nums[index_last]
                swap(i, index_first)
                index_first+=1

    swap(index_first, index_last)

    return index_first
```

Quickselect



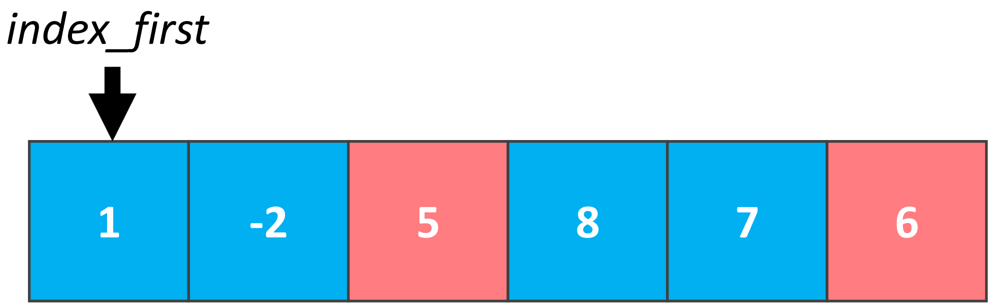

```
partition(index_first, index_last)

    pivot = random(index_first, index_last)
    swap(index_last, pivot)

    for i = index_first upto index_last
        if nums[i] < nums[index_last]
                swap(i, index_first)
                index_first+=1

    swap(index_first, index_last)

    return index_first
```

Quickselect



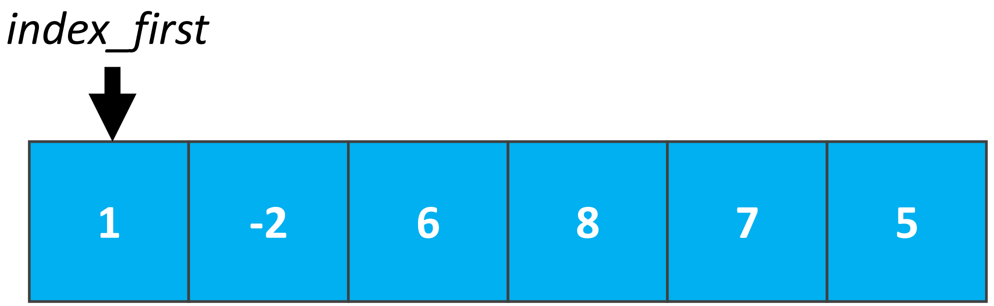

```
partition(index_first, index_last)

    pivot = random(index_first, index_last)
    swap(index_last, pivot)

    for i = index_first upto index_last
        if nums[i] < nums[index_last]
                swap(i, index_first)
                index_first+=1

    swap(index_first, index_last)

    return index_first
```

Quickselect



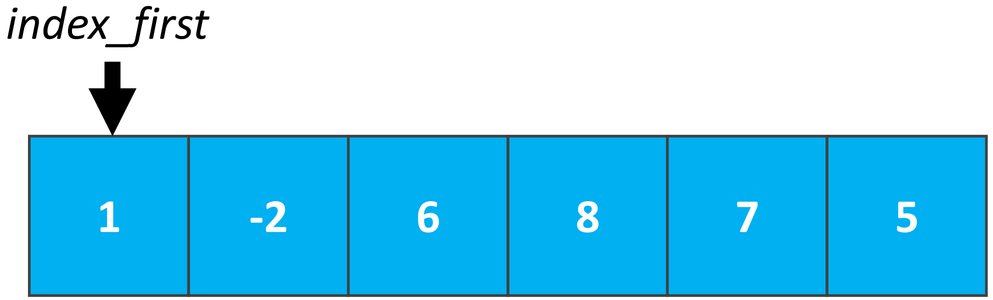

```
partition(index_first, index_last)

    pivot = random(index_first, index_last)
    swap(index_last, pivot)

    for i = index_first upto index_last
        if nums[i] < nums[index_last]
                swap(i, index_first)
                index_first+=1

    swap(index_first, index_last)

    return index_first
```

Quickselect



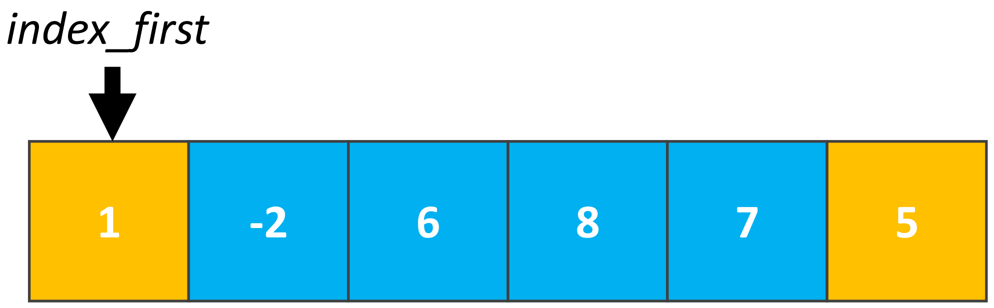

```
partition(index_first, index_last)

    pivot = random(index_first, index_last)
    swap(index_last, pivot)

    for i = index_first upto index_last
        if nums[i] < nums[index_last]
                swap(i, index_first)
                index_first+=1

    swap(index_first, index_last)

    return index_first
```

Quickselect



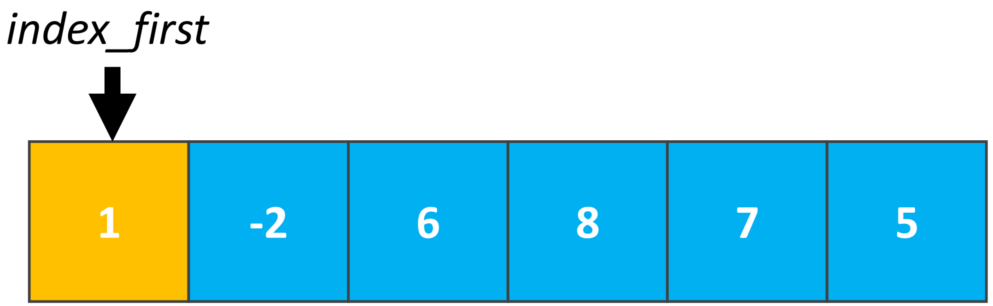

```
partition(index_first, index_last)

    pivot = random(index_first, index_last)
    swap(index_last, pivot)

    for i = index_first upto index_last
        if nums[i] < nums[index_last]
                swap(i, index_first)
                index_first+=1

    swap(index_first, index_last)

    return index_first
```

Quickselect



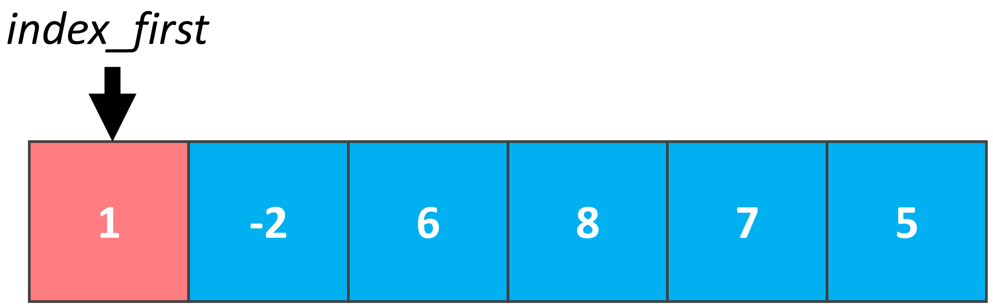

```
partition(index_first, index_last)

    pivot = random(index_first, index_last)
    swap(index_last, pivot)

    for i = index_first upto index_last
        if nums[i] < nums[index_last]
                swap(i, index_first)
                index_first+=1

    swap(index_first, index_last)

    return index_first
```

Quickselect



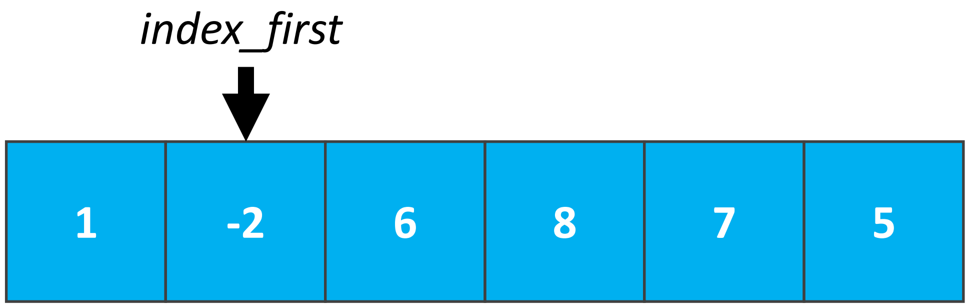

```
partition(index_first, index_last)

    pivot = random(index_first, index_last)
    swap(index_last, pivot)

    for i = index_first upto index_last
        if nums[i] < nums[index_last]
                swap(i, index_first)
                index_first+=1

    swap(index_first, index_last)

    return index_first
```

Quickselect



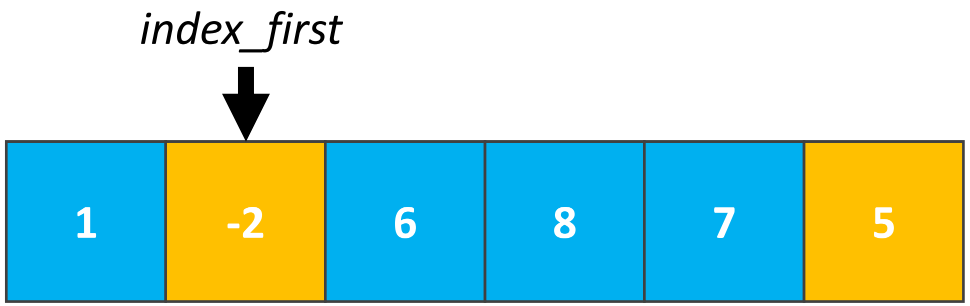

```
partition(index_first, index_last)

    pivot = random(index_first, index_last)
    swap(index_last, pivot)

    for i = index_first upto index_last
        if nums[i] < nums[index_last]
                swap(i, index_first)
                index_first+=1

    swap(index_first, index_last)

    return index_first
```

Quickselect



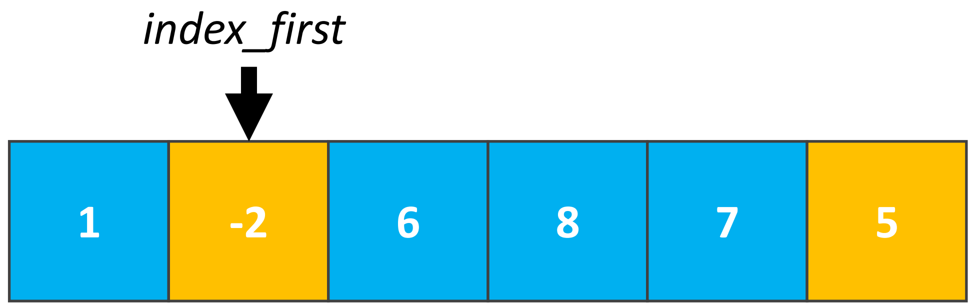

```
partition(index_first, index_last)

    pivot = random(index_first, index_last)
    swap(index_last, pivot)

    for i = index_first upto index_last
        if nums[i] < nums[index_last]
                swap(i, index_first)
                index_first+=1

    swap(index_first, index_last)

    return index_first
```

Quickselect



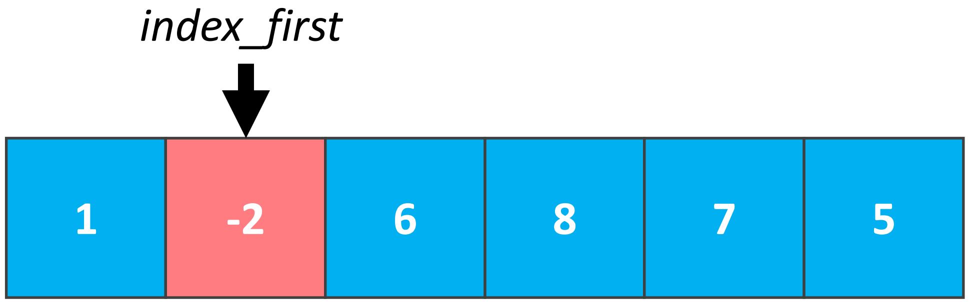

```
partition(index_first, index_last)

    pivot = random(index_first, index_last)
    swap(index_last, pivot)

    for i = index_first upto index_last
        if nums[i] < nums[index_last]
                swap(i, index_first)
                index_first+=1

    swap(index_first, index_last)

    return index_first
```

Quickselect



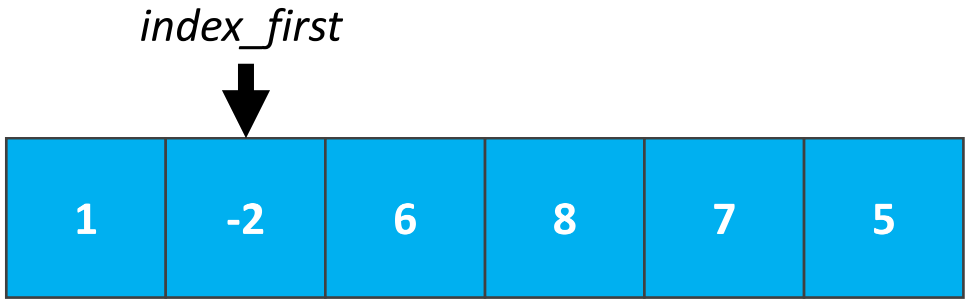

```
partition(index_first, index_last)

    pivot = random(index_first, index_last)
    swap(index_last, pivot)

    for i = index_first upto index_last
        if nums[i] < nums[index_last]
                swap(i, index_first)
                index_first+=1

    swap(index_first, index_last)

    return index_first
```

Quickselect



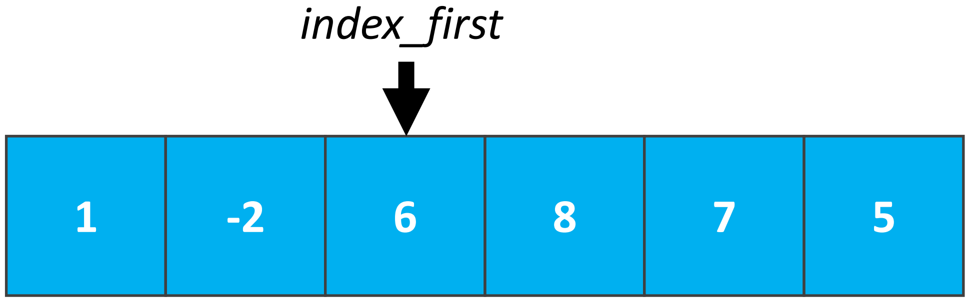

```
partition(index_first, index_last)

    pivot = random(index_first, index_last)
    swap(index_last, pivot)

    for i = index_first upto index_last
        if nums[i] < nums[index_last]
                swap(i, index_first)
                index_first+=1

    swap(index_first, index_last)

    return index_first
```

Quickselect



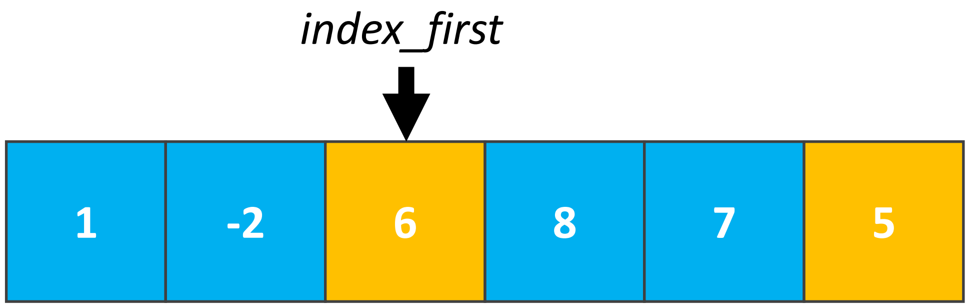

```
partition(index_first, index_last)

    pivot = random(index_first, index_last)
    swap(index_last, pivot)

    for i = index_first upto index_last
        if nums[i] < nums[index_last]
                swap(i, index_first)
                index_first+=1

    swap(index_first, index_last)

    return index_first
```

Quickselect



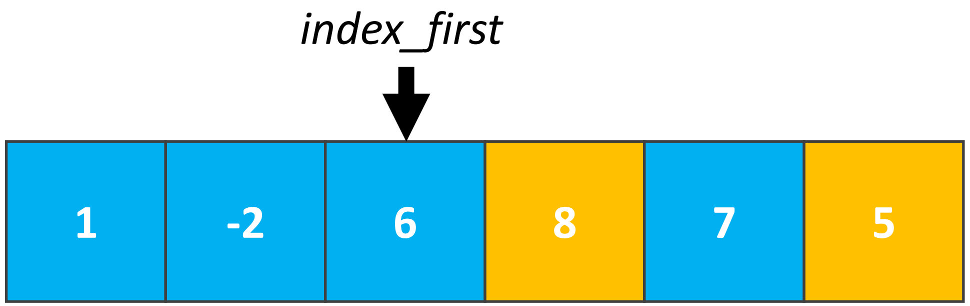

```
partition(index_first, index_last)

    pivot = random(index_first, index_last)
    swap(index_last, pivot)

    for i = index_first upto index_last
        if nums[i] < nums[index_last]
               swap(i, index_first)
               index_first+=1

    swap(index_first, index_last)

    return index_first
```

Quickselect



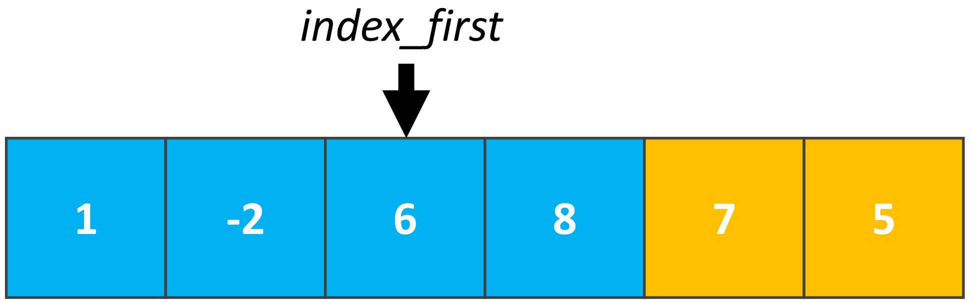

```
partition(index_first, index_last)

    pivot = random(index_first, index_last)
    swap(index_last, pivot)

    for i = index_first upto index_last
        if nums[i] < nums[index_last]
                swap(i, index_first)
                index_first+=1

    swap(index_first, index_last)

    return index_first
```

Quickselect



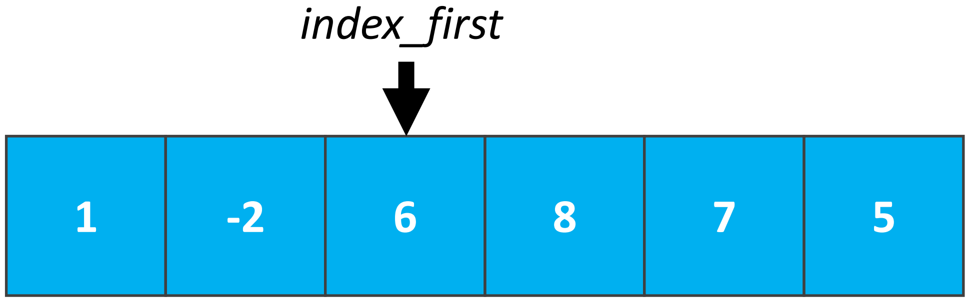

```
partition(index_first, index_last)

    pivot = random(index_first, index_last)
    swap(index_last, pivot)

    for i = index_first upto index_last
        if nums[i] < nums[index_last]
                swap(i, index_first)
                index_first+=1

    swap(index_first, index_last)

    return index_first
```

Quickselect



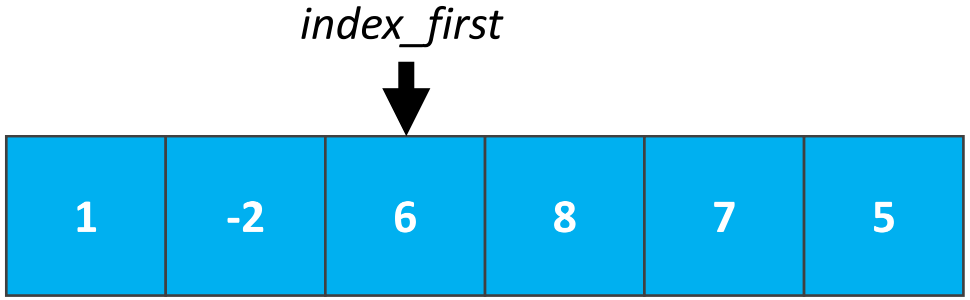

```
partition(index_first, index_last)

    pivot = random(index_first, index_last)
    swap(index_last, pivot)

    for i = index_first upto index_last
        if nums[i] < nums[index_last]
                swap(i, index_first)
                index_first+=1

    swap(index_first, index_last)

    return index_first
```

Quickselect



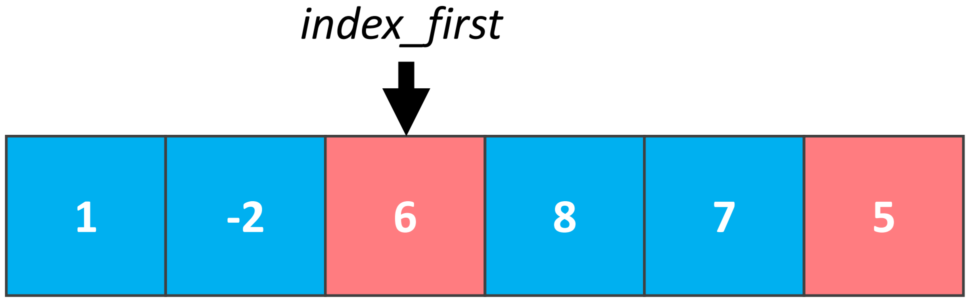

```
partition(index_first, index_last)

    pivot = random(index_first, index_last)
    swap(index_last, pivot)

    for i = index_first upto index_last
        if nums[i] < nums[index_last]
                swap(i, index_first)
                index_first+=1

    swap(index_first, index_last)

    return index_first
```

Quickselect



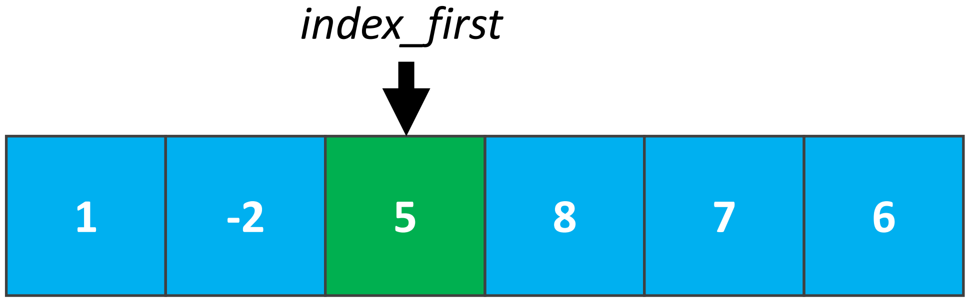

```
partition(index_first, index_last)

