Sorting Algorithms

Do Now

Given the following list of numbers : –2, 45, 0, 11, -9

Describe a strategy to put these numbers in order from smallest to largest

Note: Your strategy should work for any list of numbers, including a list that has some duplicate values

Bubble Sort

Bubble Sort is one of the most widely discussed algorithms, simply because of its lack of efficiency for sorting arrays.

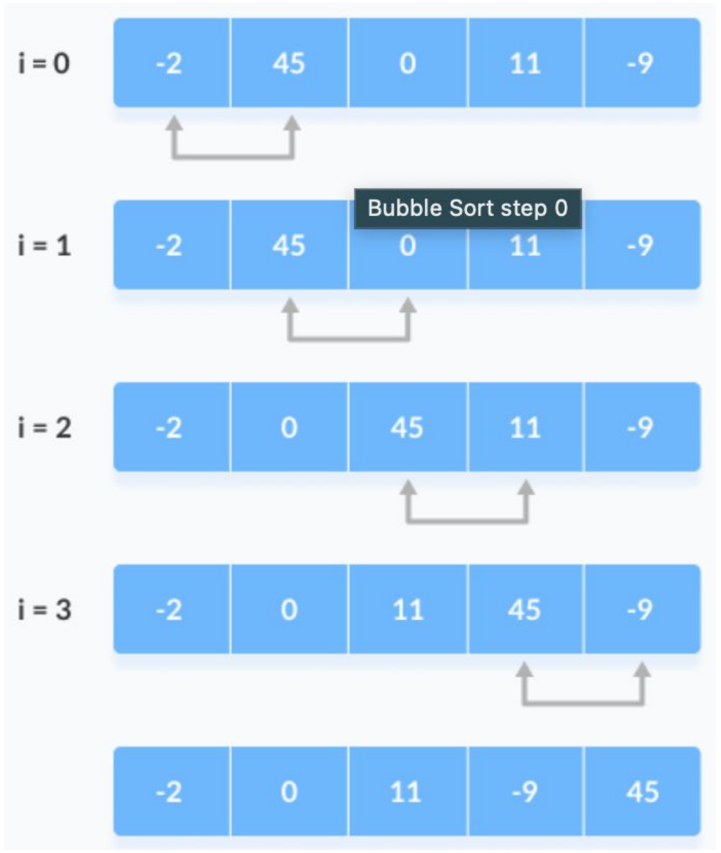
If an array is already sorted, Bubble Sort will only pass through the array once.

How does Bubble Sort work?

This algorithm compares two adjacent elements and swaps them until they are in the intended order.

As the algorithm progresses we have a sorted partition and an unsorted partition (This is a logical partition, we do not have two arrays)

Bubble Sort - Step 1 (Pass 1)