    pivot = random(index_first, index_last)
    swap(index_last, pivot)

    for i = index_first upto index_last
        if nums[i] < nums[index_last]
                swap(i, index_first)
                index_first+=1

    swap(index_first, index_last)

    return index_first
```

Quickselect



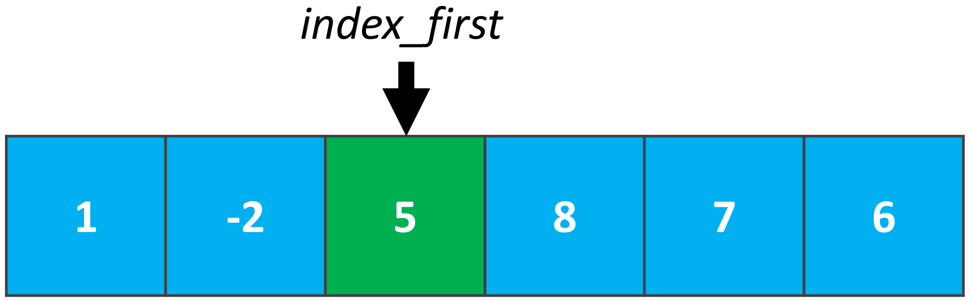

```
partition(index_first, index_last)

    pivot = random(index_first, index_last)
    swap(index_last, pivot)

    for i = index_first upto index_last
        if nums[i] < nums[index_last]
                swap(i, index_first)
                index_first+=1

    swap(index_first, index_last)

    return index_first
```

Quickselect




```
partition(index_first, index_last)

   pivot = random(index_first, index_last)
   swap(index_last, pivot)

   for i = index_first upto index_last
       if nums[i] < nums[index_last]
               swap(i, index_first)
               index_first+=1

   swap(index_first, index_last)

   return index_first
```

Quickselect




```
partition(index_first, index_last)

    pivot = random(index_first, index_last)
    swap(index_last, pivot)

    for i = index_first upto index_last
        if nums[i] < nums[index_last]
                swap(i, index_first)
                index_first+=1

    swap(index_first, index_last)

    return index_first
```

Quickselect

2.) RECURSION PHASE

```
quicksort(array, low, high)

    if low >=  high
        return

    pivot = partition(array,low,high)
    quicksort(array,low,pivot-1)
    quicksort(array,pivot+1,high)

end
```

*there is the **partition** phase when we keep finding the pivot item (in every iteration the pivot will be sorted)*

*call the same **quicksort** function recursively on the left subarray and right subarray*

Problem with Quicksort
(Algorithmic Problems)

Quicksort

- **quicksort algorithm** is extremely sensitive to the pivot item
- each partition phase takes **O(N)** linear running time – of course **N** is smaller and smaller in every recursive call
- if we are not able to discard many items: the **O(N)** linear running time may be reduced to **$O(N^2)$** running time
- the pivot selection approach is crucial (!!!)

Quicksort

- let’s assume we are looking for the smallest value
- the **wors-case scenario** happens when we pick the largest item in every iteration to be the pivot
- the partition phase takes **O(N)** time and we make **N** iteration

Quicksort

1	-2	5	8	7	6	10	4

Quicksort

1	-2	5	8	7	6	10	4

Quicksort

1	-2	5	8	7	6	10	4

Quicksort

1	-2	5	8	7	6	4	10

Quicksort

1	-2	5	8	7	6	4	10

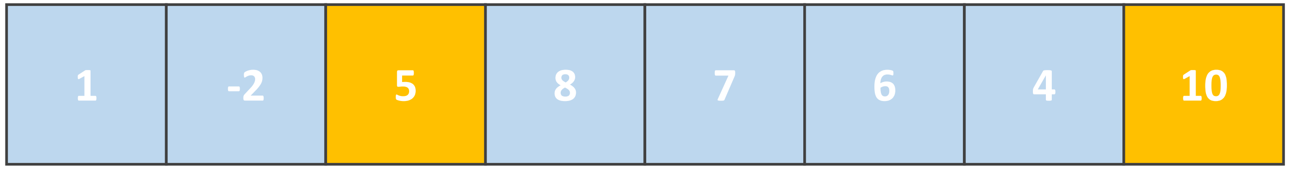
Quicksort

1	-2	5	8	7	6	4	10

Quicksort

1	-2	5	8	7	6	4	10

Quicksort



Quicksort

1	-2	5	8	7	6	4	10

Quicksort

1	-2	5	8	7	6	4	10

Quicksort

1	-2	5	8	7	6	4	10

Quicksort

1	-2	5	8	7	6	4	10

Quicksort

1	-2	5	8	7	6	4	10

Quicksort

1	-2	5	8	7	6	4	10

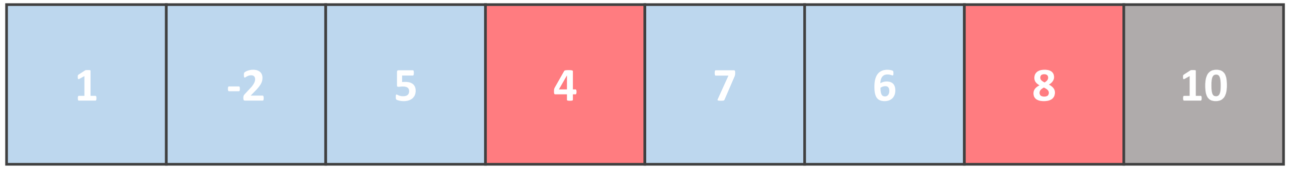
Quicksort

1	-2	5	8	7	6	4	10

Quicksort

1	-2	5	8	7	6	4	10

Quicksort



Quicksort

1	-2	5	4	7	6	8	10

Quicksort

1	-2	5	4	7	6	8	10

Quicksort

1	-2	5	4	7	6	8	10

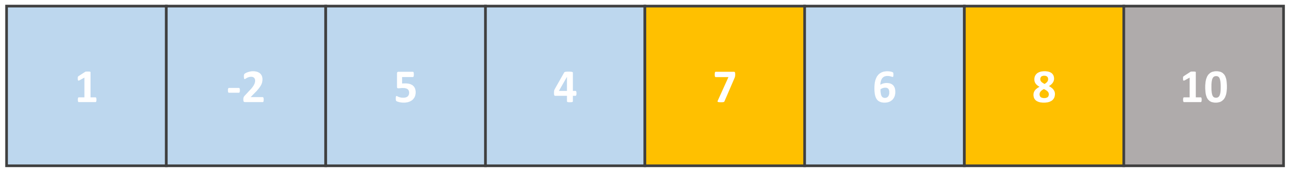
Quicksort

1	-2	5	4	7	6	8	10

Quicksort

1	-2	5	4	7	6	8	10

Quicksort



Quicksort

1	-2	5	4	7	6	8	10

Quicksort

1	-2	5	4	7	6	8	10

Quicksort

1	-2	5	4	7	6	8	10

Quicksort

1	-2	5	4	7	6	8	10

Quicksort

1	-2	5	4	7	6	8	10

Quicksort

1	-2	5	4	6	7	8	10

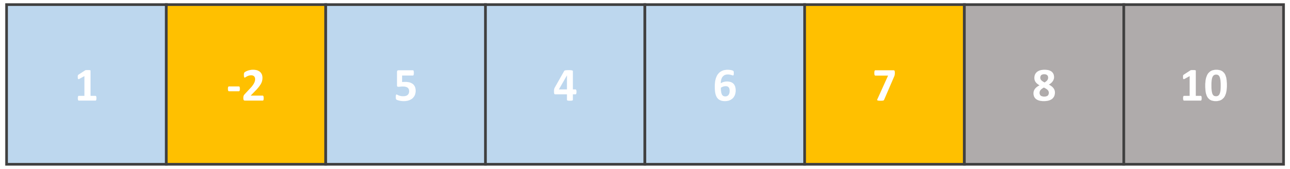
Quicksort

1	-2	5	4	6	7	8	10

Quicksort

1	-2	5	4	6	7	8	10

Quicksort



Quicksort

1	-2	5	4	6	7	8	10

Quicksort

1	-2	5	4	6	7	8	10

Quicksort

1	-2	5	4	6	7	8	10

Quicksort

1	-2	5	4	6	7	8	10

Quicksort

1	-2	5	4	6	7	8	10

Quicksort

1	-2	5	4	6	7	8	10

Quicksort

1	-2	5	4	6	7	8	10

Quicksort

1	-2	5	4	6	7	8	10

Quicksort

1	-2	5	4	6	7	8	10

Quicksort

1	-2	5	4	6	7	8	10

Quicksort

1	-2	5	4	6	7	8	10

Quicksort

1	-2	5	4	6	7	8	10

Quicksort

1	-2	5	4	6	7	8	10

Quicksort

1	-2	5	4	6	7	8	10

Quicksort

1	-2	5	4	6	7	8	10

Hybrid Sorting Algorithms

(Algorithms and Data Structures)

Hybrid Sorting Algorithms

- **hybrid algorithms** combine more algorithms to solve a given problem
- it choses one algorithm depending on the data or switching between them over the course of the algorithm
- this is generally done to combine desired features of the algorithms so that the overall algorithm is better than the individual components
- hybrid algorithm does not refer to simply combining multiple algorithms to solve a different problem
- it is about combining algorithms that solve the same problem – but differ in other characteristics (such as performance)

Hybrid Sorting Algorithms

- **heapsort** has a guaranteed **O(NlogN)** linearithmic running time complexity
- optimal implementations of **quicksort** is the fastest sorting approach but it may reduce to **$O(N^2)$** quadratic running time in worst-case
- the pivot selection approach is crucial

QUICKSORT + HEAPSORT = INTROSORT

Hybrid Sorting Algorithms

- **intro sort** (introspective sort) is the combination of quicksort and heapsort algorithms
- it is a hybrid sorting algorithm that provides both fast avarage performance and optimal worst-case performance
- it begins with quicksort and switches to heapsort when quicksort becomes too slow

Hybrid Sorting Algorithms

- **insertion sort** has several advantages in the main – and it is very efficient on small datasets (**5 - 10** elements)
- **merge sort** is asymptotically optimal on large datasets but the overhead becomes significant if applying them to small datasets
- the recusive calls on small arrays makes the algorithms slower

MERGE SORT + INSERTION SORT = TIMSORT

Hybrid Sorting Algorithms

- **timsort** is the combination of merge sort and insertion sort
- it is a stable sorting algorithms which is a huge advantage
- it was implemented by **Tim Peters** in **2002** for use in the Python programming language
- best-case running time is **O(N)** linear
- worst-case running time is **O(NlogN)** linearithmic
- worst-case space complexity is **O(N)** linear – of course merge sort is not an in-place approach

Merge Sort Algorithm

(Algorithmic Problems)

Merge Sort Algorithm

- **merge sort** is a divide and conquer algorithm that was invented by **John von Neumann** in **1945**
- it is a comparison based algorithm – which means that the algorithm relies heavily on comparing the items
- merge sort has an **O(NlogN)** linerithmic running time complexity
- it is a **stable sorting** algorithm – maintains the relative orders of items with equal values
- not an in-place approach – it requires **O(N)** additional memory

Merge Sort Algorithm

- **merge sort** is a divide and conquer algorithm that was invented by **John von Neumann** in **1945**
- although heapsort has the same time bounds as merge sort but heapsort requires only **$\Theta(1)$** auxiliary space
- an efficient quicksort implementations generally outperforms merge sort
- merge sort is often the best choice for **sorting a linked lists** - in this situation it is relatively easy to implement a merge sort in such a way that it requires only **$\Theta(1)$** extra space

Merge Sort Algorithm

DIVIDE

1.) divide the array into two subarrays recursively

2.) sort these subarrays recursively with mergesort again

CONQUER

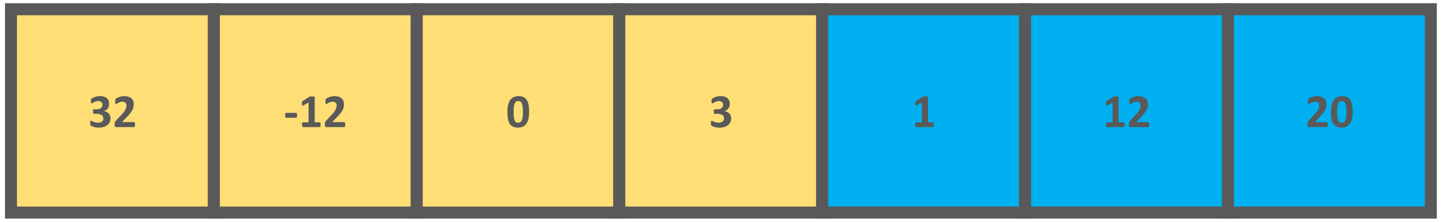
3.) if there is only a single item left in the subarray: we consider it to be sorted by definition (or we can use **insertion sort** on small arrays)

4.) merge the subarrays to get the final sorted array

Divide Phase

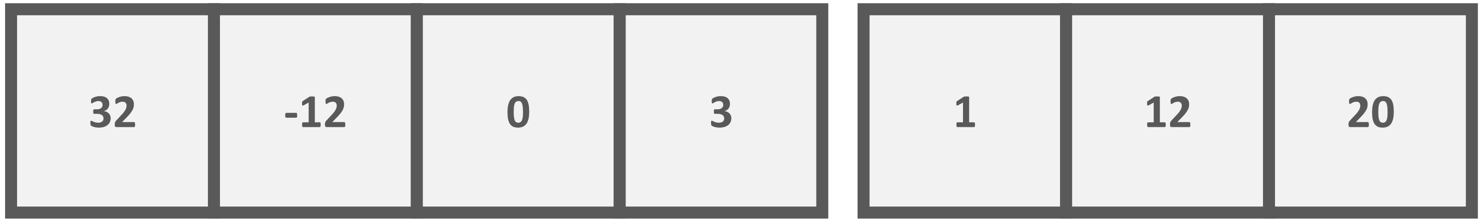
32	-12	0	3	1	12	20

Divide Phase

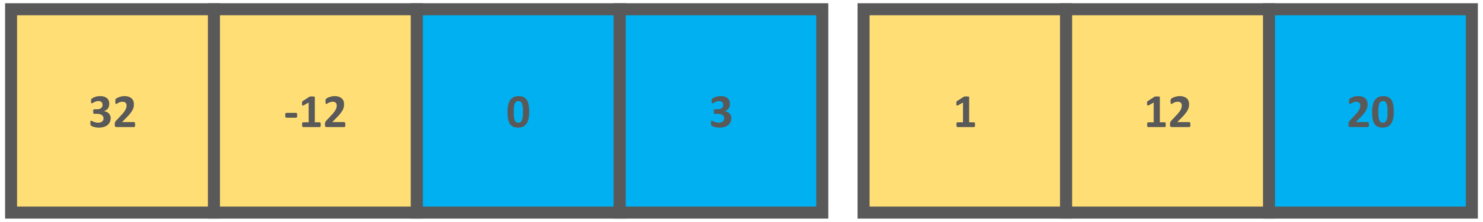


Divide Phase

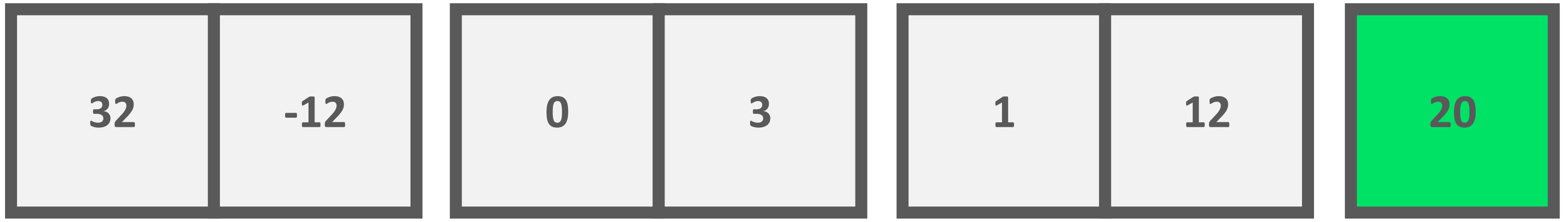
Divide Phase



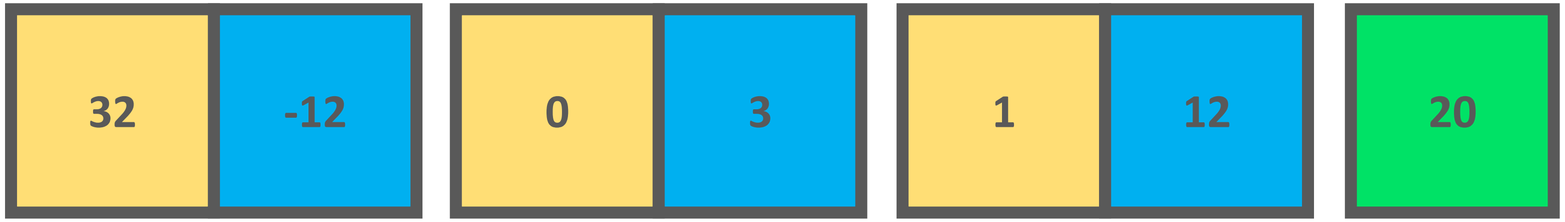
Divide Phase



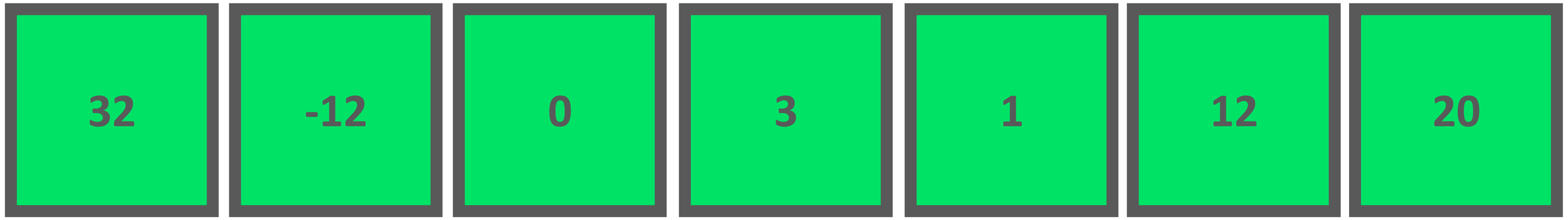
Divide Phase



Divide Phase



Divide Phase



Divide Phase

- **divide phase** keeps splitting the array into smaller and smaller subarrays
- we can use recursion until every subarray has just a single item
- not necessarily the best approach: there may be too many recursive funtion calls
- we can use **insertion sort** on small subarrays (**<5** items)
- insertion sort is efficient on datasets that are already substantially sorted – it can have **O(N+d)** linear running time in best case (**d** is the number of inversions)

Conquer Phase

- after the **divide phase** we have several small subarrays that are already sorted
- we have to merge these arrays one by one to get the final result
- this is the **conquer phase –** it runs in **O(N)** running time and this is why the final running time is **O(NlogN)**

Conquer Phase

3	5	6	10

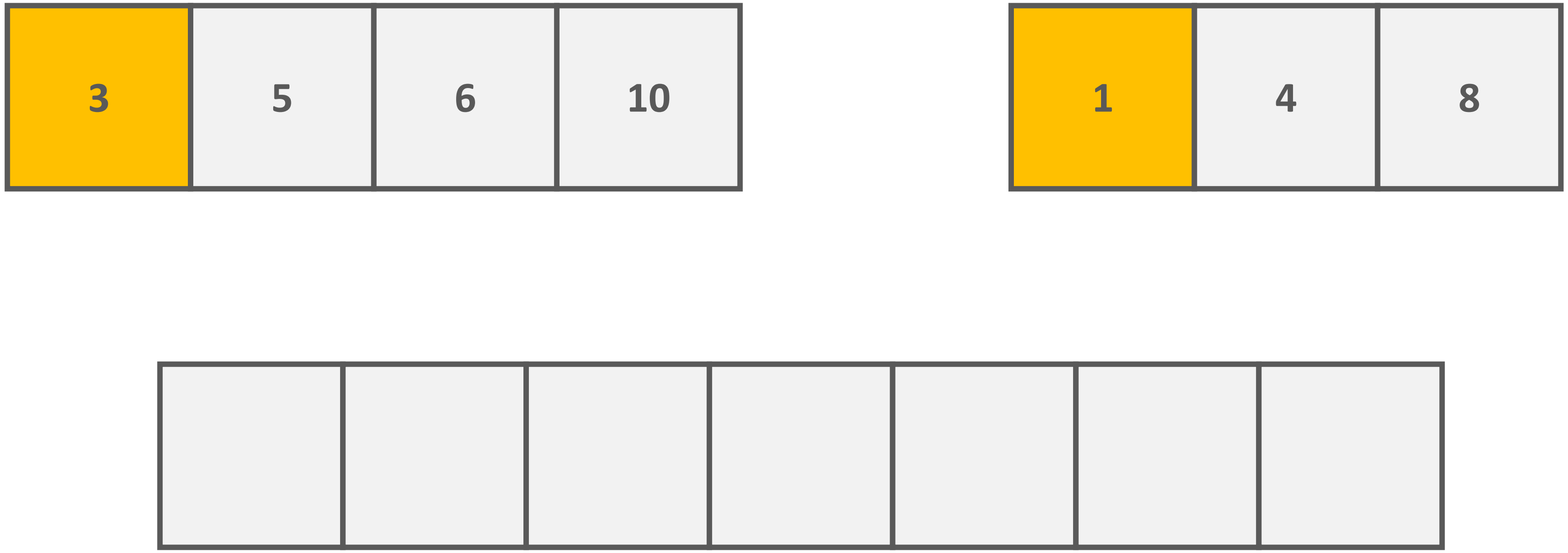
1	4	8

Conquer Phase

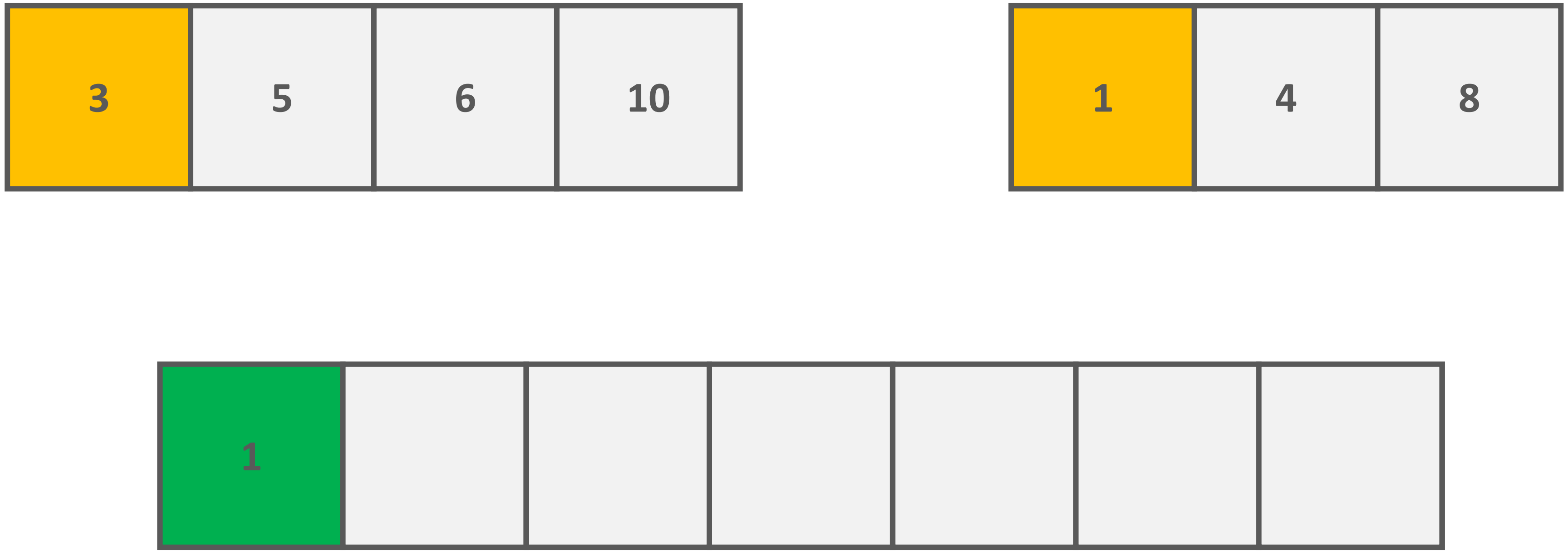
3	5	6	10

1	4	8

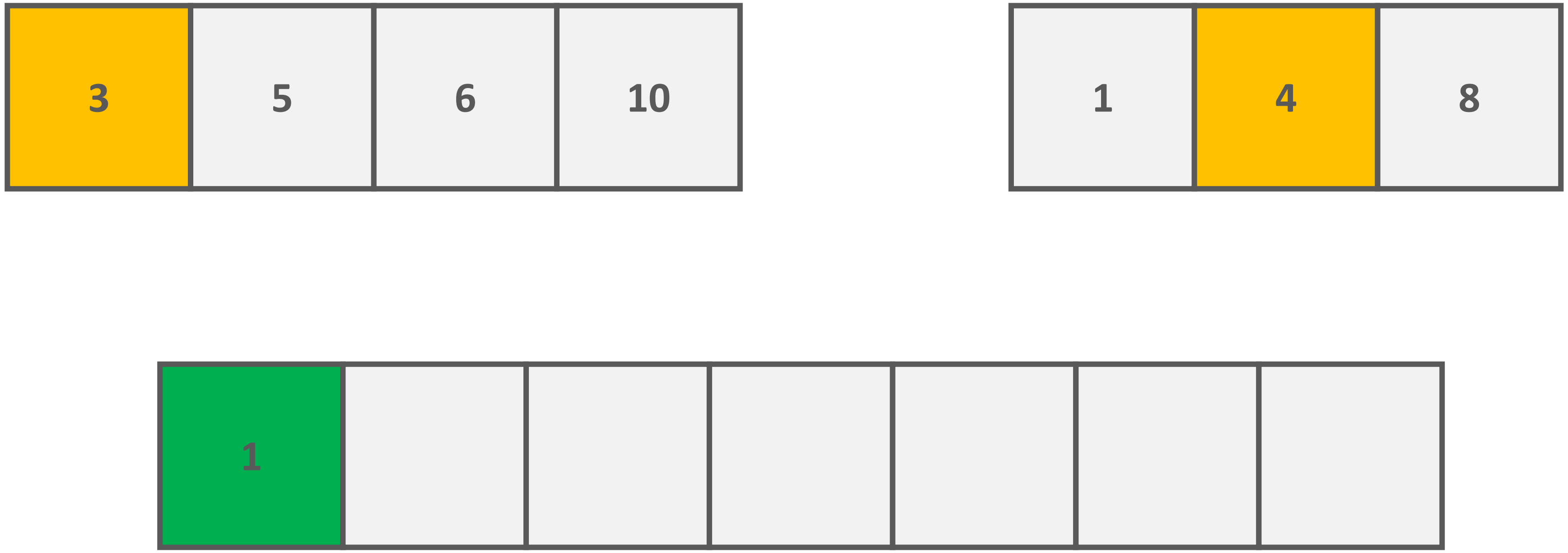
Conquer Phase



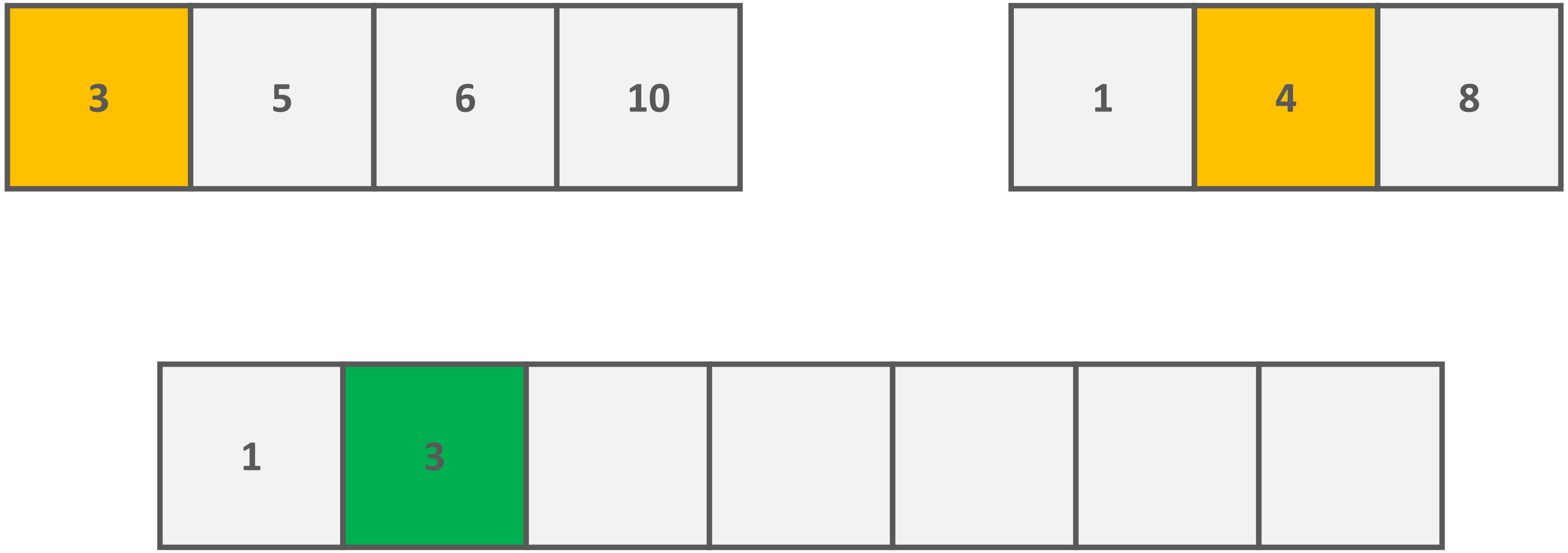
Conquer Phase



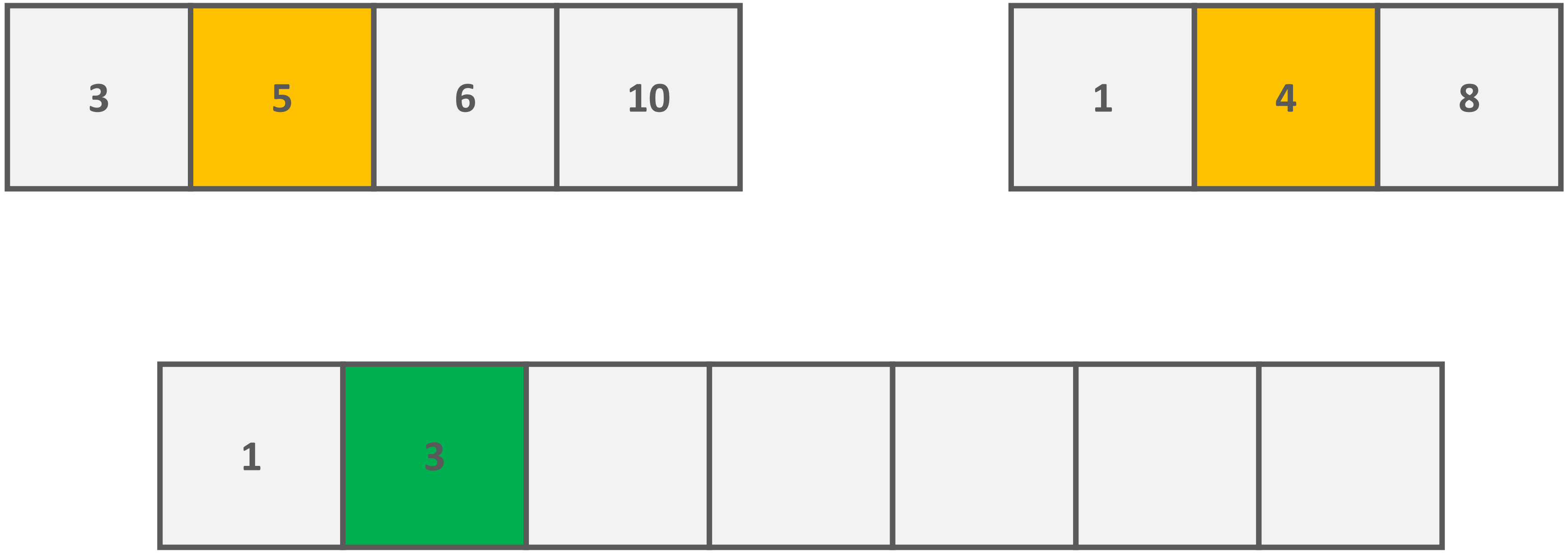
Conquer Phase



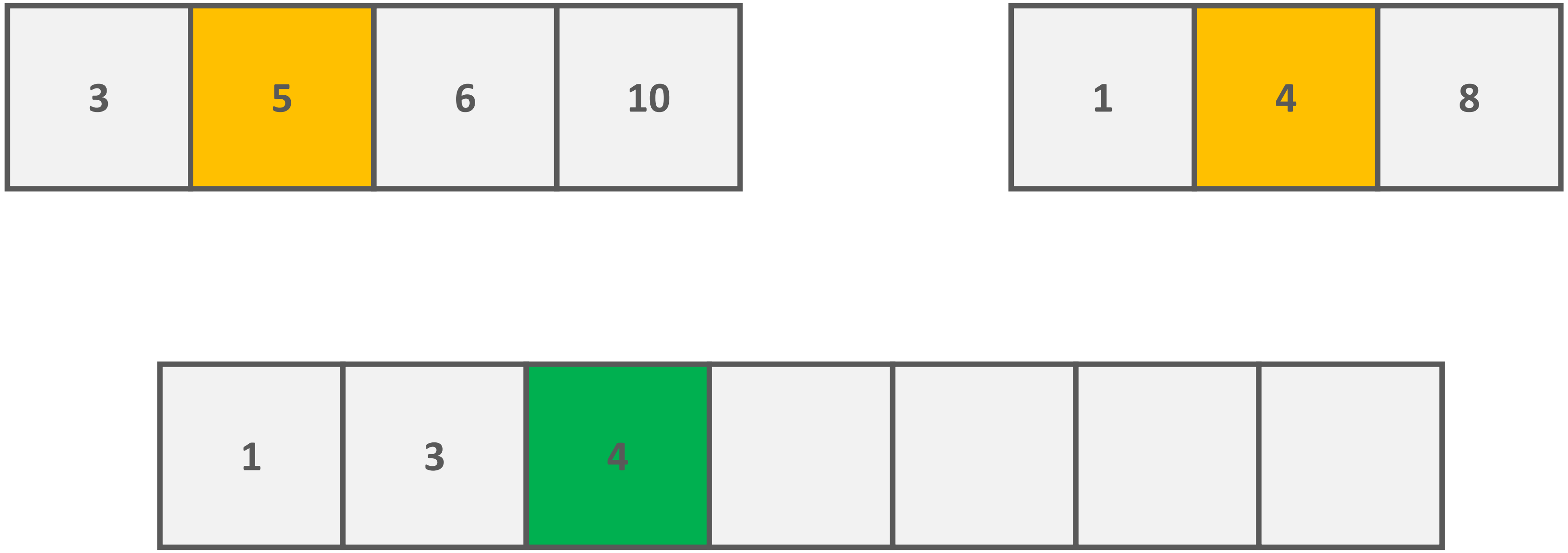
Conquer Phase



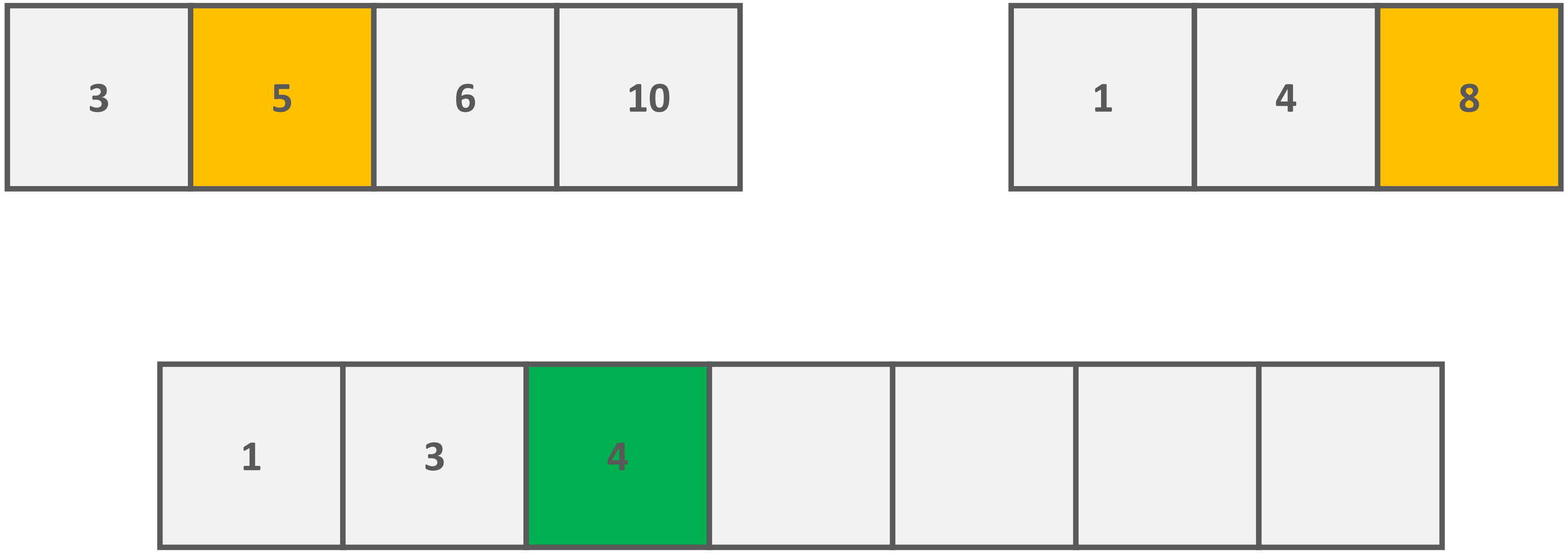
Conquer Phase



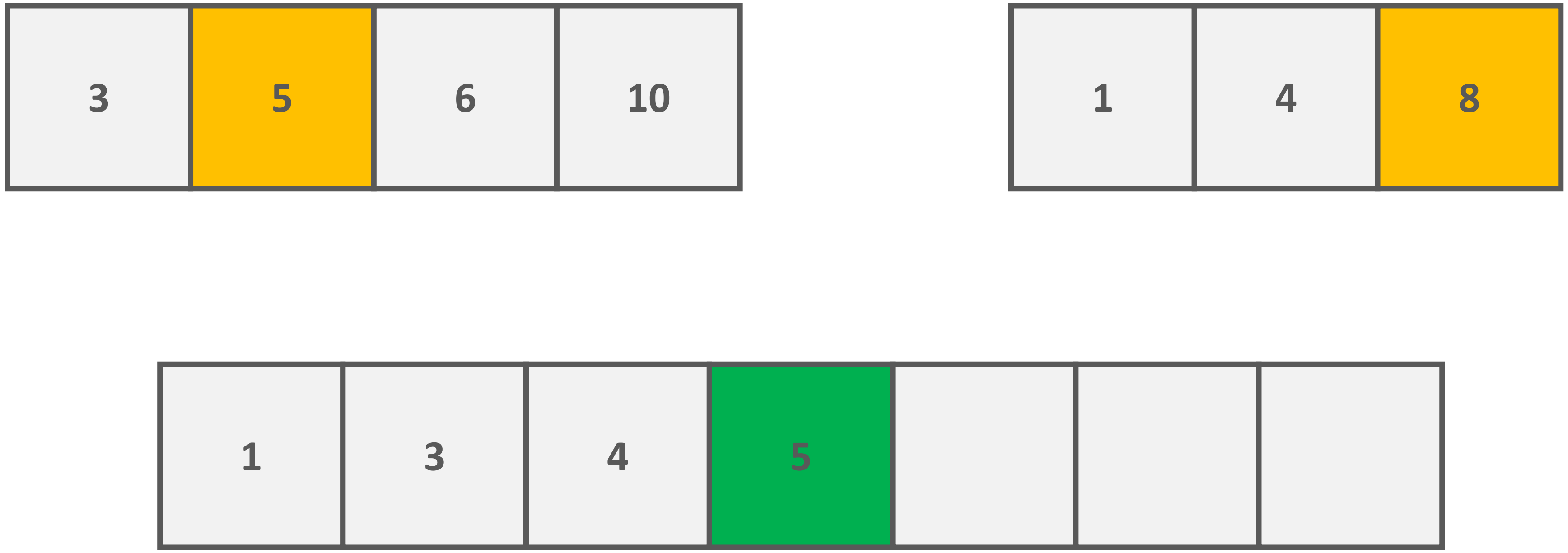
Conquer Phase



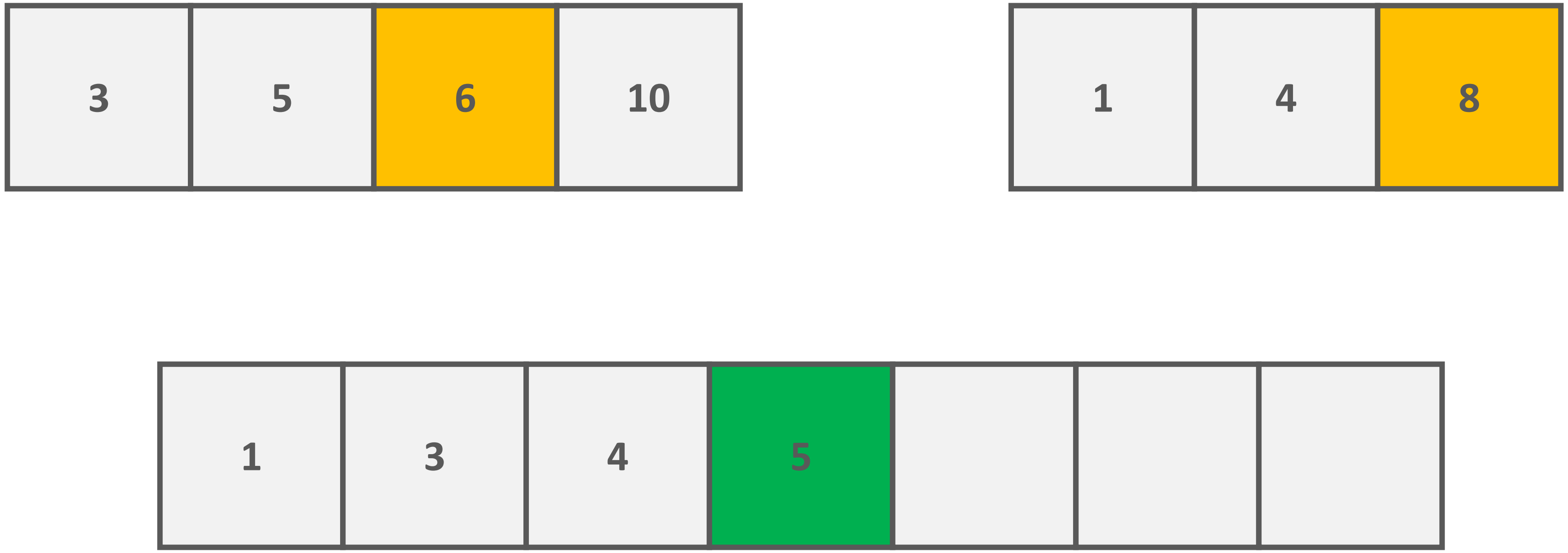
Conquer Phase



Conquer Phase



Conquer Phase



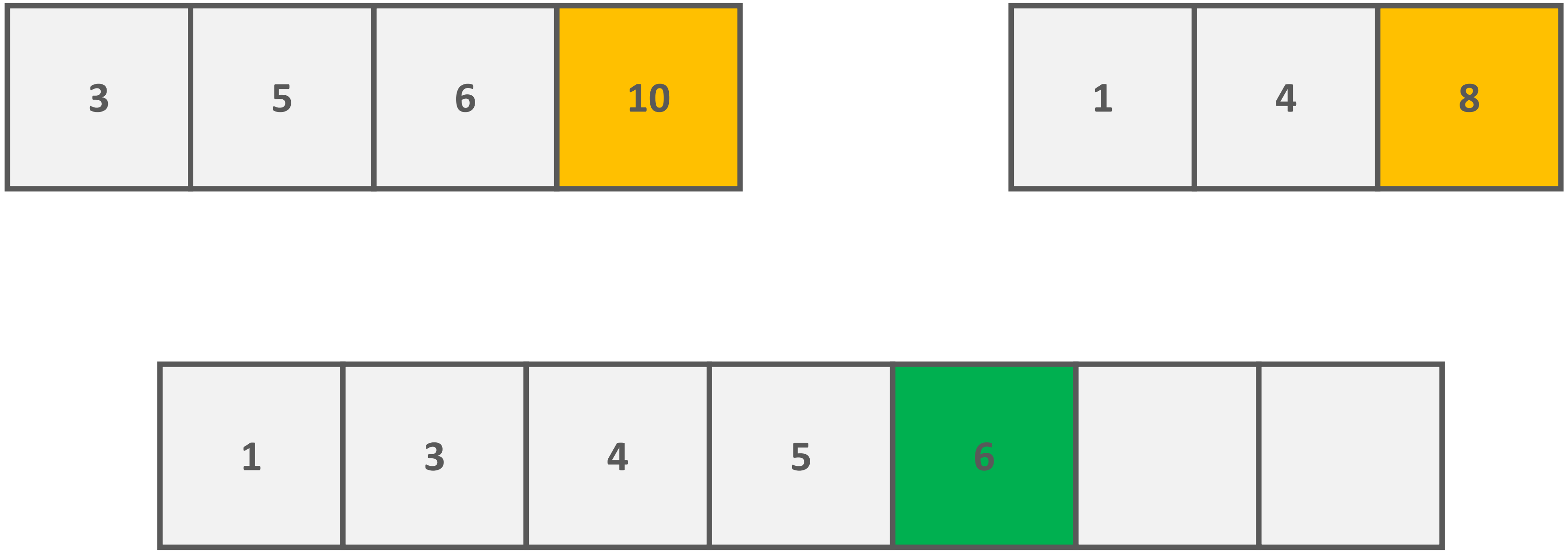
Conquer Phase

3	5	6	10

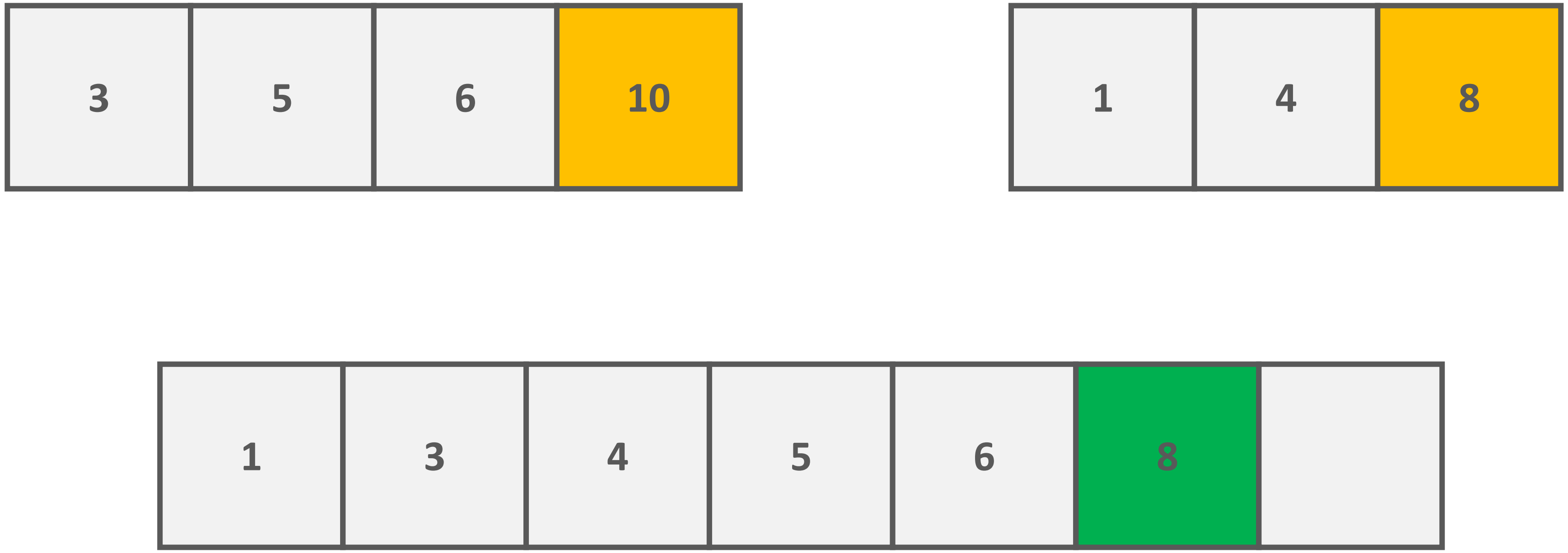
1	4	8

1	3	4	5	6		

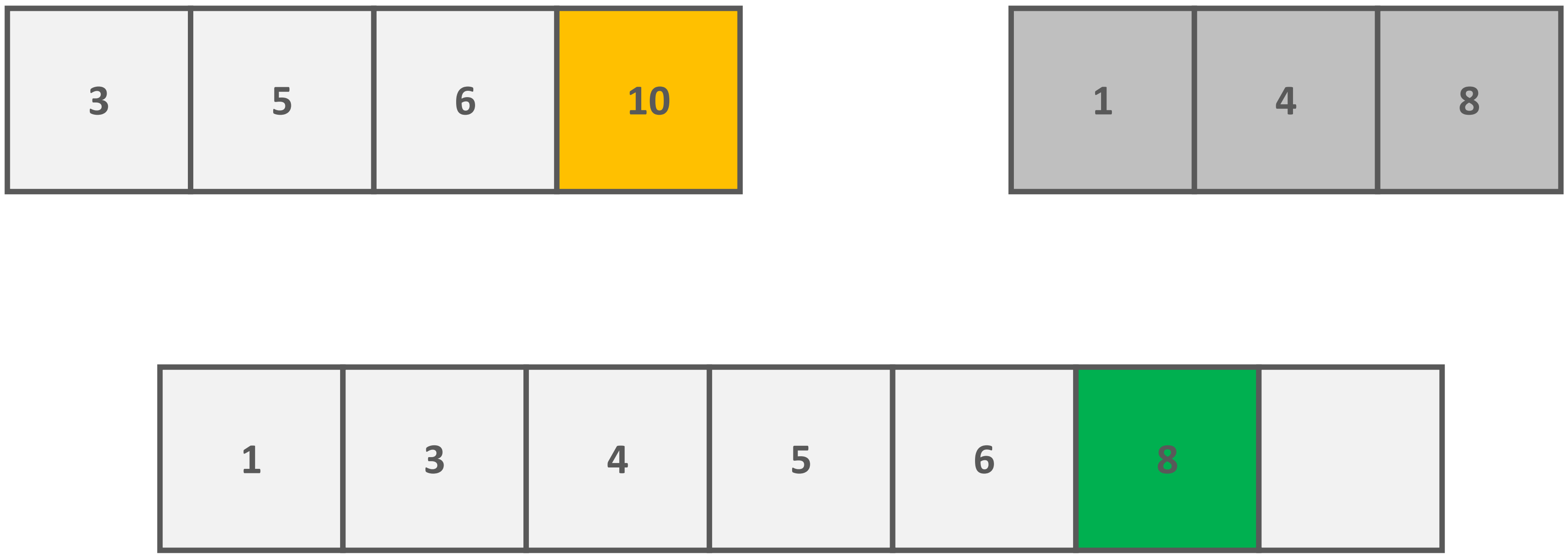
Conquer Phase



Conquer Phase



Conquer Phase



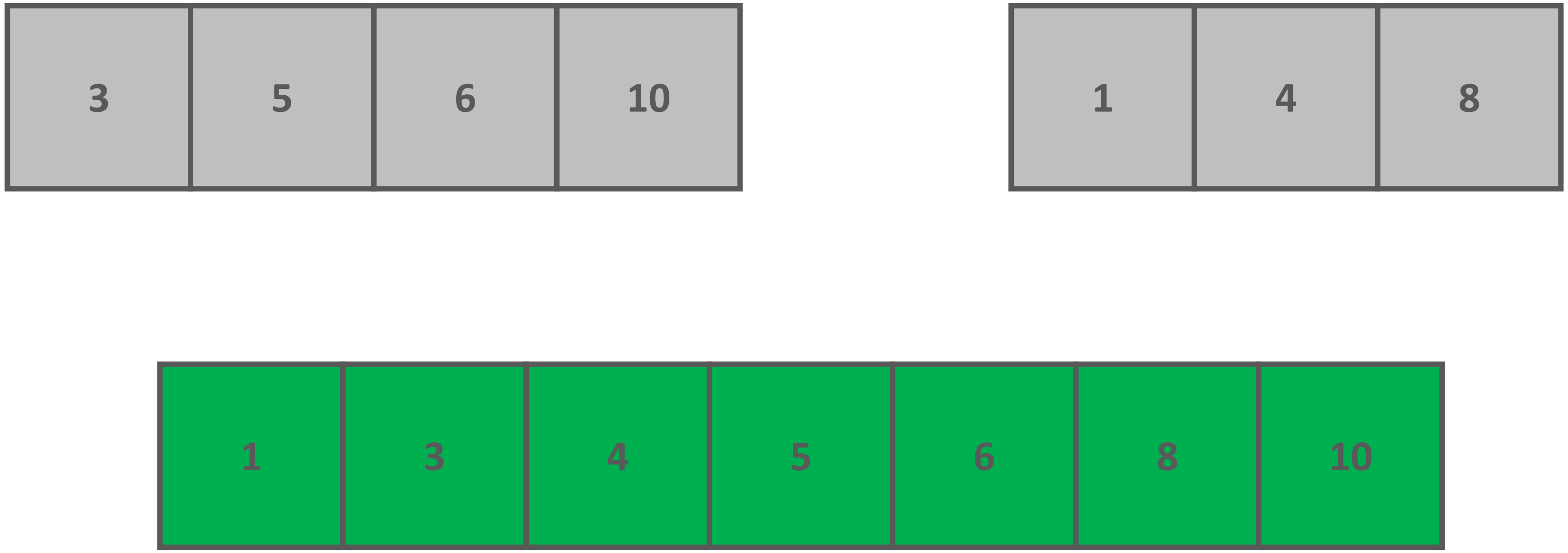
Conquer Phase

3	5	6	10

1	4	8

1	3	4	5	6	8	10

Conquer Phase



Merge Sort Example
(Algorithms and Data Structures)

Merge Sort

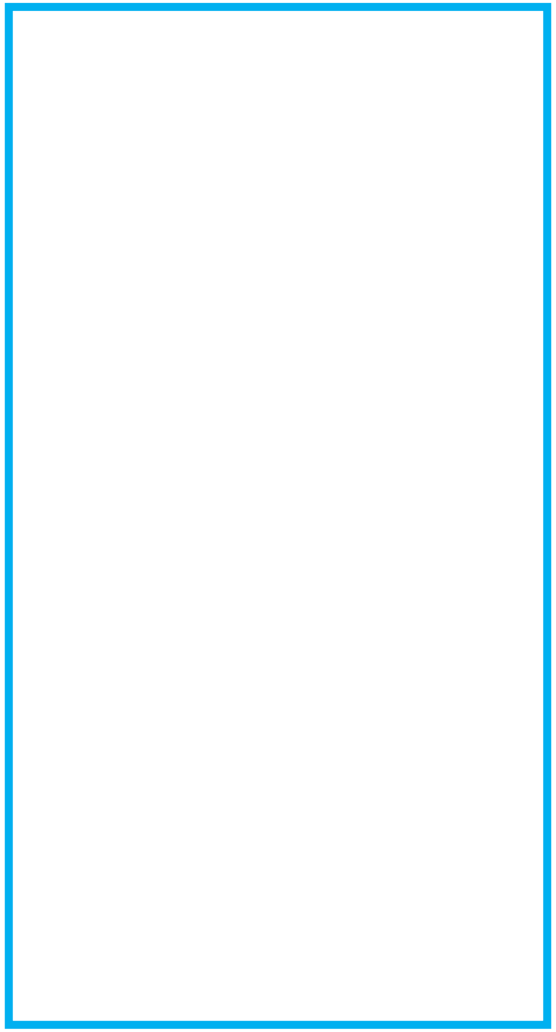
Merge Sort and the Stack Memory

(Algorithms and Data Structures)

Merge Sort

3	2	6	4	1