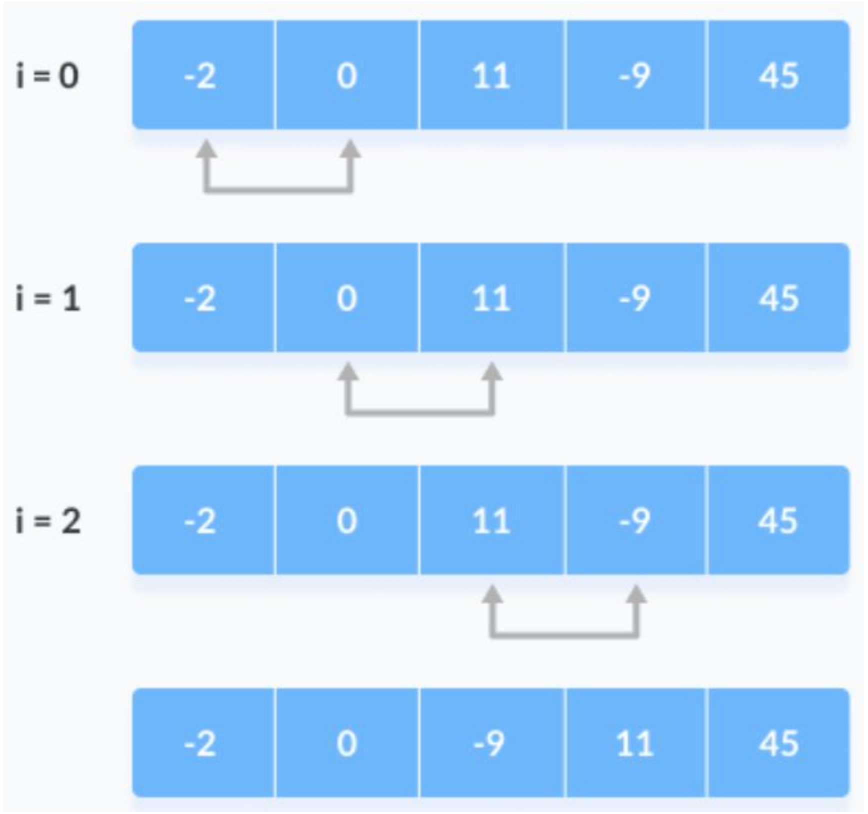


1. Starting at index 0, compare the first and the second elements.
2. Swap the element if the first one is greater than the second one.
3. Next, compare the second and the third elements and swap them if the order is not correct.
4. Keep doing the process until your algorithm reaches the last element.

What do you notice? Is any element at its correct position?

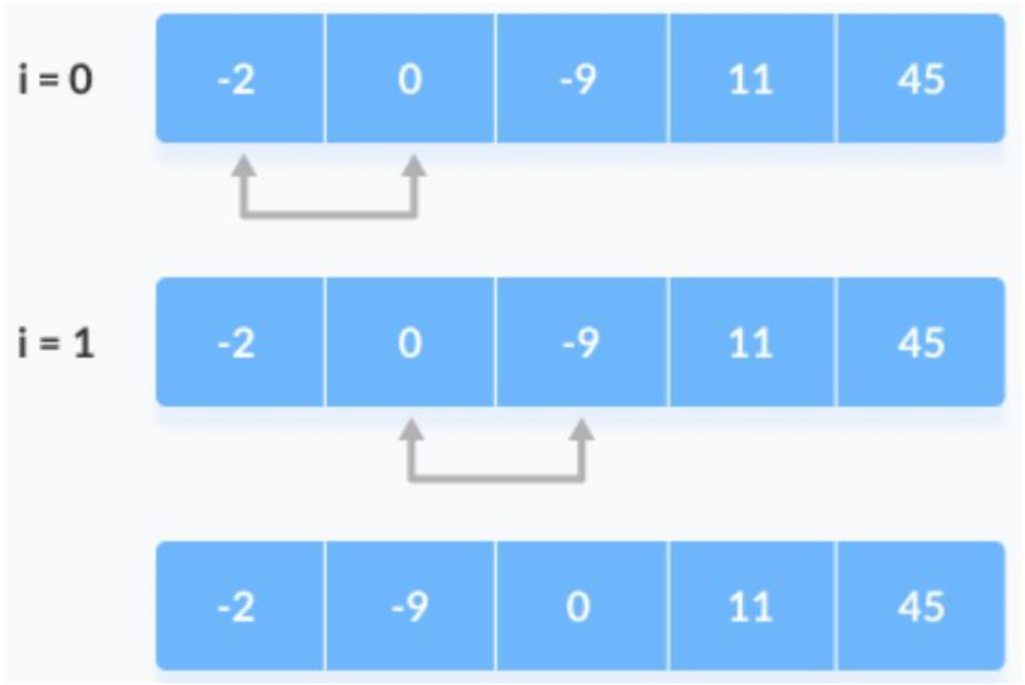
A logical sorted partition starts forming with the largest element placed at the end.

Bubble Sort - Step 2 (Pass 2)