```
sort(data):

        if data size == 1:
            return

        select middle item and split array (left and right)
        sort(left side)
        sort(right side)

        merge arrays(left side and right side)
```



STACK

Merge Sort

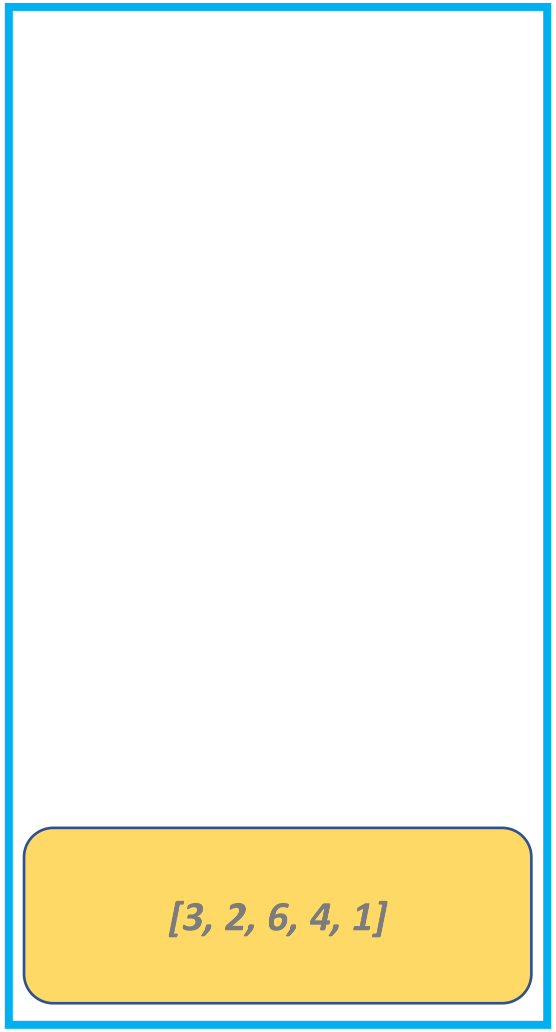
3	2	6	4	1

sort(data):

if data size == 1:
return

select middle item and split array (left and right)
sort(left side)
sort(right side)

merge arrays(left side and right side)

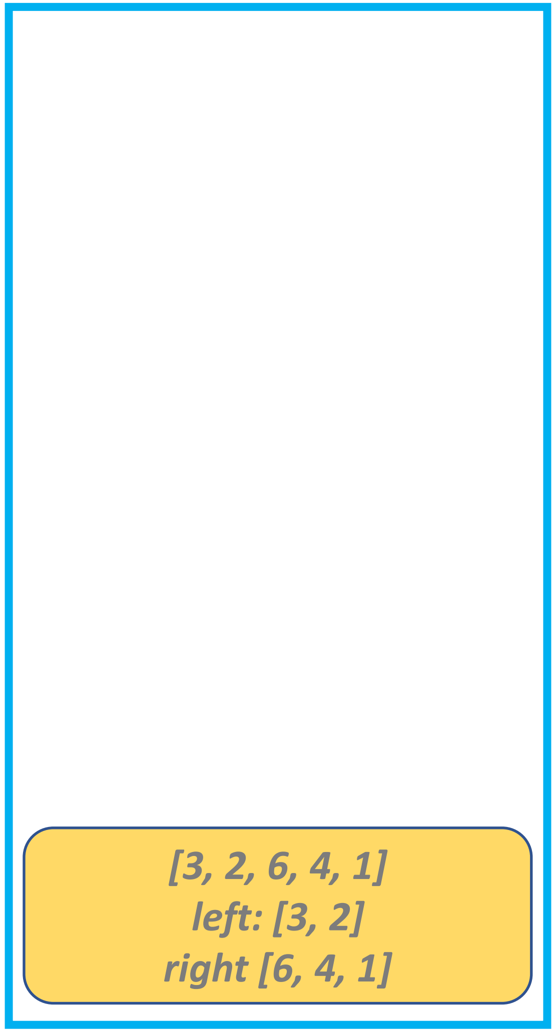


STACK

Merge Sort

3	2	6	4	1

```
sort(data):

        if data size == 1:
            return

        select middle item and split array (left and right)
        sort(left side)
        sort(right side)

        merge arrays(left side and right side)
```



STACK

Merge Sort

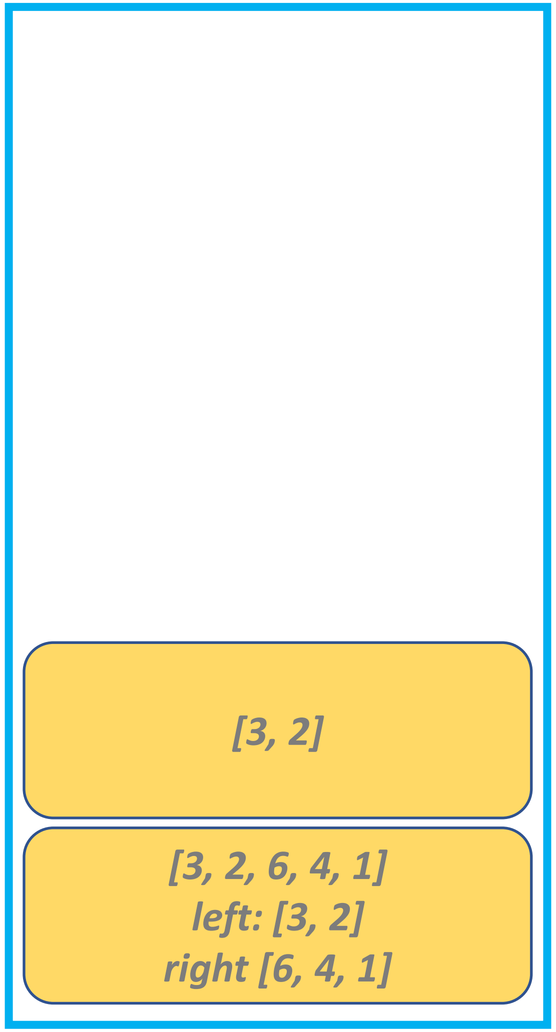
3	2	6	4	1

sort(data):

if data size == 1:
return

select middle item and split array (left and right)
sort(left side)
sort(right side)

merge arrays(left side and right side)



STACK

Merge Sort

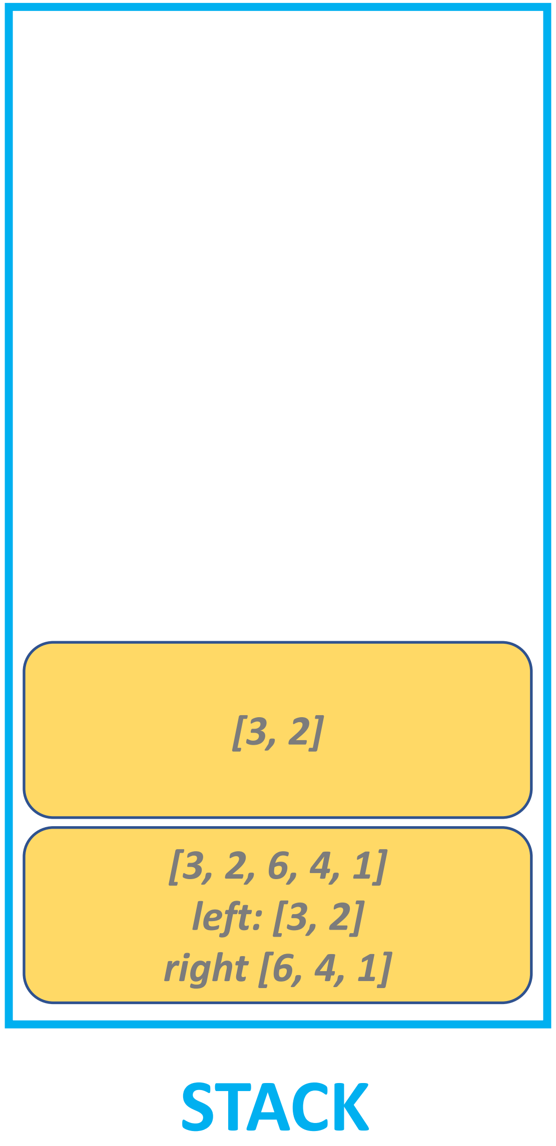
3	2	6	4	1

sort(data):

if data size == 1:
    return

select middle item and split array (left and right)
sort(left side)
sort(right side)

merge arrays(left side and right side)



Merge Sort

3	2	6	4	1

```
sort(data):

        if data size == 1:
            return

        select middle item and split array (left and right)
        sort(left side)
        sort(right side)

        merge arrays(left side and right side)
```

[3, 2]
left: [3]
right: [2]

[3, 2, 6, 4, 1]
left: [3, 2]
right [6, 4, 1]

STACK

Merge Sort

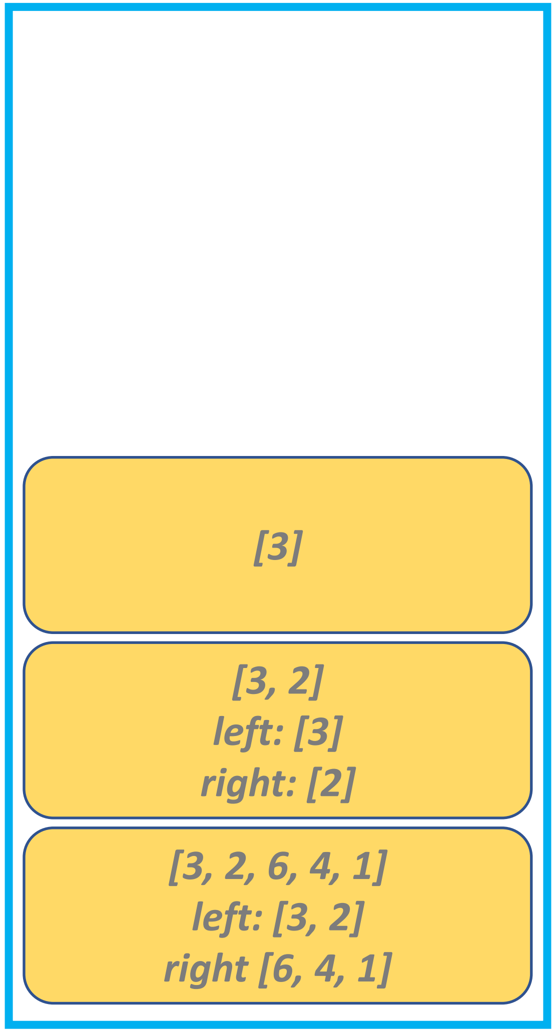
3	2	6	4	1

sort(data):

if data size == 1:
return

select middle item and split array (left and right)
sort(left side)
sort(right side)

merge arrays(left side and right side)

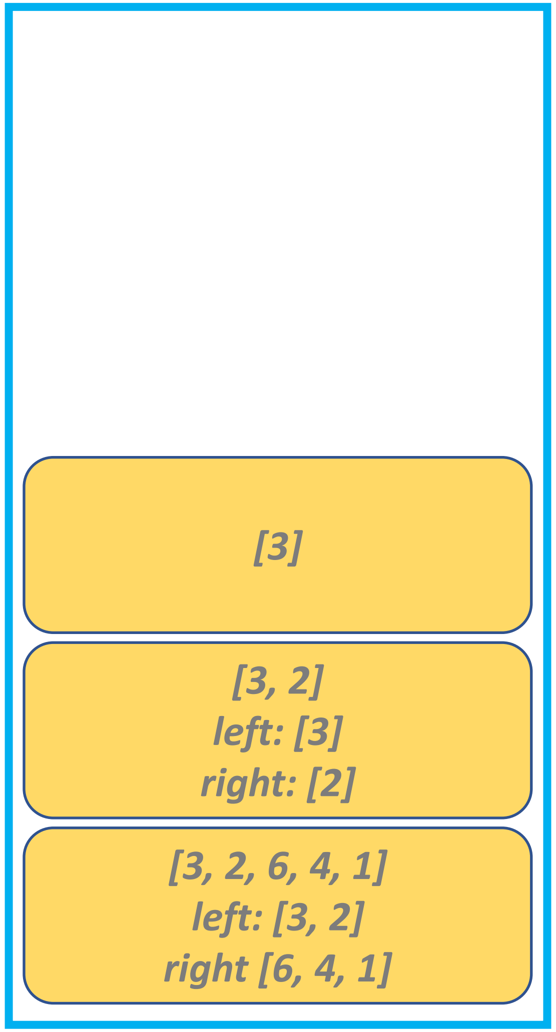


STACK

Merge Sort

3	2	6	4	1

```
sort(data):

    if data size == 1:
        return

    select middle item and split array (left and right)
    sort(left side)
    sort(right side)

    merge arrays(left side and right side)
```

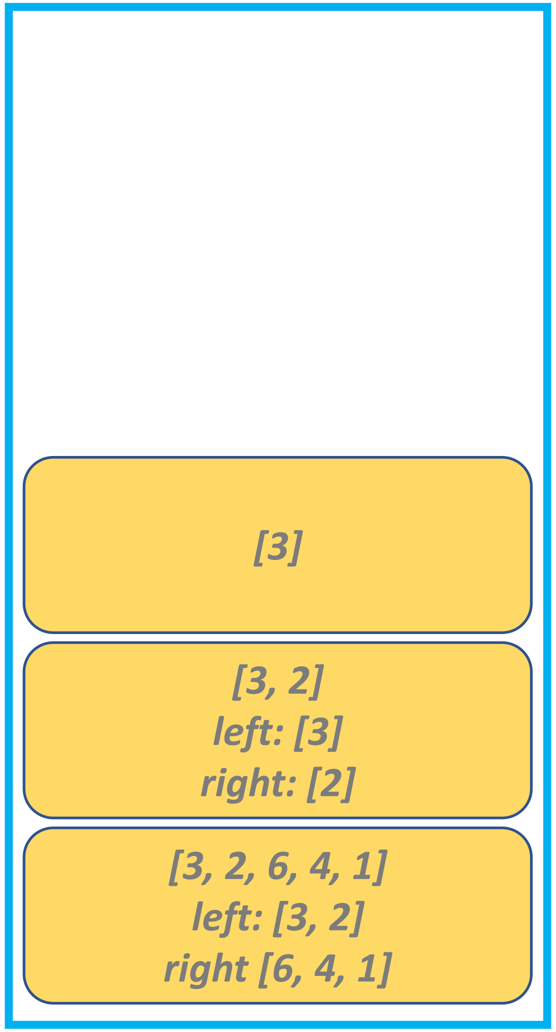


STACK

Merge Sort

3	2	6	4	1

```
sort(data):

        if data size == 1:
            return

        select middle item and split array (left and right)
        sort(left side)
        sort(right side)

        merge arrays(left side and right side)
```



STACK

Merge Sort

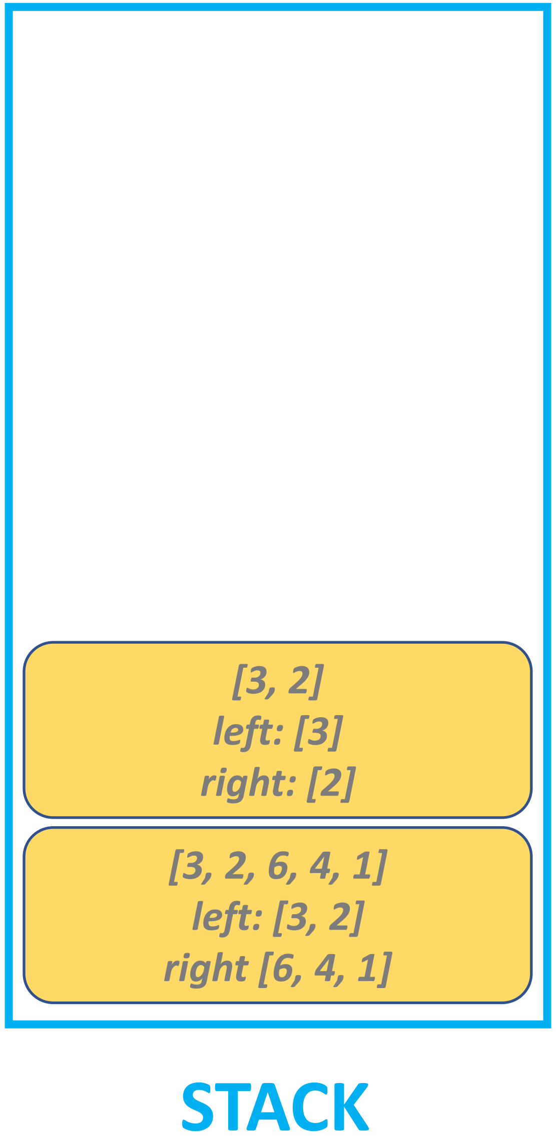
3	2	6	4	1

sort(data):

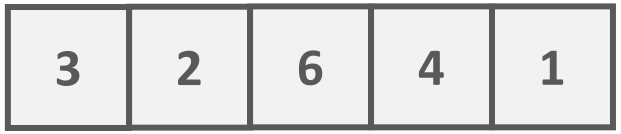
if data size == 1:
return

select middle item and split array (left and right)
sort(left side)
sort(right side)

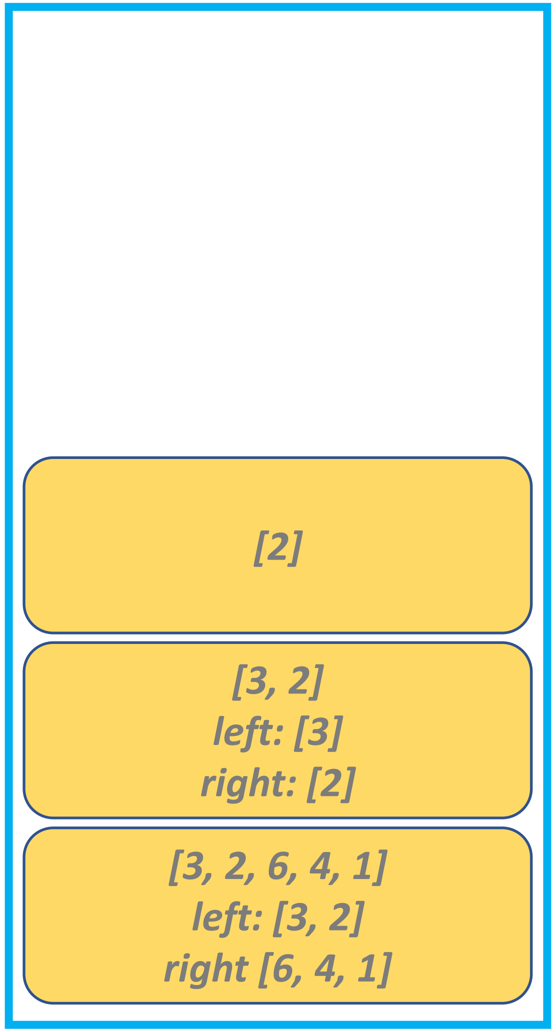
merge arrays(left side and right side)



Merge Sort




```
sort(data):

        if data size == 1:
            return

        select middle item and split array (left and right)
        sort(left side)
        sort(right side)

        merge arrays(left side and right side)
```



STACK

Merge Sort

3	2	6	4	1

```
sort(data):

    if data size == 1:
        return

    select middle item and split array (left and right)
    sort(left side)
    sort(right side)

    merge arrays(left side and right side)
```

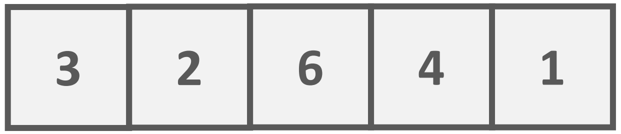
[2]

[3, 2]
left: [3]
right: [2]

[3, 2, 6, 4, 1]
left: [3, 2]
right [6, 4, 1]

STACK

Merge Sort

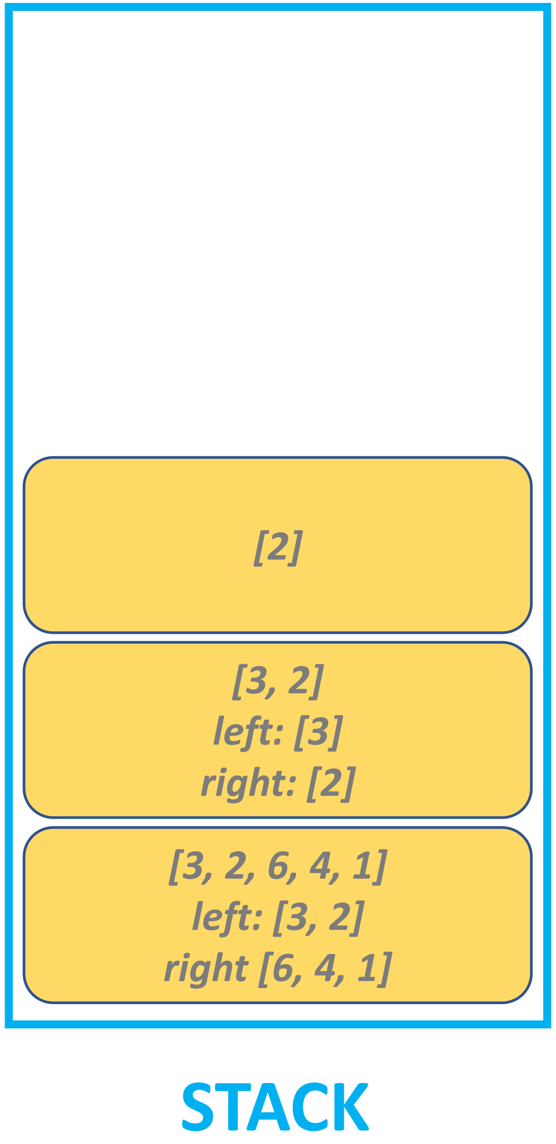


sort(data):

if data size == 1:
return

select middle item and split array (left and right)
sort(left side)
sort(right side)

merge arrays(left side and right side)



Merge Sort

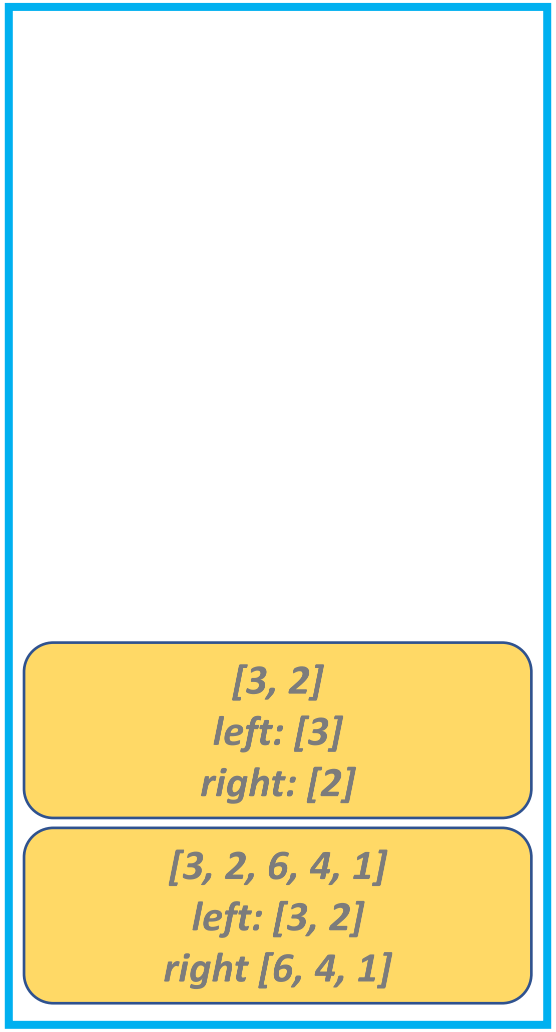
3	2	6	4	1

sort(data):

if data size == 1:
return

select middle item and split array (left and right)
sort(left side)
sort(right side)

merge arrays(left side and right side)



STACK

Merge Sort

3	2	6	4	1

sort(data):

if data size == 1:
return

select middle item and split array (left and right)
sort(left side)
sort(right side)

merge arrays(left side and right side)

[2, 3]

[3, 2, 6, 4, 1]
left: [3, 2]
right [6, 4, 1]

STACK

Merge Sort

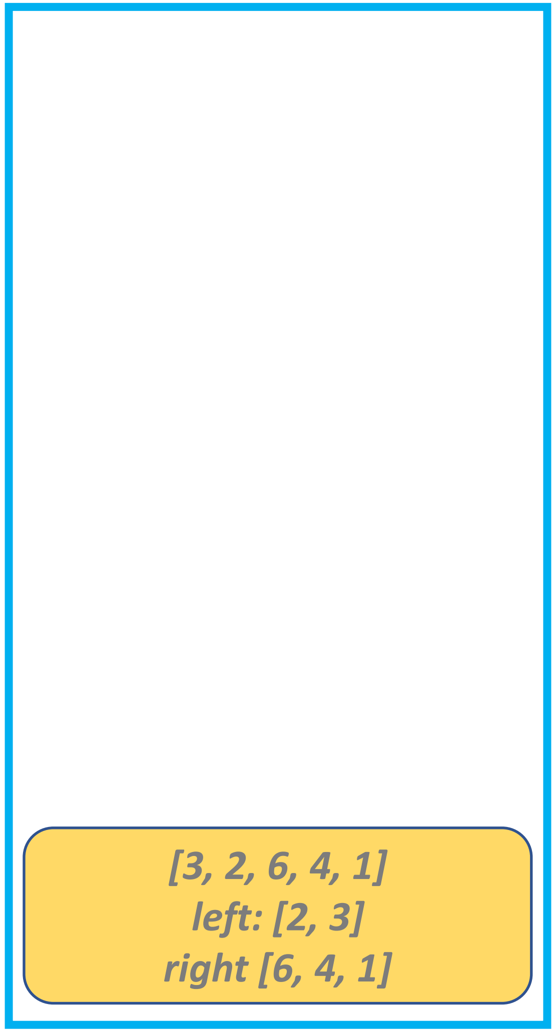
3	2	6	4	1

sort(data):

if data size == 1:
return

select middle item and split array (left and right)
sort(left side)
sort(right side)

merge arrays(left side and right side)

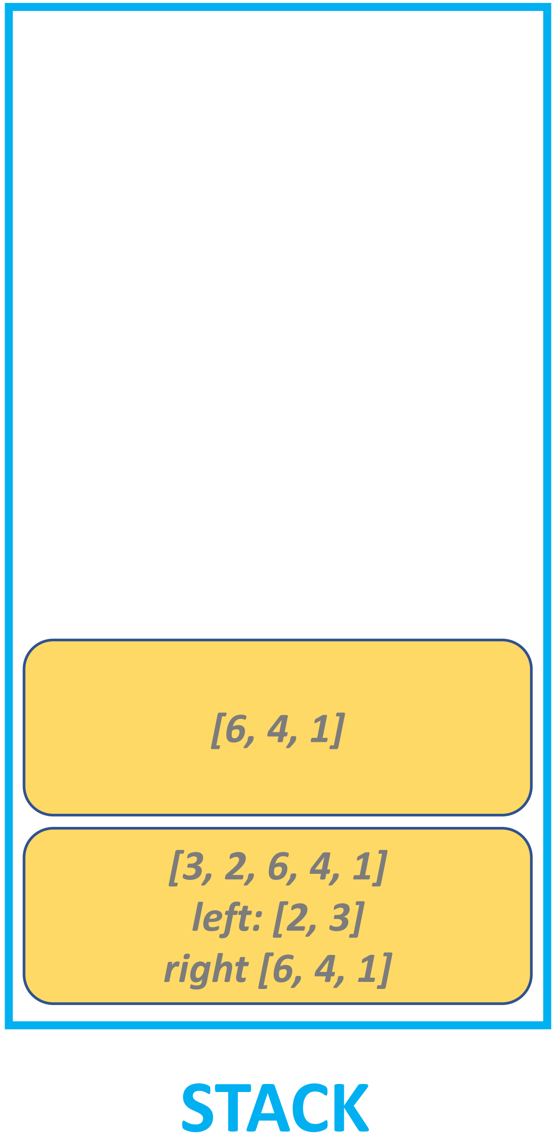


STACK

Merge Sort

3	2	6	4	1

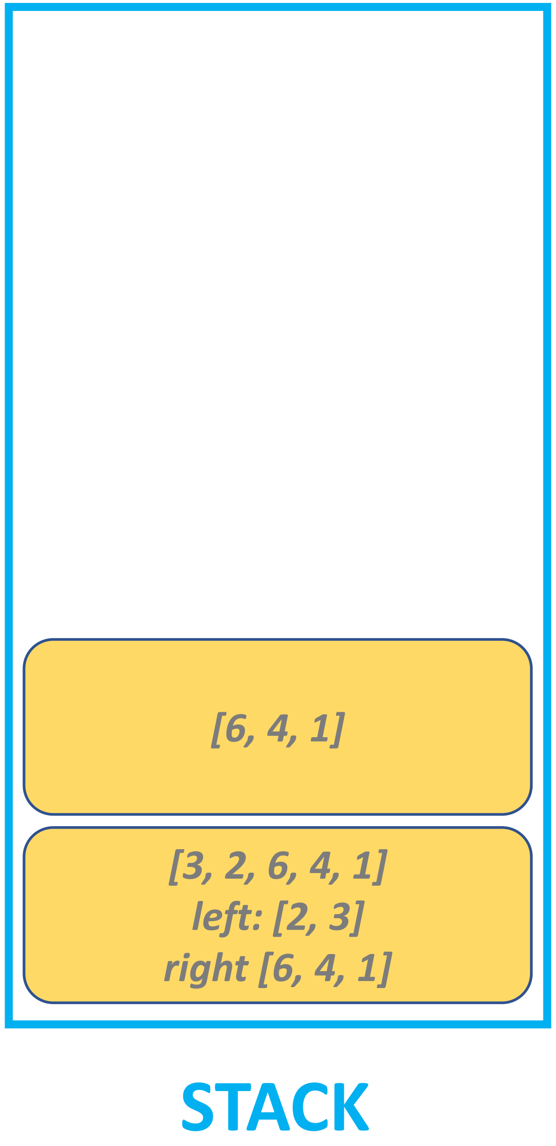
```
sort(data):

    if data size == 1:
        return

    select middle item and split array (left and right)
    sort(left side)
    sort(right side)

    merge arrays(left side and right side)
```



Merge Sort

3	2	6	4	1

```
sort(data):

        if data size == 1:
            return

        select middle item and split array (left and right)
        sort(left side)
        sort(right side)

        merge arrays(left side and right side)
```



Merge Sort

3	2	6	4	1

```
sort(data):

        if data size == 1:
            return

        select middle item and split array (left and right)
        sort(left side)
        sort(right side)

        merge arrays(left side and right side)
```

[6, 4, 1]
left: [6]
right: [4, 1]

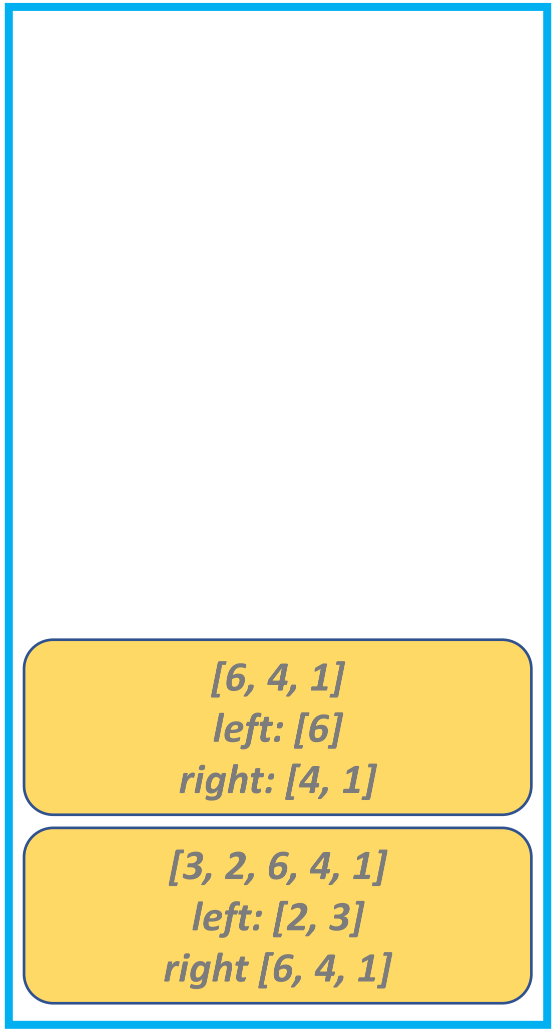
[3, 2, 6, 4, 1]
left: [2, 3]
right [6, 4, 1]

STACK

Merge Sort

3	2	6	4	1

```
sort(data):

        if data size == 1:
            return

        select middle item and split array (left and right)
        sort(left side)
        sort(right side)