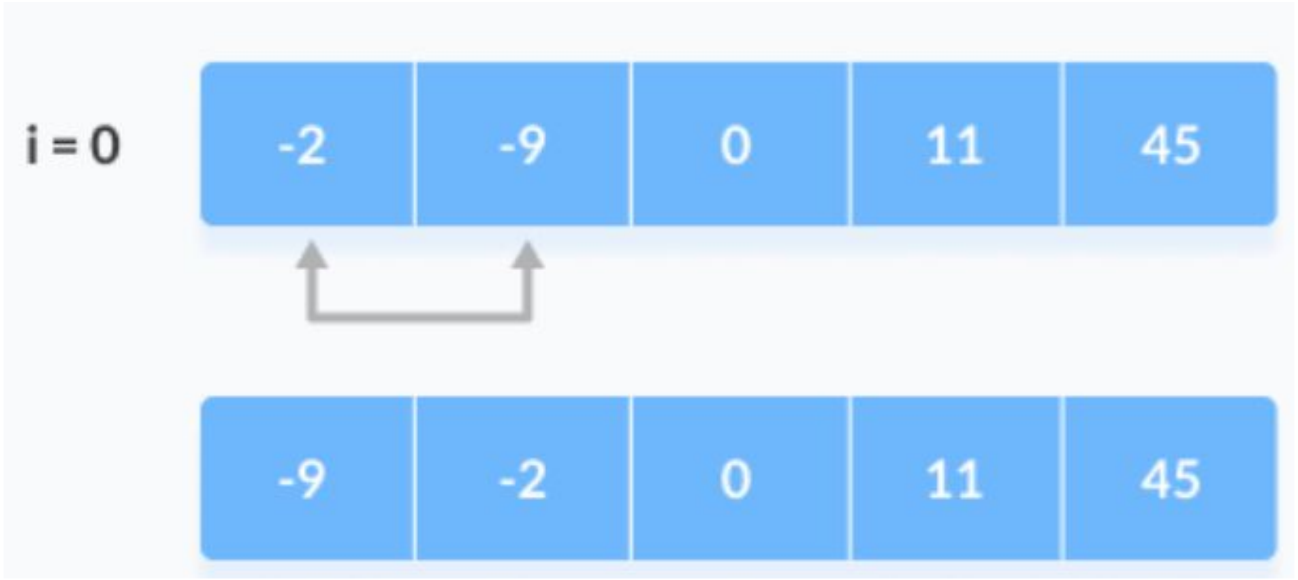


Repeat the comparing and swapping for the remaining iterations (unsorted partition).

Bubble Sort - Step 3 (Pass 3)



Bubble Sort - Step 4 (Pass 4)



Discussion Time

Read the Bubble Sort algorithm again. Then answer the following:

1. If the array is filled with all 0’s, what should happen during sorting?
2. How many swaps should occur in this case, and why?
3. What could cause this algorithm to run much slower than it should?
4. How could you improve the algorithm so it runs faster in this scenario?

Bubble Sort Algorithm

1. Starting at index 0, compare the first and the second elements.
2. Swap the element if the first one is greater than the second one.
3. Next, compare the second and the third elements and swap them if the order is not correct.
4. Keep doing the process until your algorithm reaches the last element.

Optimized Bubble Sort

This is a **faster version of Bubble Sort** that stops early when the list is already sorted.

Strategy

Use a **swap flag** to detect whether any swap happened during a pass.

- If a swap occurs => keep sorting
- If **no** swaps occur => array is sorted => **stop immediately**

Why is it better?

- Avoids unnecessary passes
- Improves best-case time complexity
- Makes Bubble Sort **adaptive** (performs better on nearly sorted data)

Optimized Bubble Sort - Steps

1. Start each pass with **swappedFlag = false**
2. Compare adjacent elements; swap if needed
3. If any swap occurs, set **swappedFlag = true**
4. After the pass, if **swappedFlag == false**, **break** out of the loop

Bubble Sort Advantages

- Bubble sort is easy to understand and implement.
- It’s an adaptive sorting algorithm. The order of elements affects the time complexity of the sorting.

Bubble Sort Disadvantages

- Time complexity of **?** which makes it very slow for large data sets.
- It is not efficient for large data sets, because it requires multiple passes through the data.

Bubble Sort Applications

- It is often used to introduce the concept of a sorting algorithm because it is very simple.
- This algorithm is not efficient for real life applications unless:
    - Complexity does not matter
    - Short and simple code is preferred

Bubble Sort Visualizer

https://www.hackerearth.com/practice/algorithms/sorting/bubble-sort/visualize/

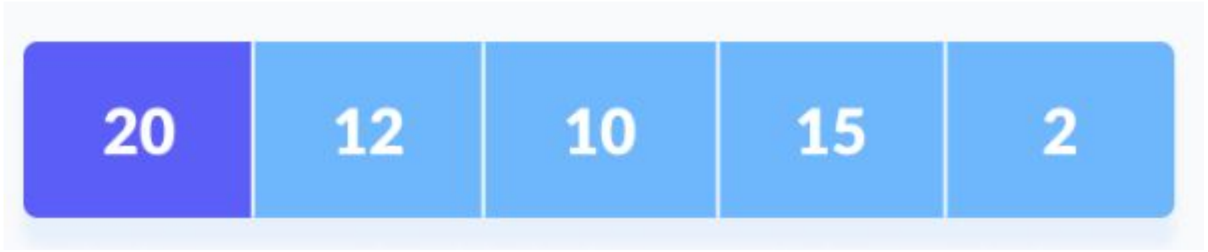
Selection Sort

This algorithm selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.