        merge arrays(left side and right side)
```



STACK

Merge Sort

3	2	6	4	1

sort(data):

if data size == 1:
return

select middle item and split array (left and right)
sort(left side)
sort(right side)

merge arrays(left side and right side)

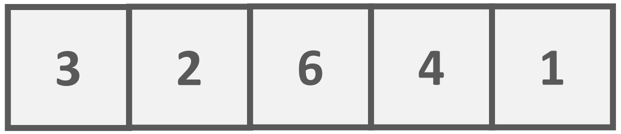
[6]

[6, 4, 1]
left: [6]
right: [4, 1]

[3, 2, 6, 4, 1]
left: [2, 3]
right [6, 4, 1]

STACK

Merge Sort

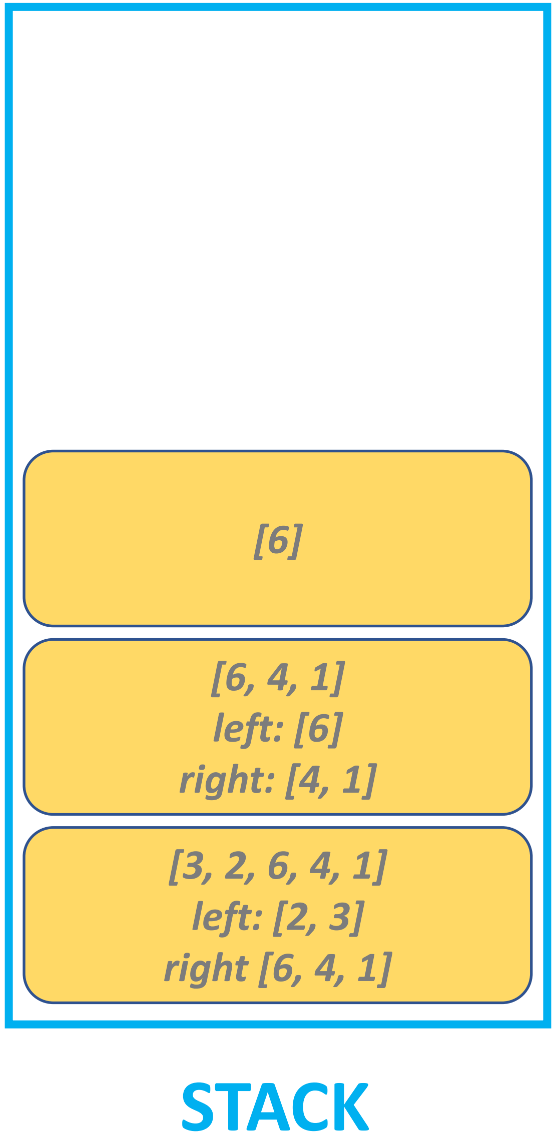


sort(data):

if data size == 1:
return

select middle item and split array (left and right)
sort(left side)
sort(right side)

merge arrays(left side and right side)



Merge Sort

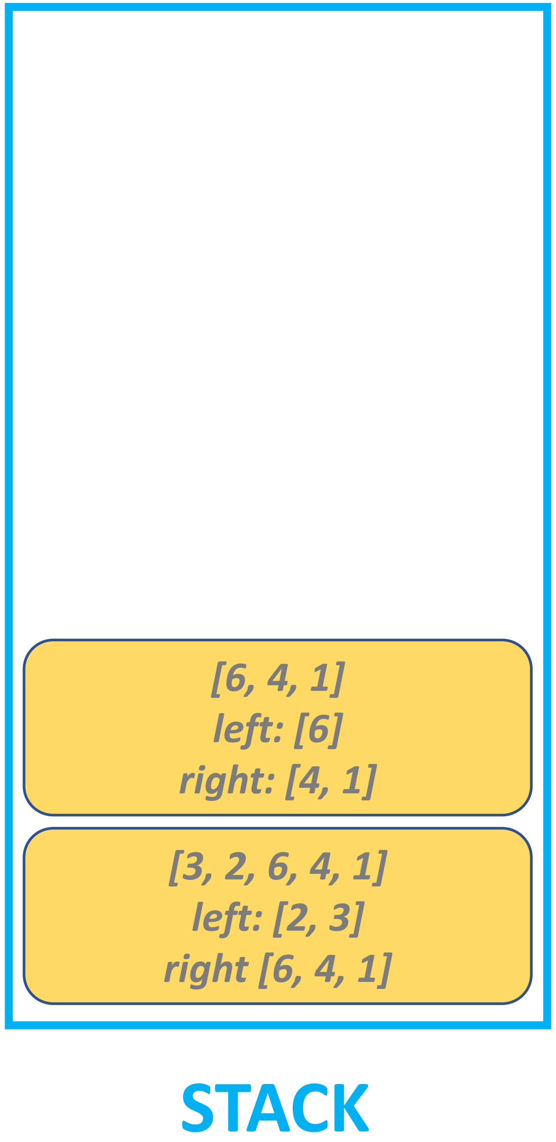
3	2	6	4	1

sort(data):

if data size == 1:
return

select middle item and split array (left and right)
sort(left side)
sort(right side)

merge arrays(left side and right side)



Merge Sort

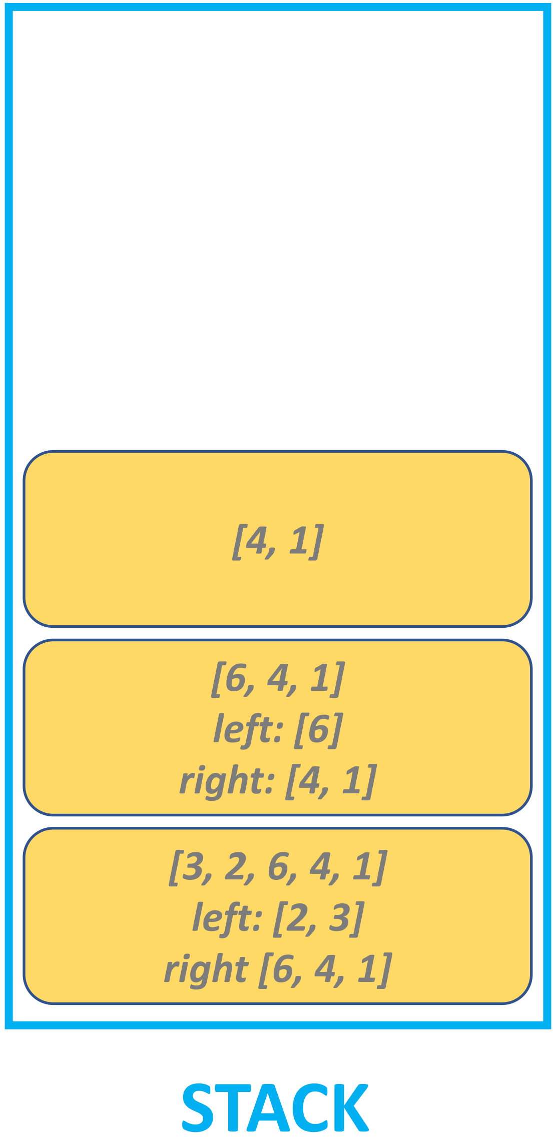
3	2	6	4	1

sort(data):

if data size == 1:
return

select middle item and split array (left and right)
sort(left side)
sort(right side)

merge arrays(left side and right side)



Merge Sort

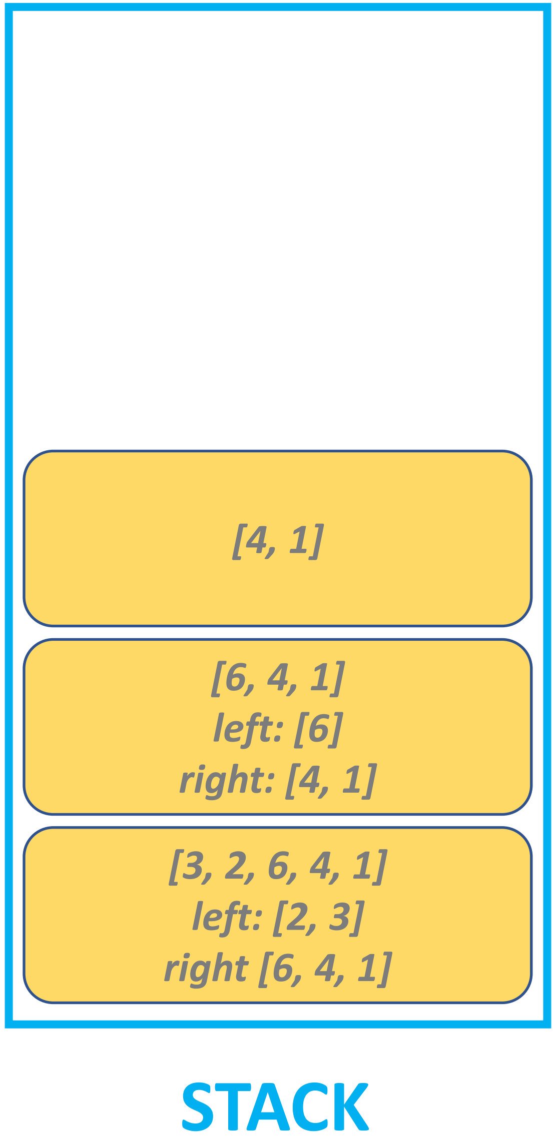
3	2	6	4	1

sort(data):

if data size == 1:
return

select middle item and split array (left and right)
sort(left side)
sort(right side)

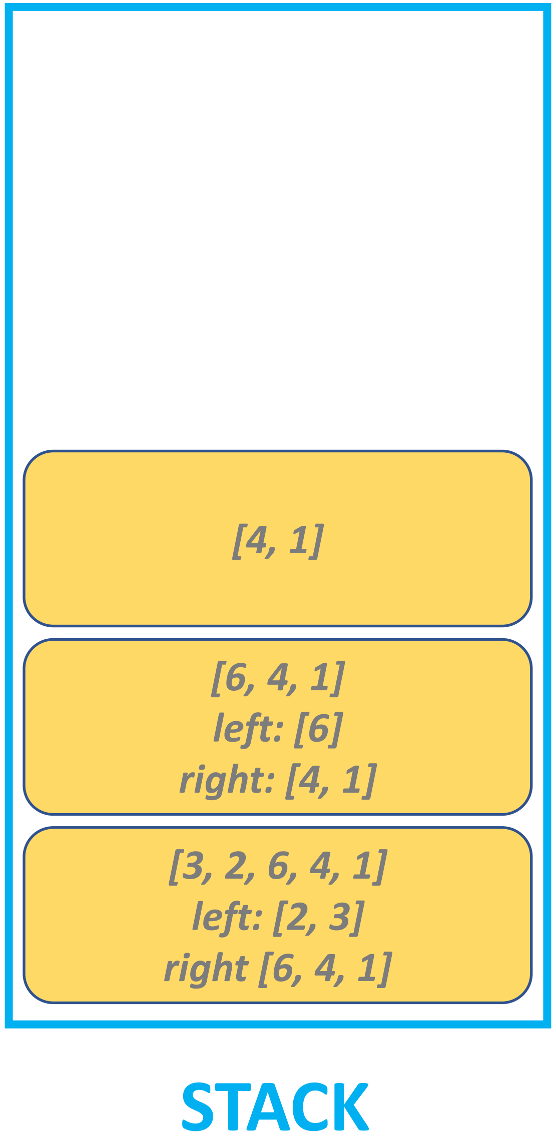
merge arrays(left side and right side)



Merge Sort

3	2	6	4	1

```
sort(data):

    if data size == 1:
        return

    select middle item and split array (left and right)
    sort(left side)
    sort(right side)

    merge arrays(left side and right side)
```



Merge Sort

3	2	6	4	1

```
sort(data):

        if data size == 1:
            return

        select middle item and split array (left and right)
        sort(left side)
        sort(right side)

        merge arrays(left side and right side)
```

[4, 1]
left: [4]
right: [1]

[6, 4, 1]
left: [6]
right: [4, 1]

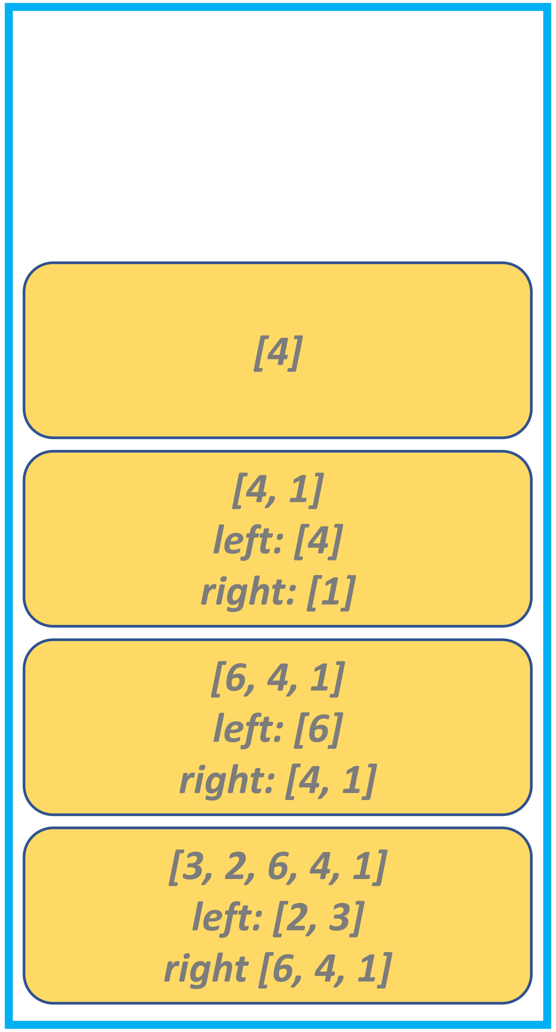
[3, 2, 6, 4, 1]
left: [2, 3]
right [6, 4, 1]

STACK

Merge Sort

3	2	6	4	1

```
sort(data):

    if data size == 1:
        return

    select middle item and split array (left and right)
    sort(left side)
    sort(right side)

    merge arrays(left side and right side)
```



STACK

Merge Sort

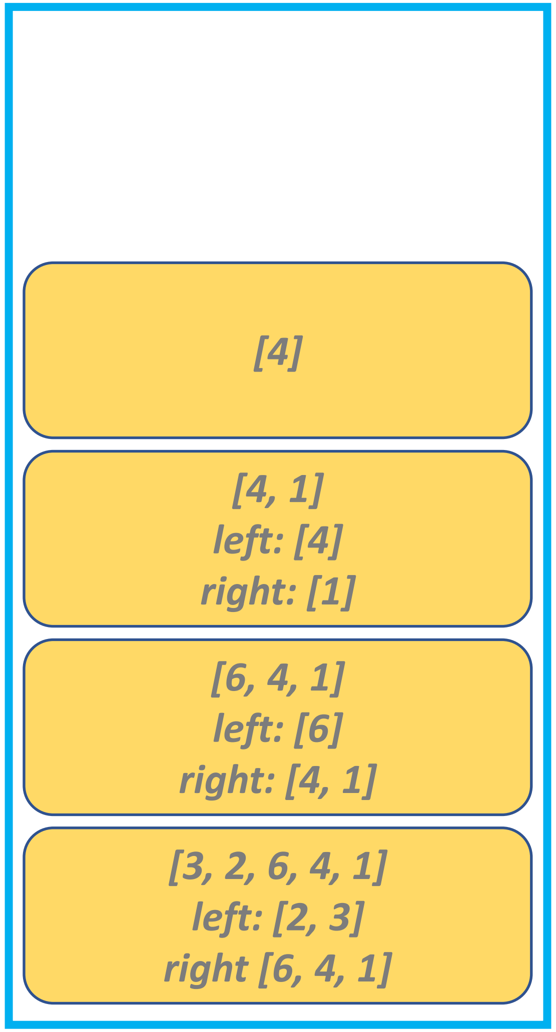
3	2	6	4	1

sort(data):

if data size == 1:
return

select middle item and split array (left and right)
sort(left side)
sort(right side)

merge arrays(left side and right side)



STACK

Merge Sort

3	2	6	4	1

```
sort(data):

        if data size == 1:
            return

        select middle item and split array (left and right)
        sort(left side)
        sort(right side)

        merge arrays(left side and right side)
```

[4]

[4, 1]
left: [4]
right: [1]

[6, 4, 1]
left: [6]
right: [4, 1]

[3, 2, 6, 4, 1]
left: [2, 3]
right [6, 4, 1]

STACK

Merge Sort

3	2	6	4	1

```
sort(data):

        if data size == 1:
            return

        select middle item and split array (left and right)
        sort(left side)
        sort(right side)

        merge arrays(left side and right side)
```

[4, 1]
left: [4]
right: [1]

[6, 4, 1]
left: [6]
right: [4, 1]

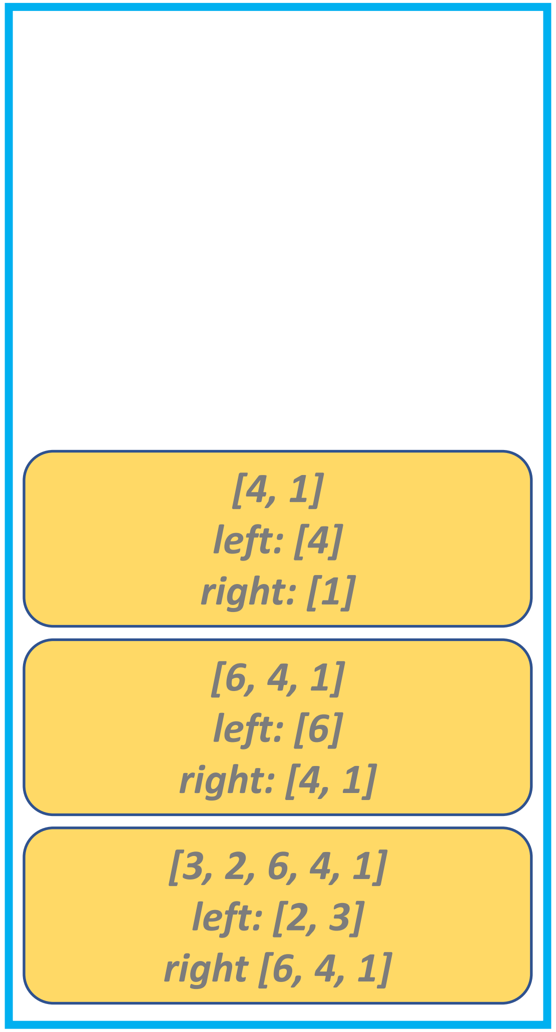
[3, 2, 6, 4, 1]
left: [2, 3]
right [6, 4, 1]

STACK

Merge Sort

3	2	6	4	1

```
sort(data):

    if data size == 1:
        return

    select middle item and split array (left and right)
    sort(left side)
    sort(right side)

    merge arrays(left side and right side)
```



STACK

Merge Sort

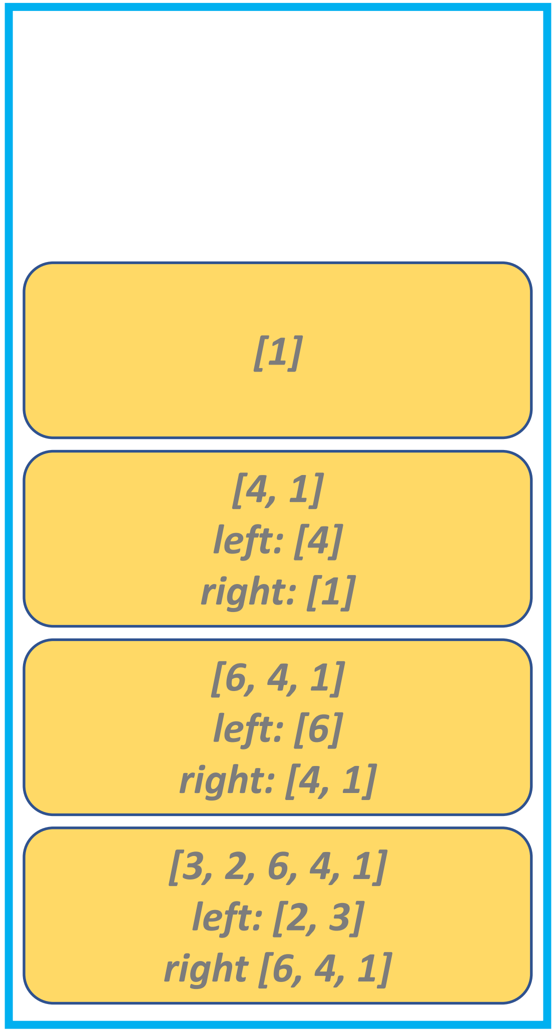
3	2	6	4	1

sort(data):

if data size == 1:
return

select middle item and split array (left and right)
sort(left side)
sort(right side)

merge arrays(left side and right side)



STACK

Merge Sort

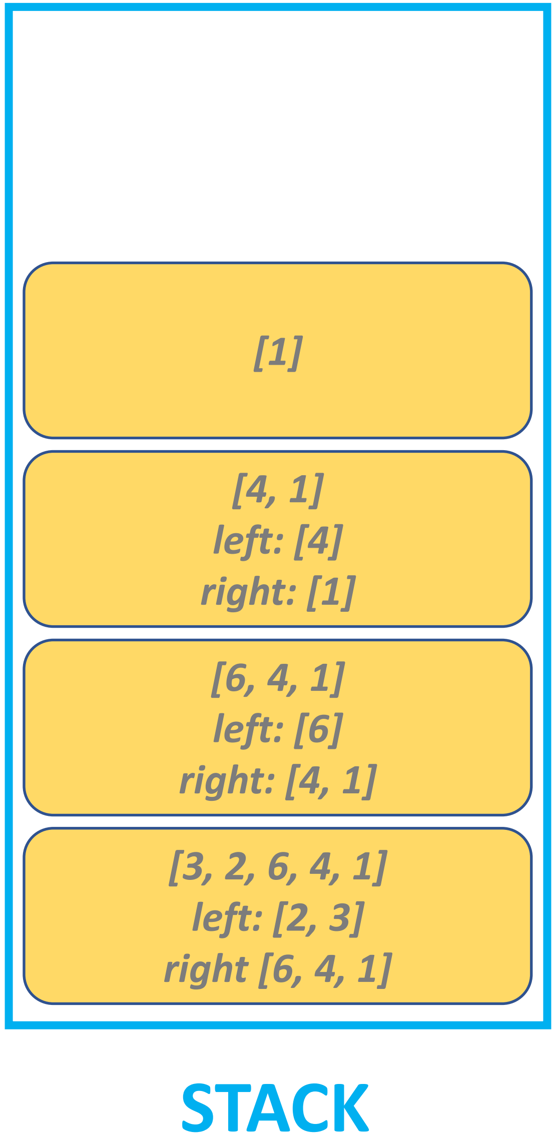
3	2	6	4	1

sort(data):

    if data size == 1:
        return

    select middle item and split array (left and right)
    sort(left side)
    sort(right side)

    merge arrays(left side and right side)



Merge Sort

3	2	6	4	1

```
sort(data):

    if data size == 1:
        return

    select middle item and split array (left and right)
    sort(left side)
    sort(right side)

    merge arrays(left side and right side)
```

[1]

[4, 1]
left: [4]
right: [1]

[6, 4, 1]
left: [6]
right: [4, 1]

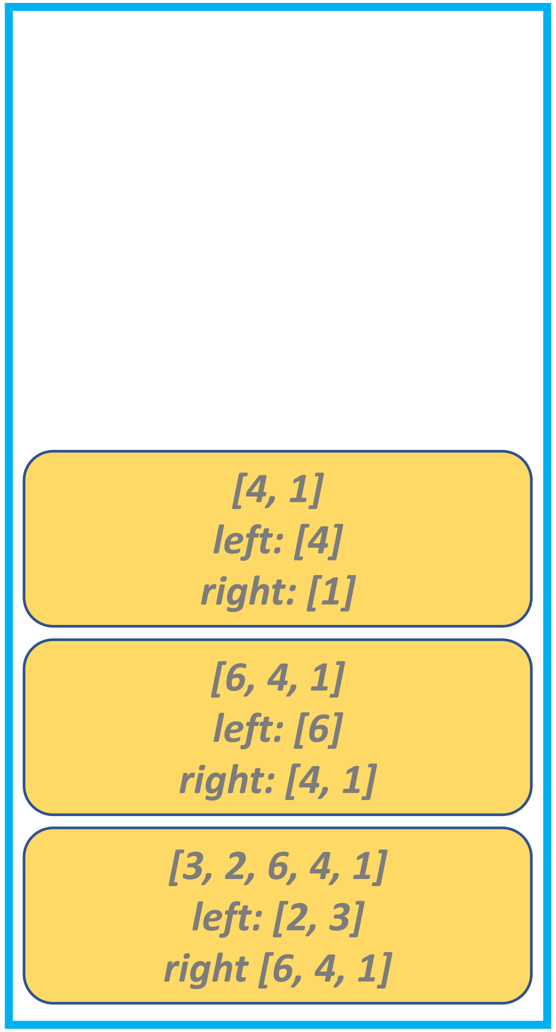
[3, 2, 6, 4, 1]
left: [2, 3]
right [6, 4, 1]

STACK

Merge Sort

3	2	6	4	1

```
sort(data):

        if data size == 1:
            return

        select middle item and split array (left and right)
        sort(left side)
        sort(right side)

        merge arrays(left side and right side)
```



STACK

Merge Sort

3	2	6	4	1

```
sort(data):

    if data size == 1:
        return

    select middle item and split array (left and right)
    sort(left side)
    sort(right side)

    merge arrays(left side and right side)
```

[4, 1]
left: [4]
right: [1]

[6, 4, 1]
left: [6]
right: [4, 1]

[3, 2, 6, 4, 1]
left: [2, 3]
right [6, 4, 1]

STACK

Merge Sort

3	2	6	4	1

sort(data):

if data size == 1:
return

select middle item and split array (left and right)
sort(left side)
sort(right side)

merge arrays(left side and right side)

[1, 4]

[6, 4, 1]
left: [6]
right: [4, 1]

[3, 2, 6, 4, 1]
left: [2, 3]
right [6, 4, 1]

STACK

Merge Sort

3	2	6	4	1

```
sort(data):

        if data size == 1:
            return

        select middle item and split array (left and right)
        sort(left side)
        sort(right side)

        merge arrays(left side and right side)
```

[1, 4]

[6, 4, 1]
left: [6]
right: [4, 1]

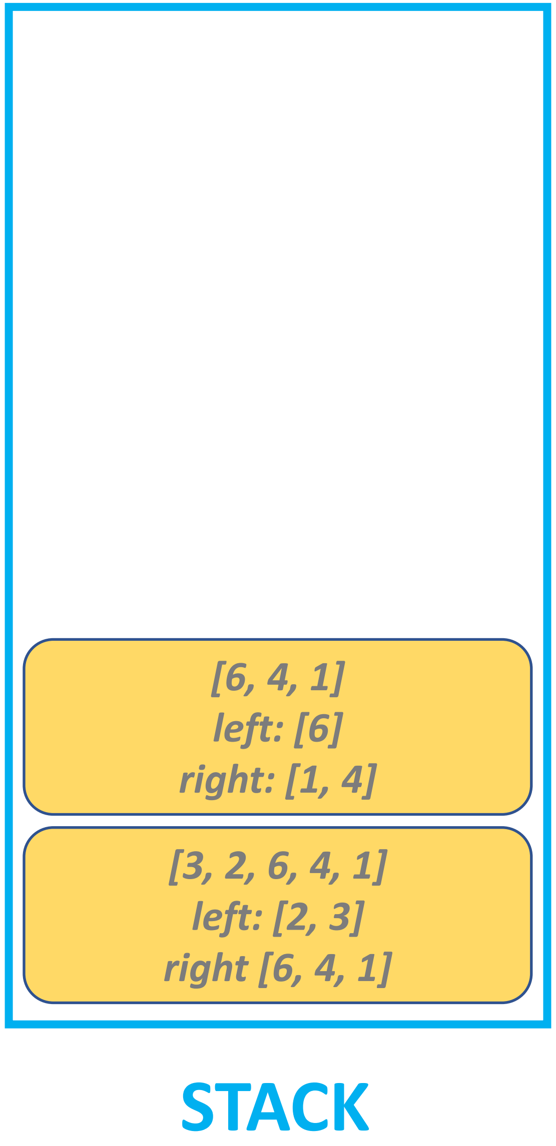
[3, 2, 6, 4, 1]
left: [2, 3]
right [6, 4, 1]

STACK

Merge Sort

3	2	6	4	1

```
sort(data):

    if data size == 1:
        return

    select middle item and split array (left and right)
    sort(left side)
    sort(right side)

    merge arrays(left side and right side)
```



Merge Sort

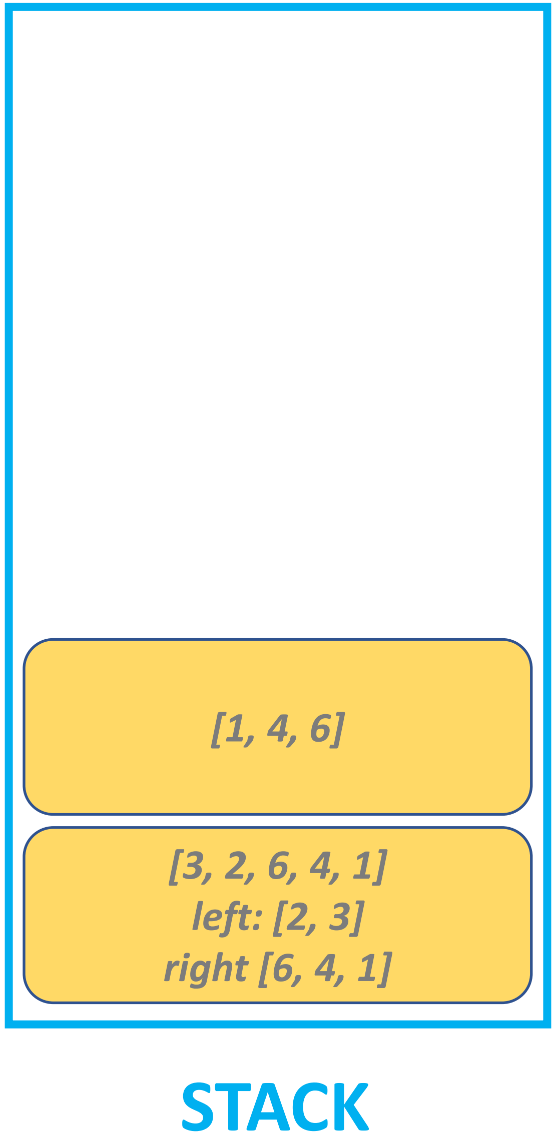
3	2	6	4	1

sort(data):

if data size == 1:
return

select middle item and split array (left and right)
sort(left side)
sort(right side)

merge arrays(left side and right side)



Merge Sort

3	2	6	4	1

```
sort(data):

    if data size == 1:
        return

    select middle item and split array (left and right)
    sort(left side)
    sort(right side)