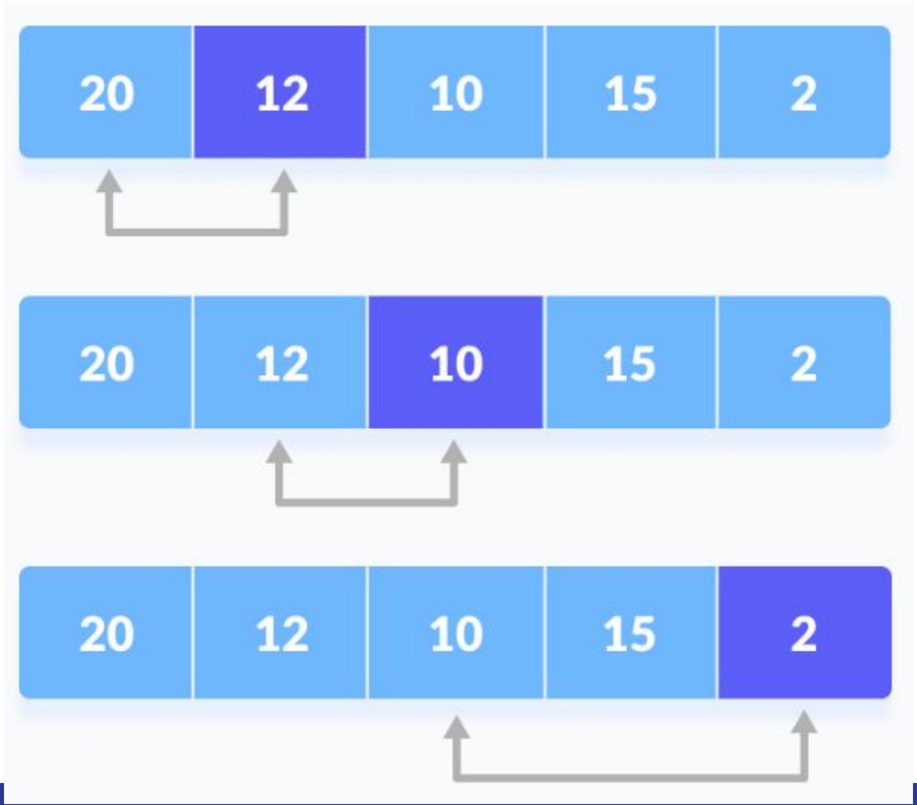
It is an in-place algorithm, you only need extra memory for loop variables and swap space.

Selection Sort - Initial Array

Set the first element as **minimum:**



Selection Sort - Comparing



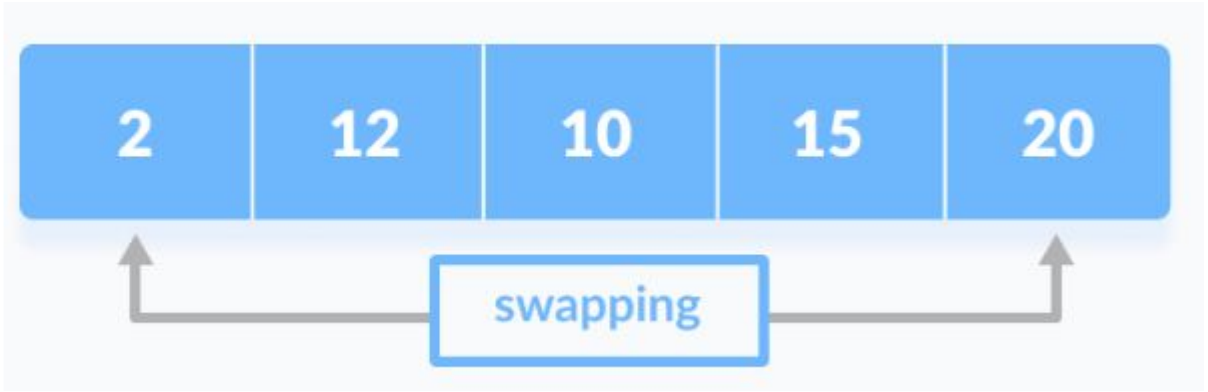
Compare **`minimum`** with the second element. If the second element is smaller than **`minimum`**, assign the second element as **`minimum`**.

Compare **`minimum`** with the third element. Again, if the third element is smaller, then assign **`minimum`** to the third element otherwise do nothing.

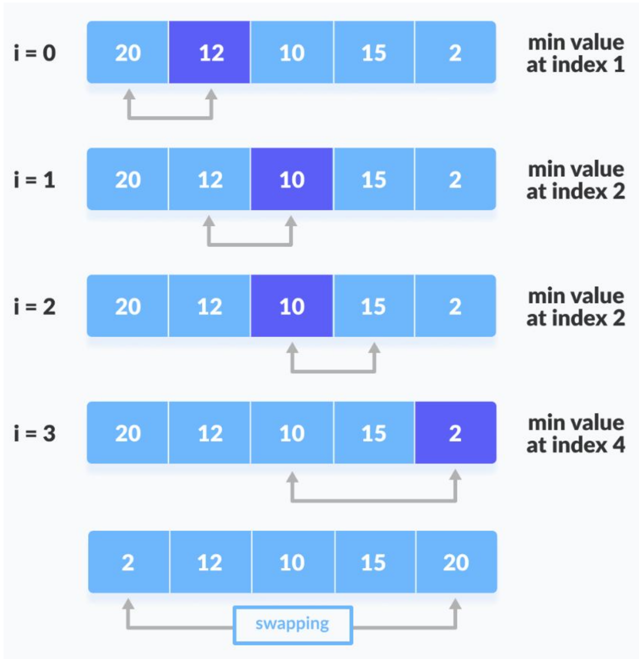
Repeat the process until the last element.

Selection Sort - Swapping

After each iteration, **minimum** is placed in the front of the unsorted list which becomes the logical sorted partition.



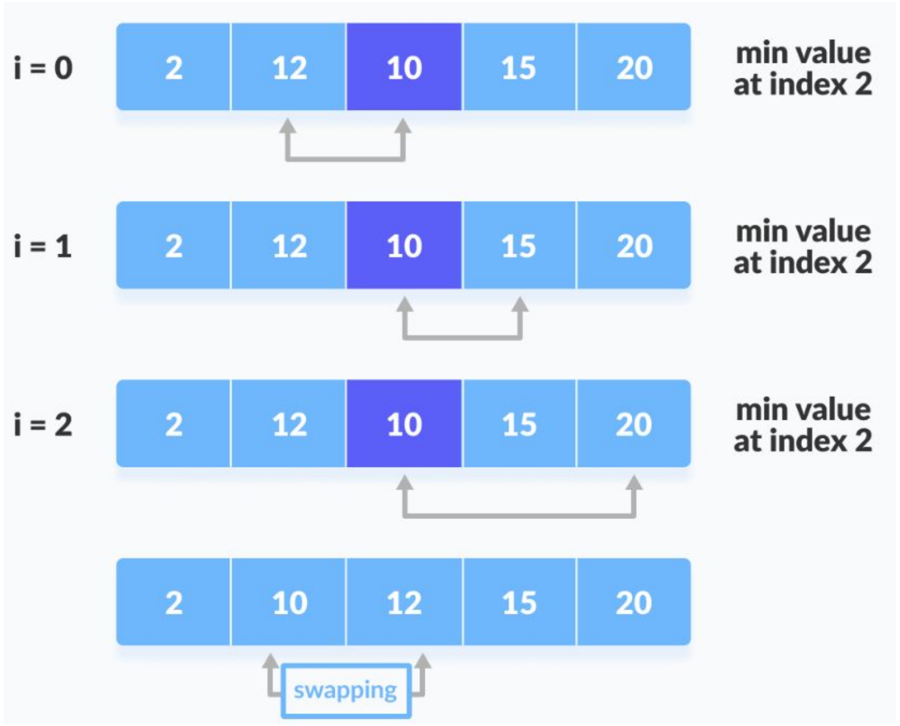
Selection Sort - Step 1 (Pass 1)