    merge arrays(left side and right side)
```

[3, 2, 6, 4, 1]
left: [2, 3]
right [1, 4, 6]

STACK

Merge Sort

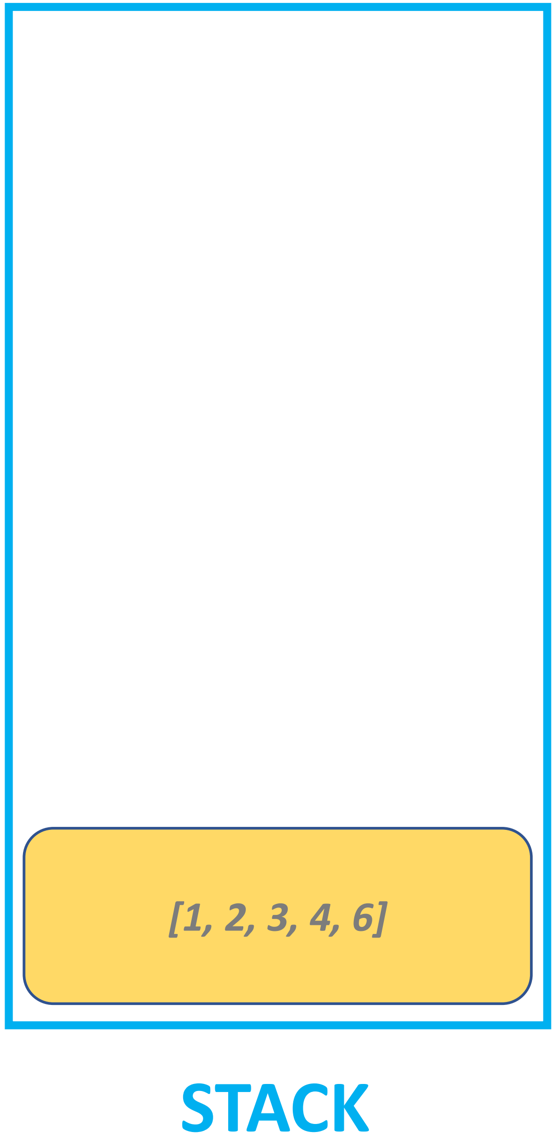
3	2	6	4	1

sort(data):

if data size == 1:
return

select middle item and split array (left and right)
sort(left side)
sort(right side)

merge arrays(left side and right side)



Non-Comparison Based Sorting

(Algorithms and Data Structures)

Comparison Based Sorting

What does comparison based sorting mean?

if nums[i] > nums[j]
swap(i,j)

We keep comparing items (strings, characters, doubles ...)
~ keep making decisions according to these comparisons

RESULT: we have to make at least $\log_2 n!$ comparisons to sort an array
that can be reduced to **O(NlogN)** with *Stirling-formula*

Stirling formula yields: $\Omega(N \log N)$
This is a lower bound, we are not able to
do any better if we use comparisons !!!

Non-Comparison Based Sorting

Can we do better? **YES**, the solution is not to use comparisons

There are simpler algorithms that can sort a list using partial information about the keys (items)

FOR EXAMPLE: **radix sort** or **bucket sort**

Counting Sort
(Algorithms and Data Structures)

Counting Sort

- it operates by counting the number of objects that have each **distinct key** value
- counting sort is an **integer sorting algorithm** - we assume the values to be integers
- it uses arithmetic on those counts to determine the positions of each key value in the output sequence
- it is only suitable for direct use in situations where the variation in keys is not significantly greater than the number of items
- it can be used as a subroutine in **radix sort**
- because counting sort uses key values as indexes into an array: it is not a comparison based sorting algorithm – so we can achieve **O(N)** linear running time

Counting Sort

- counting sort has **O(N+k)** linear running time complexity
- **N** is the number of items we want to sort
- **k** is the difference between the maximum and minimum key values – so basically the number of possible keys
- **CONCLUSION:** it is only suitable for direct use in situations where the variation in keys is not significantly greater than the number of items

Counting Sort

1	4	1	7	1	7	10	3

Let's allocate memory for an array size **k**, we want to track and count that how many occurances are there in the original array for the given key

1.) **i**terate through the original array **O(N)**
2.) the value in the array will be the index of the temporary array: we increment the counter there
3.) traverse the array of counters (array size **k**) and print out the values **O(k)**
4.) it is going to yield the numerical ordering

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9

the so-called
count array *with*
as many items as the ***radix***
*(**10** or **max-min+1**)*

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min+1**)*

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0

the so-called
count array *with*
as many items as the ***radix***
*(**10** or **max-min+1**)*

Let's allocate memory for an array size **k**, we want to track and count that how many occurances are there in the original array for the given key

1.) iterate through the original array **O(N)**
2.) the value in the array will be the index of the temporary array: we increment the counter there

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min+1**)*

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	1	0	0	0	0	0	0	0	0

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min+1**)*

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	1	0	0	0	0	0	0	0	0

the so-called
count array *with*
as many items as the ***radix***
*(**10** or **max-min+1**)*

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	1	0	0	1	0	0	0	0	0

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min+1**)*

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	1	0	0	1	0	0	0	0	0

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min+1**)*

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	2	0	0	1	0	0	0	0	0

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min+1**)*

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	2	0	0	1	0	0	0	0	0

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min+1**)*

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	2	0	0	1	0	0	1	0	0

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min+1**)*

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	2	0	0	1	0	0	1	0	0

the so-called
count array *with*
as many items as the ***radix***
*(**10** or **max-min+1**)*

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	0	1	0	0	1	0	0

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min+1)***

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	0	1	0	0	2	0	0

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min+1**)*

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	0	1	0	0	2	0	1

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min+1**)*

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	0	2	0	1

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min+1**)*

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	0	2	1	1

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min+1**)*

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	0	2	1	2

the so-called
count array *with*
as many items as the ***radix***
*(**10** or **max-min+1**)*

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min+1**)*

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min+1**)*

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min+1**)*

THERE IS A PROBLEM WITH THIS REPRESENTATION

we have to transform the
***count array** to know the postions of the items*
*in the **final sorted array** – this is why to construct the*
cumulative count array

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2

the so-called
count array *with*
as many items as the ***radix***
*(**10** or **max-min+1**)*

0	1	2	3	4	5	6	7	8	9	10
1	1	1	3	4	6	7	7	8	9	9

we see that we have the value ***7*** *two times*
but ***what are their indexes in the sorted order?***

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2

the so-called ***count array*** *with as many items as the* ***radix*** *(**10** or **max-min+1**)*

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	**3**	**0**	**1**	**1**	**0**	**1**	**2**	**1**	**2**

0									

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min+1**)*

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2

0	3								

the so-called
count array *with*
as many items as the ***radix***
*(**10** or **max-min+1**)*

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2
0	3	3							

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min+1**)*

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2
0	3	3	4						

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min+1**)*

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2
0	3	3	4	5					

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min+1**)*

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2
0	3	3	4	5	5				

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min+1**)*

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2
0	3	3	4	5	5	6			

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min+1**)*

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2
0	3	3	4	5	5	6	8		

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min+1**)*

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2
0	3	3	4	5	5	6	8	9	

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min+1**)*

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2
0	3	3	4	5	5	6	8	9	11

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min+1**)*

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2
0	3	3	4	5	5	6	8	9	11

the so-called
count array *with*
as many items as the ***radix***
*(**10** or **max-min+1**)*

THIS IS A GOOD REPRESENTATION

*the values in the **cumulative array** have*
something to do with their final positions in the sorted order

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2
0	3	3	4	5	5	6	8	9	11

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min+1**)*

0	1	2	3	4	5	6	7	8	9	10

output array

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

WE ARE AFTER A STABLE SORTING ALGORITHM
*(because it is crucial in **radix sort**)*

*we could consider the items from **left to right** and we get the sorted order **but it is not a stable approach***

*if we want to **guarantee stability:** we have to consdier the Items from **right to left***

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2
0	3	3	4	5	5	6	8	9	11

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min-1**)*

0	1	2	3	4	5	6	7	8	9	10

output array

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2
0	3	3	4	5	5	5	8	9	11

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min-1**)*

0	1	2	3	4	5	6	7	8	9	10

output array

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2
0	3	3	4	5	5	5	8	9	11

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min-1**)*

0	1	2	3	4	5	6	7	8	9	10
					6					

output array

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2
0	3	3	4	5	5	5	8	9	11

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min-1**)*

0	1	2	3	4	5	6	7	8	9	10
					6					

output array

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2

0	3	3	4	5	5	5	8	9	10

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min-1**)*

0	1	2	3	4	5	6	7	8	9	10
					6					

output array

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2
0	3	3	4	5	5	5	8	9	10

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min-1**)*

0	1	2	3	4	5	6	7	8	9	10
					6					9

output array

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2
0	3	3	4	5	5	5	8	9	10

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min-1**)*

0	1	2	3	4	5	6	7	8	9	10
					6					9

output array

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2
0	3	3	4	5	5	5	8	8	10

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min-1**)*

0	1	2	3	4	5	6	7	8	9	10
					6					9

output array

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

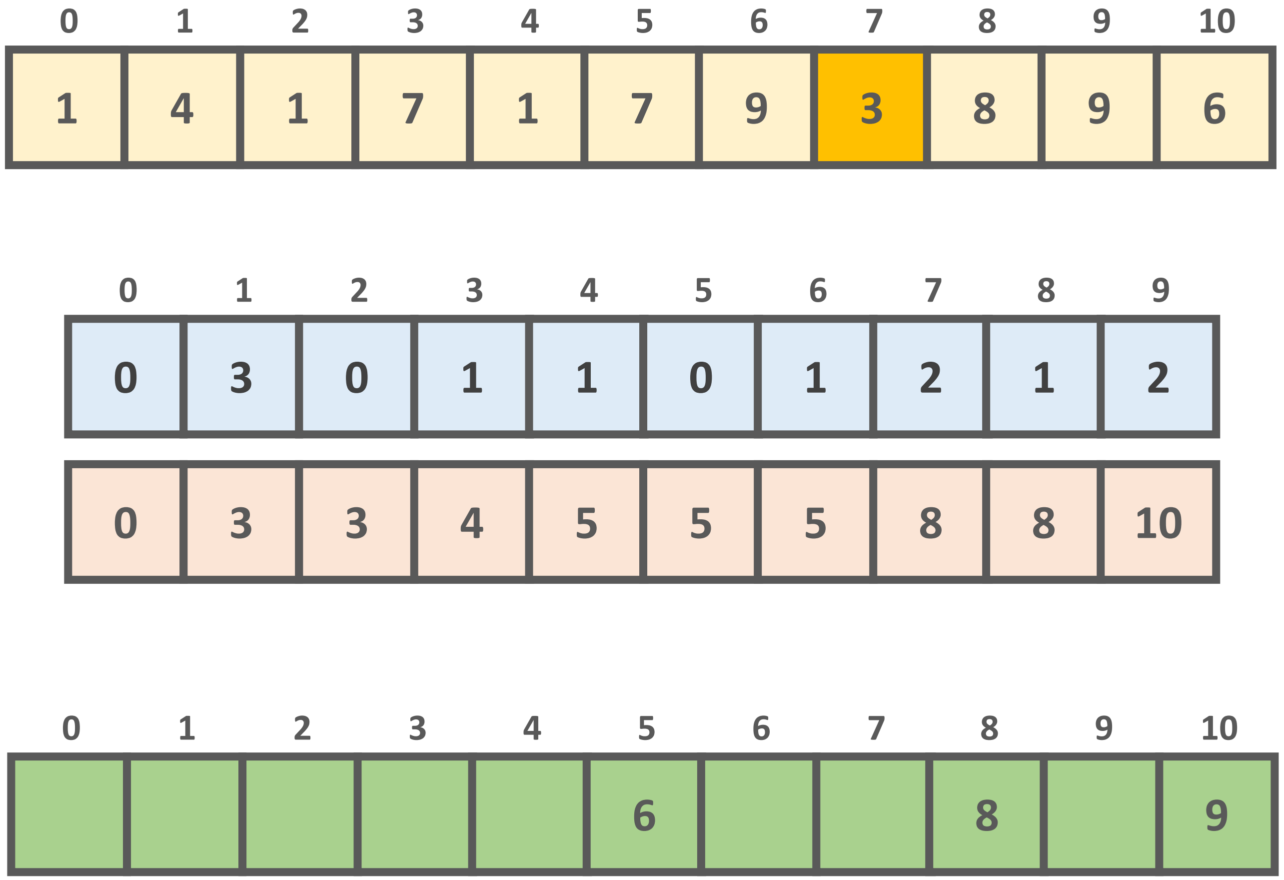
0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2
0	3	3	4	5	5	5	8	8	10

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min-1**)*

0	1	2	3	4	5	6	7	8	9	10
					6			8		9

output array

Counting Sort



the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min-1)***

output array

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2
0	3	3	3	5	5	5	8	8	10

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min-1**)*

0	1	2	3	4	5	6	7	8	9	10
					6			8		9

output array

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2
0	3	3	3	5	5	5	8	8	10

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min-1)***

0	1	2	3	4	5	6	7	8	9	10
			3		6			8		9

output array

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2
0	3	3	3	5	5	5	8	8	10

the so-called
count array *with*
as many items as the ***radix***
*(**10** or **max-min-1**)*

0	1	2	3	4	5	6	7	8	9	10
			3		6			8		9

output array

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2
0	3	3	3	5	5	5	8	8	9

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min-1**)*

0	1	2	3	4	5	6	7	8	9	10
			3		6			8		9

output array

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2
0	3	3	3	5	5	5	8	8	9

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min-1**)*

0	1	2	3	4	5	6	7	8	9	10
			3		6			8	9	9

output array

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2
0	3	3	3	5	5	5	8	8	9

the so-called
count array *with*
as many items as the ***radix***
*(**10** or **max-min-1**)*

0	1	2	3	4	5	6	7	8	9	10
			3		6			8	9	9

output array

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2
0	3	3	3	5	5	5	7	8	9

the so-called
count array *with*
as many items as the ***radix***
*(**10** or **max-min-1**)*

0	1	2	3	4	5	6	7	8	9	10
			3		6			8	9	9

output array

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2
0	3	3	3	5	5	5	7	8	9

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min-1)***

0	1	2	3	4	5	6	7	8	9	10
			3		6		7	8	9	9

output array

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2
0	3	3	3	5	5	5	7	8	9

*the so-called **count array** with as many items as the **radix** (**10** or **max-min-1**)*

0	1	2	3	4	5	6	7	8	9	10
			3		6		7	8	9	9

output array

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2
0	2	3	3	5	5	5	7	8	9

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min-1)***

0	1	2	3	4	5	6	7	8	9	10
			3		6		7	8	9	9

output array

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2
0	2	3	3	5	5	5	7	8	9

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min-1**)*

0	1	2	3	4	5	6	7	8	9	10
		1	3		6		7	8	9	9

output array

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2
0	2	3	3	5	5	5	7	8	9

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min-1**)*

0	1	2	3	4	5	6	7	8	9	10
		1	3		6		7	8	9	9

output array

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2
0	2	3	3	5	5	5	6	8	9

the so-called
count array *with*
as many items as the ***radix***
*(**10** or **max-min-1**)*

0	1	2	3	4	5	6	7	8	9	10
		1	3		6		7	8	9	9

output array

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2
0	2	3	3	5	5	5	6	8	9

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min-1**)*

0	1	2	3	4	5	6	7	8	9	10
		1	3		6	7	7	8	9	9

output array

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2

0	2	3	3	5	5	5	6	8	9

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min-1)***

0	1	2	3	4	5	6	7	8	9	10
		1	3		6	7	7	8	9	9

output array

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2

0	1	3	3	5	5	5	6	8	9

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min-1**)*

0	1	2	3	4	5	6	7	8	9	10
		1	3		6	7	7	8	9	9

output array

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2
0	1	3	3	5	5	5	6	8	9

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min-1**)*

0	1	2	3	4	5	6	7	8	9	10
	1	1	3		6	7	7	8	9	9

output array

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2
0	1	3	3	5	5	5	6	8	9

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min-1)***

0	1	2	3	4	5	6	7	8	9	10
	1	1	3		6	7	7	8	9	9

output array

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2
0	1	3	3	4	5	5	6	8	9

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min-1**)*

0	1	2	3	4	5	6	7	8	9	10
	1	1	3		6	7	7	8	9	9

output array

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2
0	1	3	3	4	5	5	6	8	9

the so-called ***count array*** *with as many items as the* ***radix*** *(****10*** *or* ***max-min-1****)*

0	1	2	3	4	5	6	7	8	9	10
	1	1	3	4	6	7	7	8	9	9

output array

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2
0	1	3	3	4	5	5	6	8	9

the so-called ***count array*** *with as many items as the* ***radix*** *(**10** or **max-min-1**)*

0	1	2	3	4	5	6	7	8	9	10
	1	1	3	4	6	7	7	8	9	9

output array

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2
0	0	3	3	4	5	5	6	8	9

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min-1**)*

0	1	2	3	4	5	6	7	8	9	10
	1	1	3	4	6	7	7	8	9	9

output array

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2

0	0	3	3	4	5	5	6	8	9

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min-1**)*

0	1	2	3	4	5	6	7	8	9	10
1	1	1	3	4	6	7	7	8	9	9

output array

Counting Sort

0	1	2	3	4	5	6	7	8	9	10
1	4	1	7	1	7	9	3	8	9	6

0	1	2	3	4	5	6	7	8	9
0	3	0	1	1	0	1	2	1	2
0	0	3	3	4	5	5	6	8	9

the so-called
***count array** with*
*as many items as the **radix***
*(**10** or **max-min-1**)*

0	1	2	3	4	5	6	7	8	9	10
1	1	1	3	4	6	7	7	8	9	9

output array

Radix Sort

(Algorithms and Data Structures)

What is the problem with counting sort?

- **counting sort** is working fine but th problem is that **k** may be way larger than **N**
- if that is the case than **O(N+k)** is not a good running time

1	4	2	1000

*in this case **k=1000** so the algorithm has to deal with*
***O(N+k)** so **1004** steps when there are*
*just **4** items we have to sort*

Radix Sort

- radix sort is another **non-comparison based** integer (string) sorting approach – the algorithm threats integers as a string of digits
- can be very efficient because there are no comparisons
- so even linear **O(N)** running time complexity can be reached
- we sort the elements according to individual characters
- radix sort is a **stable sorting** algorithm

Radix Sort

1	0	6	4	5	6	3	9
3	4	9	1	8	5	4	3
9	5	3	8	1	1	1	0
3	2	1	5	8	4	3	9
1	8	5	9	3	7	2	5
1	9	4	7	5	6	3	3

most significant
digit of the number

least significant
digit of the number

Radix Sort

most-significant-digit (MSD) first radix sort algorithm

least-significant-digit (LSD) first radix sort algorithm

- we sort the integer starting with the **MSD**
- the **first pass** would go a long way toward sorting the entire range
- each pass after that would simply handle the details
- implemented with **recursion** usually

- we sort the integer starting with the **LSD**
- in every iteration it uses counting sort to sort the integrers based on a given digit

LSD Radix Sort

- **least-significant-digit-first** string sorting
- it considers characters from right to left
- we can use it to fixed length strings or fixed length numbers for example integers
- it sorts the characters at the last column then keep going left and sort the columns **independently**
- typical interview question: *how to sort one million **32-bit** integers?*

LSD Radix Sort

- **most-significant-digit-first** string sorting
- it considers characters from left to right
- LDS radix sort is sensitive to **ASCII** and **Unicode** representations
- it has several advantages – **MSD** examines just enough characters to sort the key
- which means that **it can be sublinear in input size**
- **MSD** is not always fast because of the recursive function calls
- **SOLUTION:** we should combine it with quicksort – this is the **3-way radix quicksort** algorithm

LSD Radix Sort

00120

00450

43589

73141

31975

52455

60433

21271

BUCKETS

0:

1:

2:

3:

4:

5:

6:

7:

8:

9:

LSD Radix Sort

00120

00450

43589

73141

31975

52455

60433

21271

BUCKETS

0:

1:

2:

3:

4:

5:

6:

7:

8:

9:

LSD Radix Sort

00120

00450

43589

73141

31975

52455

60433

21271

BUCKETS

0: 00120

1:

2:

3:

4:

5:

6:

7:

8:

9:

LSD Radix Sort

00120

00450

43589

73141

31975

52455

60433

21271

BUCKETS

0: 00120, 00450

1:

2:

3:

4:

5:

6:

7:

8:

9:

LSD Radix Sort

00120

00450

43589

73141

31975

52455

60433

21271

BUCKETS

0: 00120, 00450

1:

2:

3:

4:

5:

6:

7:

8:

9: 43589

LSD Radix Sort

00120

00450

43589

73141

31975

52455

60433

21271

BUCKETS

0: 00120, 00450

1: 73141

2:

3:

4:

5:

6:

7:

8:

9: 43589

LSD Radix Sort

00120

00450

43589

73141

31975

52455

60433

21271

BUCKETS

0: 00120, 00450

1: 73141

2:

3:

4:

5: 31975

6:

7:

8:

9: 43589

LSD Radix Sort

00120

00450

43589

73141

31975

52455

60433

21271

BUCKETS

0: 00120, 00450

1: 73141

2:

3:

4:

5: 31975, 52455

6:

7:

8:

9: 43589

LSD Radix Sort

00120

00450

43589

73141

31975

52455

60433

21271

BUCKETS

0: 00120, 00450

1: 73141

2:

3: 60433

4:

5: 31975, 52455

6:

7:

8:

9: 43589

LSD Radix Sort

00120

00450

43589

73141

31975

52455

60433

21271

BUCKETS

0: 00120, 00450

1: 73141, 21271

2:

3: 60433

4:

5: 31975, 52455

6:

7:

8:

9: 43589

LSD Radix Sort

BUCKETS

0: 00120, 00450

1: 73141, 21271

2:

3: 60433

4:

5: 31975, 52455

6:

7:

8:

9: 43589

LSD Radix Sort

00120

BUCKETS

0: 00120, 00450

1: 73141, 21271

2:

3: 60433

4:

5: 31975, 52455

6:

7:

8:

9: 43589

LSD Radix Sort

00120

00450

BUCKETS

0: 00120, 00450

1: 73141, 21271

2:

3: 60433

4:

5: 31975, 52455

6:

7:

8:

9: 43589

LSD Radix Sort

00120

00450

73141

BUCKETS

0: 00120, 00450

1: 73141, 21271

2:

3: 60433

4:

5: 31975, 52455

6:

7:

8:

9: 43589

LSD Radix Sort

00120

00450

73141

21271

BUCKETS

0: 00120, 00450

1: 73141, 21271

2:

3: 60433

4:

5: 31975, 52455

6:

7:

8:

9: 43589

LSD Radix Sort

00120

00450

73141

21271

60433

BUCKETS

0: 00120, 00450

1: 73141, 21271

2:

3: 60433

4:

5: 31975, 52455

6:

7:

8:

9: 43589

LSD Radix Sort

00120

00450

73141

21271

60433

31975

BUCKETS

0: 00120, 00450

1: 73141, 21271

2:

3: 60433

4:

5: 31975, 52455

6:

7:

8:

9: 43589

LSD Radix Sort

00120

00450

73141

21271

60433

31975

52455

BUCKETS

0: 00120, 00450

1: 73141, 21271

2:

3: 60433

4:

5: 31975, 52455

6:

7:

8:

9: 43589

LSD Radix Sort

00120

00450

73141

21271

60433

31975

52455

43589

BUCKETS

0: 00120, 00450

1: 73141, 21271

2:

3: 60433

4:

5: 31975, 52455

6:

7:

8:

9: 43589

LSD Radix Sort

00120

00450

73141

21271

60433

31975

52455

43589

BUCKETS

0:

1:

2:

3:

4:

5:

6:

7:

8:

9:

LSD Radix Sort

00120

00450

73141

21271

60433

31975

52455

43589

BUCKETS

0:

1:

2:

3:

4:

5:

6:

7:

8:

9:

LSD Radix Sort

00120

00450

73141

21271

60433

31975

52455

43589

BUCKETS

0:

1:

2: 00120

3:

4:

5:

6:

7:

8:

9:

LSD Radix Sort

00120

00450

73141

21271

60433

31975

52455

43589

BUCKETS

0:

1:

2: 00120

3:

4:

5: 00450

6:

7:

8:

9:

LSD Radix Sort

00120

00450

73141

21271

60433

31975

52455

43589

BUCKETS

0:

1:

2: 00120

3:

4: 73141

5: 00450

6:

7:

8:

9:

LSD Radix Sort

00120

00450

73141

21271

60433

31975

52455

43589

BUCKETS

0:

1:

2: 00120

3:

4: 73141

5: 00450

6:

7: 21271

8:

9:

LSD Radix Sort

00120

00450

73141

21271

60433

31975

52455

43589

BUCKETS

0:

1:

2: 00120

3: 60433

4: 73141

5: 00450

6:

7: 21271

8:

9:

LSD Radix Sort

00120

00450

73141

21271

60433

31975

52455

43589

BUCKETS

0:

1:

2: 00120

3: 60433

4: 73141

5: 00450

6:

7: 21271, 31975

8:

9:

LSD Radix Sort

00120

00450

73141

21271

60433

31975

52455

43589

BUCKETS

0:

1:

2: 00120

3: 60433

4: 73141

5: 00450, 52455

6:

7: 21271, 31975

8:

9:

LSD Radix Sort

00120

00450

73141

21271

60433

31975

52455

43589

BUCKETS

0:

1:

2: 00120

3: 60433

4: 73141

5: 00450, 52455

6:

7: 21271, 31975

8: 43589

9:

LSD Radix Sort

BUCKETS

0:

1:

2: 00120

3: 60433

4: 73141

5: 00450, 52455

6:

7: 21271, 31975

8: 43589

9:

LSD Radix Sort

00120

BUCKETS

0:

1:

2: 00120

3: 60433

4: 73141

5: 00450, 52455

6:

7: 21271, 31975

8: 43589

9:

LSD Radix Sort

00120

60433

BUCKETS

0:

1:

2: 00120

3: 60433

4: 73141

5: 00450, 52455

6:

7: 21271, 31975

8: 43589

9:

LSD Radix Sort

00120

60433

73141

BUCKETS

0:

1:

2: 00120

3: 60433

4: 73141

5: 00450, 52455

6:

7: 21271, 31975

8: 43589

9:

LSD Radix Sort

00120

60433

73141

00450

BUCKETS

0:

1:

2: 00120

3: 60433

4: 73141

5: 00450, 52455

6:

7: 21271, 31975

8: 43589

9:

LSD Radix Sort

00120

60433

73141

00450

52455

BUCKETS

0:

1:

2: 00120

3: 60433

4: 73141

5: 00450, 52455

6:

7: 21271, 31975

8: 43589

9:

LSD Radix Sort

00120

60433

73141

00450

52455

21271

BUCKETS

0:

1:

2: 00120

3: 60433

4: 73141

5: 00450, 52455

6:

7: 21271, 31975

8: 43589

9:

LSD Radix Sort

00120

60433

73141

00450

52455

21271

31975

BUCKETS

0:

1:

2: 00120

3: 60433

4: 73141

5: 00450, 52455

6:

7: 21271, 31975

8: 43589

9:

LSD Radix Sort

00120

60433

73141

00450

52455

21271

31975

43589

BUCKETS

0:

1:

2: 00120

3: 60433

4: 73141

5: 00450, 52455

6:

7: 21271, 31975

8: 43589

9:

LSD Radix Sort

00120

60433

73141

00450

52455

21271

31975

43589

BUCKETS

0:

1:

2:

3:

4:

5:

6:

7:

8:

9:

LSD Radix Sort

00120

60433

73141

00450

52455

21271

31975

43589

BUCKETS

0:

1:

2:

3:

4:

5:

6:

7:

8:

9:

LSD Radix Sort

00120

60433

73141

00450

52455

21271

31975

43589

BUCKETS

0:

1: 00120

2:

3:

4:

5:

6:

7:

8:

9:

LSD Radix Sort

00120

60433

73141

00450

52455

21271

31975

43589

BUCKETS

0:

1: 00120

2:

3:

4: 60433

5:

6:

7:

8:

9:

LSD Radix Sort

00120

60433

73141

00450

52455

21271

31975

43589

BUCKETS

0:

1: 00120, 73141

2:

3:

4: 60433

5:

6:

7:

8:

9:

LSD Radix Sort

00120

60433

73141

00450

52455

21271

31975

43589

BUCKETS

0:

1: 00120, 73141

2:

3:

4: 60433, 00450

5:

6:

7:

8:

9:

LSD Radix Sort

00120

60433

73141

00450

52455

21271

31975

43589

BUCKETS

0:

1: 00120, 73141

2:

3:

4: 60433, 00450, 52455

5:

6:

7:

8:

9:

LSD Radix Sort

00120

60433

73141

00450

52455

21271

31975

43589

BUCKETS

0:

1: 00120, 73141

2: 21271

3:

4: 60433, 00450, 52455

5:

6:

7:

8:

9:

LSD Radix Sort

00120

60433

73141

00450

52455

21271

31975

43589

BUCKETS

0:

1: 00120, 73141

2: 21271

3:

4: 60433, 00450, 52455

5:

6:

7:

8:

9: 31975

LSD Radix Sort

00120

60433

73141

00450

52455

21271

31975

43589

BUCKETS

0:

1: 00120, 73141

2: 21271

3:

4: 60433, 00450, 52455

5: 43589

6:

7:

8:

9: 31975

LSD Radix Sort

BUCKETS

0:

1: 00120, 73141

2: 21271

3:

4: 60433, 00450, 52455

5: 43589

6:

7:

8:

9: 31975

LSD Radix Sort

00120

BUCKETS

0:

1: 00120, 73141

2: 21271

3:

4: 60433, 00450, 52455

5: 43589

6:

7:

8:

9: 31975

LSD Radix Sort

00120

73141

BUCKETS

0:

1: 00120, 73141

2: 21271

3:

4: 60433, 00450, 52455

5: 43589

6:

7:

8:

9: 31975

LSD Radix Sort

00120

73141

21271

BUCKETS

0:

1: 00120, 73141

2: 21271

3:

4: 60433, 00450, 52455

5: 43589

6:

7:

8:

9: 31975

LSD Radix Sort

00120

73141

21271

60433

BUCKETS

0:

1: 00120, 73141

2: 21271

3:

4: 60433, 00450, 52455

5: 43589

6:

7:

8:

9: 31975

LSD Radix Sort

00120

73141

21271

60433

00450

BUCKETS

0:

1: 00120, 73141

2: 21271

3:

4: 60433, 00450, 52455

5: 43589

6:

7:

8:

9: 31975

LSD Radix Sort

00120

73141

21271

60433

00450

52455

BUCKETS

0:

1: 00120, 73141

2: 21271

3:

4: 60433, 00450, 52455

5: 43589

6:

7:

8:

9: 31975

LSD Radix Sort

00120

73141

21271

60433

00450

52455

43589

BUCKETS

0:

1: 00120, 73141

2: 21271

3:

4: 60433, 00450, 52455

5: 43589

6:

7:

8:

9: 31975

LSD Radix Sort

00120

73141

21271

60433

00450

52455

43589

31975

BUCKETS

0:

1: 00120, 73141

2: 21271

3:

4: 60433, 00450, 52455

5: 43589

6:

7:

8:

9: 31975

LSD Radix Sort

00120

73141

21271

60433

00450

52455

43589

31975

BUCKETS

0:

1:

2:

3:

4:

5:

6:

7:

8:

9:

LSD Radix Sort

00120

73141

21271

60433

00450

52455

43589

31975

BUCKETS

0:

1:

2:

3:

4:

5:

6:

7:

8:

9:

LSD Radix Sort

00120

73141

21271

60433

00450

52455

43589

31975

BUCKETS

0: 00120

1:

2:

3:

4:

5:

6:

7:

8:

9:

LSD Radix Sort

00120

73141

21271

60433

00450

52455

43589

31975

BUCKETS

0: 00120

1:

2:

3: 73141

4:

5:

6:

7:

8:

9:

LSD Radix Sort

00120

73141

21271

60433

00450

52455

43589

31975

BUCKETS

0: 00120

1: 21271

2:

3: 73141

4:

5:

6:

7:

8:

9:

LSD Radix Sort

00120

73141

21271

60433

00450

52455

43589

31975

BUCKETS

0: 00120, 60433

1: 21271

2:

3: 73141

4:

5:

6:

7:

8:

9:

LSD Radix Sort

00120

73141

21271

60433

00450

52455

43589

31975

BUCKETS

0: 00120, 60433, 00450

1: 21271

2:

3: 73141

4:

5:

6:

7:

8:

9:

LSD Radix Sort

00120

73141

21271

60433

00450

52455

43589

31975

BUCKETS

0: 00120, 60433, 00450

1: 21271

2: 52455

3: 73141

4:

5:

6:

7:

8:

9:

LSD Radix Sort

00120

73141

21271

60433

00450

52455

43589

31975

BUCKETS

0: 00120, 60433, 00450

1: 21271

2: 52455

3: 73141, 43589

4:

5:

6:

7:

8:

9:

LSD Radix Sort

00120

73141

21271

60433

00450

52455

43589

31975

BUCKETS

0: 00120, 60433, 00450

1: 21271, 31975

2: 52455

3: 73141, 43589

4:

5:

6:

7:

8:

9:

LSD Radix Sort

BUCKETS

0: 00120, 60433, 00450

1: 21271, 31975

2: 52455

3: 73141, 43589

4:

5:

6:

7:

8:

9:

LSD Radix Sort

00120

BUCKETS

0: 00120, 60433, 00450

1: 21271, 31975

2: 52455

3: 73141, 43589

4:

5:

6:

7:

8:

9:

LSD Radix Sort

00120

60433

BUCKETS

0: 00120, 60433, 00450

1: 21271, 31975

2: 52455

3: 73141, 43589

4:

5:

6:

7:

8:

9:

LSD Radix Sort

00120

60433

00450

BUCKETS

0: 00120, 60433, 00450

1: 21271, 31975

2: 52455

3: 73141, 43589

4:

5:

6:

7:

8:

9:

LSD Radix Sort

00120

60433

00450

21271

BUCKETS

0: 00120, 60433, 00450

1: 21271, 31975

2: 52455

3: 73141, 43589

4:

5:

6:

7:

8:

9:

LSD Radix Sort

00120

60433

00450

21271

31975

BUCKETS

0: 00120, 60433, 00450

1: 21271, 31975

2: 52455

3: 73141, 43589

4:

5:

6:

7:

8:

9:

LSD Radix Sort

00120

60433

00450

21271

31975

52455

BUCKETS

0: 00120, 60433, 00450

1: 21271, 31975

2: 52455

3: 73141, 43589

4:

5:

6:

7:

8:

9:

LSD Radix Sort

00120

60433

00450

21271

31975

52455

73141

BUCKETS

0: 00120, 60433, 00450

1: 21271, 31975

2: 52455

3: 73141, 43589

4:

5:

6:

7:

8:

9:

LSD Radix Sort

00120

60433

00450

21271

31975

52455

73141

43589

BUCKETS

0: 00120, 60433, 00450

1: 21271, 31975

2: 52455

3: 73141, 43589

4:

5:

6:

7:

8:

9:

LSD Radix Sort

00120

60433

00450

21271

31975

52455

73141

43589

BUCKETS

0:

1:

2:

3:

4:

5:

6:

7:

8:

9:

LSD Radix Sort

00120

60433

00450

21271

31975

52455

73141

43589

BUCKETS

0:

1:

2:

3:

4:

5:

6:

7:

8:

9:

LSD Radix Sort

00120

60433

00450

21271

31975

52455

73141

43589

BUCKETS

0: 00120

1:

2:

3:

4:

5:

6:

7:

8:

9:

LSD Radix Sort

00120

60433

00450

21271

31975

52455

73141

43589

BUCKETS

0: 00120

1:

2:

3:

4:

5:

6: 60433

7:

8:

9:

LSD Radix Sort

00120

60433

00450

21271

31975

52455

73141

43589

BUCKETS

0: 00120, 00450

1:

2:

3:

4:

5:

6: 60433

7:

8:

9:

LSD Radix Sort

00120

60433

00450

21271

31975

52455

73141

43589

BUCKETS

0: 00120, 00450

1:

2: 21271

3:

4:

5:

6: 60433

7:

8:

9:

LSD Radix Sort

00120

60433

00450

21271

31975

52455

73141

43589

BUCKETS

0: 00120, 00450

1:

2: 21271

3: 31975

4:

5:

6: 60433

7:

8:

9:

LSD Radix Sort

00120

60433

00450

21271

31975

52455

73141

43589

BUCKETS

0: 00120, 00450

1:

2: 21271

3: 31975

4:

5: 52455

6: 60433

7:

8:

9:

LSD Radix Sort

00120

60433

00450

21271

31975

52455

73141

43589

BUCKETS

0: 00120, 00450

1:

2: 21271

3: 31975

4:

5: 52455

6: 60433

7: 73141

8:

9:

LSD Radix Sort

00120

60433

00450

21271

31975

52455

73141

43589

BUCKETS

0: 00120, 00450

1:

2: 21271

3: 31975

4: 43589

5: 52455

6: 60433

7: 73141

8:

9:

LSD Radix Sort

BUCKETS

0: 00120, 00450

1:

2: 21271

3: 31975

4: 43589

5: 52455

6: 60433

7: 73141

8:

9:

LSD Radix Sort

00120

BUCKETS

0: 00120, 00450

1:

2: 21271

3: 31975

4: 43589

5: 52455

6: 60433

7: 73141

8:

9:

LSD Radix Sort

00120

00450

BUCKETS

0: 00120, 00450

1:

2: 21271

3: 31975

4: 43589

5: 52455

6: 60433

7: 73141

8:

9:

LSD Radix Sort

00120

00450

21271

BUCKETS

0: 00120, 00450

1:

2: 21271

3: 31975

4: 43589

5: 52455

6: 60433

7: 73141

8:

9:

LSD Radix Sort

00120

00450

21271

31975

BUCKETS

0: 00120, 00450

1:

2: 21271

3: 31975

4: 43589

5: 52455

6: 60433

7: 73141

8:

9:

LSD Radix Sort

00120

00450

21271

31975

43589

BUCKETS

0: 00120, 00450

1:

2: 21271

3: 31975

4: 43589

5: 52455

6: 60433

7: 73141

8:

9:

LSD Radix Sort

00120

00450

21271

31975

43589

52455

BUCKETS

0: 00120, 00450

1:

2: 21271

3: 31975

4: 43589

5: 52455

6: 60433

7: 73141

8:

9:

LSD Radix Sort

00120

00450

21271

31975

43589

52455

60433

BUCKETS

0: 00120, 00450

1:

2: 21271

3: 31975

4: 43589

5: 52455

6: 60433

7: 73141

8:

9:

LSD Radix Sort

00120

00450

21271

31975

43589

52455

60433

73141

BUCKETS

0: 00120, 00450

1:

2: 21271

3: 31975

4: 43589

5: 52455

6: 60433

7: 73141

8:

9:

LSD Radix Sort

00120

00450

21271

31975

43589

52455

60433

73141

Radix Sort

1	4	1	7	1	7	9	3

0	3	0	1	1	0	0	2	0	1
0	1	2	3	4	5	6	7	8	9

the so-called ***count array*** *with as many items as the* ***radix*** *(**10**)*

Radix Sort

1	4	1	7	1	7	9	3

0	3	0	1	1	0	0	2	0	1
0	1	2	3	4	5	6	7	8	9

*the so-called **count array** with as many items as the **radix** (**10**)*

*we have to transform the **count array** to know the postions of the items in the **final sorted array** – this is why to construct the **cumulative count array***

Radix Sort

1	4	1	7	1	7	9	3

0	3	0	1	1	0	0	2	0	1
0	1	2	3	4	5	6	7	8	9

*the so-called **count array** with as many items as the **radix** (**10**)*

1	1	1	3	4	7	7	9

Radix Sort

1	4	1	7	1	7	9	3

0	3	0	1	1	0	0	2	0	1

*the so-called **count array** with as many items as the **radix** (**10**)*

Radix Sort

1	4	1	7	1	7	9	3

0	3	0	1	1	0	0	2	0	1

the so-called ***count array*** *with as many items as the* ***radix*** *(****10****)*

0									

Radix Sort

1	4	1	7	1	7	9	3

0	3	0	1	1	0	0	2	0	1

*the so-called **count array** with as many items as the **radix** (**10**)*

0	3								

Radix Sort

1	4	1	7	1	7	9	3

0	3	0	1	1	0	0	2	0	1

*the so-called **count array** with as many items as the **radix** (**10**)*

0	3	3							

Radix Sort

1	4	1	7	1	7	9	3

0	3	0	1	1	0	0	2	0	1

*the so-called **count array** with as many items as the **radix** (**10**)*

0	3	3	4						

Radix Sort

1	4	1	7	1	7	9	3

0	3	0	1	1	0	0	2	0	1

*the so-called **count array** with as many items as the **radix** (**10**)*

0	3	3	4	5					

Radix Sort

1	4	1	7	1	7	9	3

0	3	0	1	1	0	0	2	0	1

*the so-called **count array** with as many items as the **radix** (**10**)*

0	3	3	4	5	5				

Radix Sort

1	4	1	7	1	7	9	3

0	3	0	1	1	0	0	2	0	1

the so-called ***count array*** *with as many items as the* ***radix*** *(**10**)*

0	3	3	4	5	5	5			

Radix Sort

1	4	1	7	1	7	9	3

0	3	0	1	1	0	0	2	0	1

the so-called ***count array*** *with as many items as the* ***radix*** *(**10**)*

0	3	3	4	5	5	5	7		

Radix Sort

1	4	1	7	1	7	9	3

0	3	0	1	1	0	0	2	0	1

*the so-called **count array** with as many items as the **radix** (**10**)*

0	3	3	4	5	5	5	7	7	

Radix Sort

1	4	1	7	1	7	9	3

0	3	0	1	1	0	0	2	0	1

the so-called ***count array*** *with as many items as the* ***radix*** *(**10**)*

0	3	3	4	5	5	5	7	7	8

this is the actual positions of the sorted items in the original array – we go from right to left to ***maintain stability***