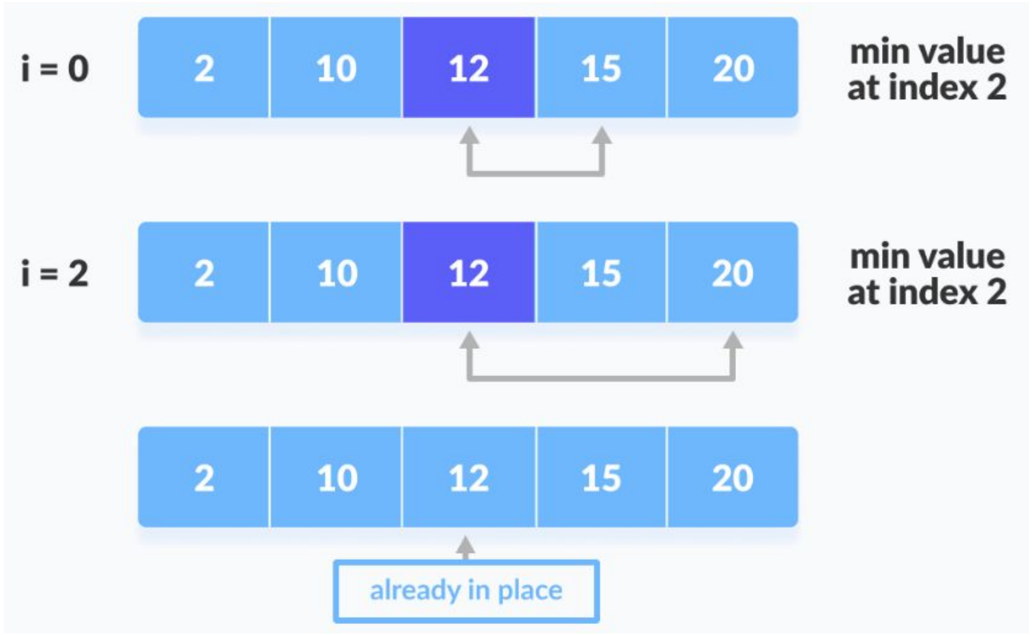
For each iteration, indexing starts from the first unsorted element.

Repeat the comparing minimum element and swapping processes until all the elements are placed at their correct positions.

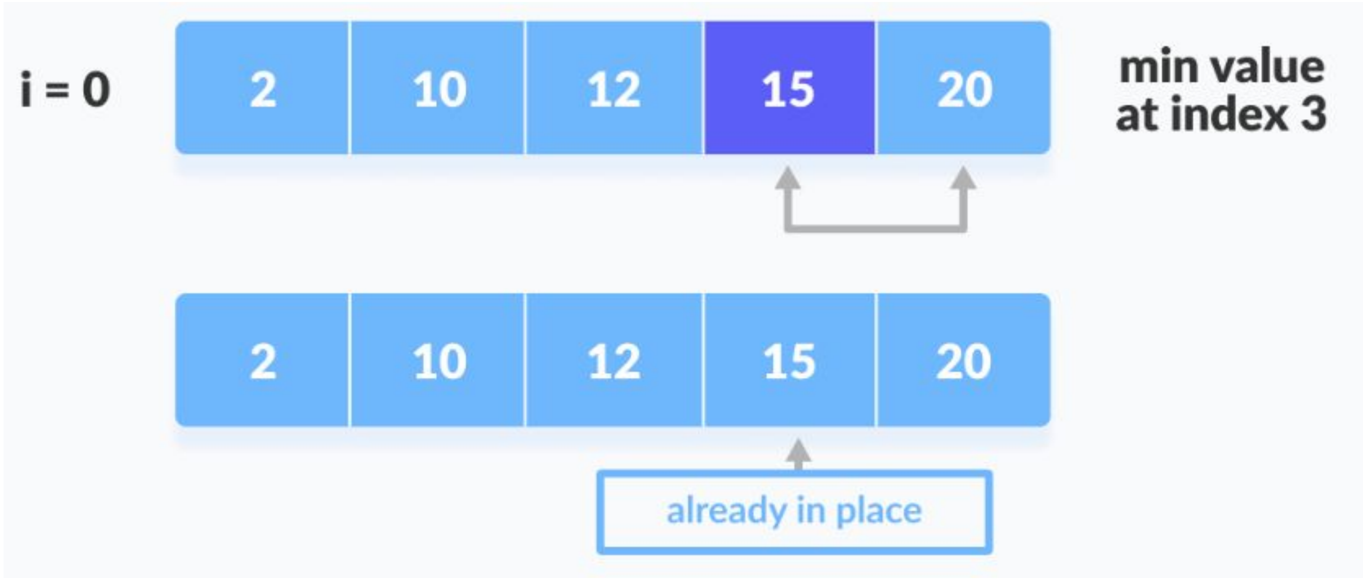
Selection Sort - Step 2 (Pass 2)



Selection Sort - Step 3 (Pass 3)



Selection Sort - Step 4 (Pass 4)



Alternative

Note that selection sort can also be implemented by selecting the **largest** element in each pass and placing it at the end of the unsorted partition, working backwards. Both approaches have identical time complexity, the choice is simply a matter of convention.

Selection Sort - Applications

The selection sort is used when:

- The user wants to sort a small list of items in ascending order.
- The user needs to make sure that all the values in the list have been checked.

Selection Sort Advantages

- It performs well on small lists.
- It is an in-place algorithm. It does not require a lot of space for sorting.
- Only one extra space is required for holding the temporal variable.

Selection Sort Disadvantages

- Poor performance with lists that are huge.
- Complexity ?

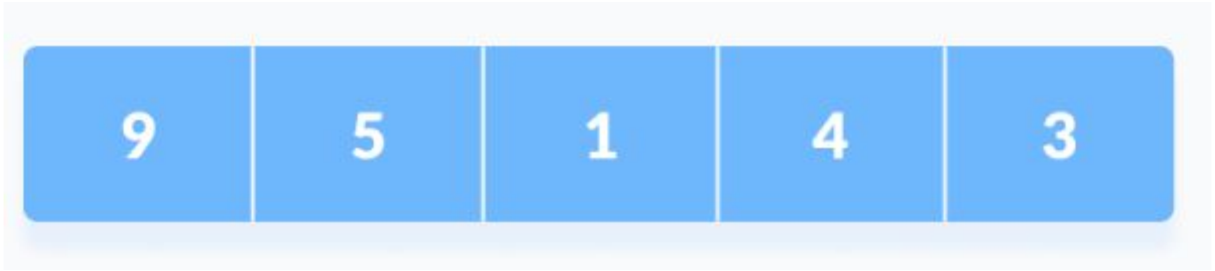
How does Insertion Sort work?

Insertion sort **places an unsorted element at its suitable place** in each iteration.

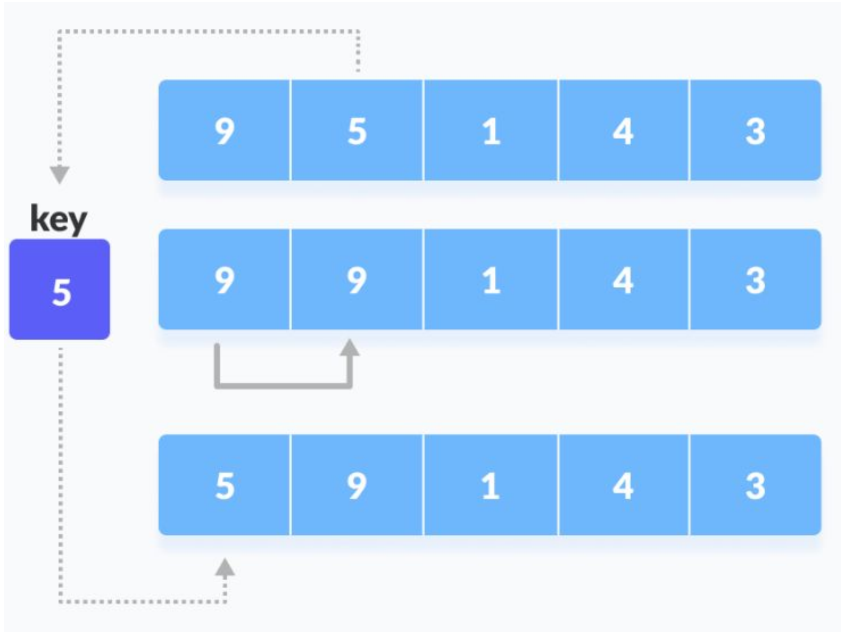
This algorithm **assumes** that the **element at position 0 is sorted.** This element becomes part of the logical sorted partition.

Then, the **other elements are compared to their previous neighbor in the logical sorted partition**. If the current element is smaller than its previous neighbor, the algorithm should compare it to the elements before and swap elements when the desired position for the current element has been found.

Insertion Sort - Initial Array



Insertion Sort - Step 1 (Pass 1)

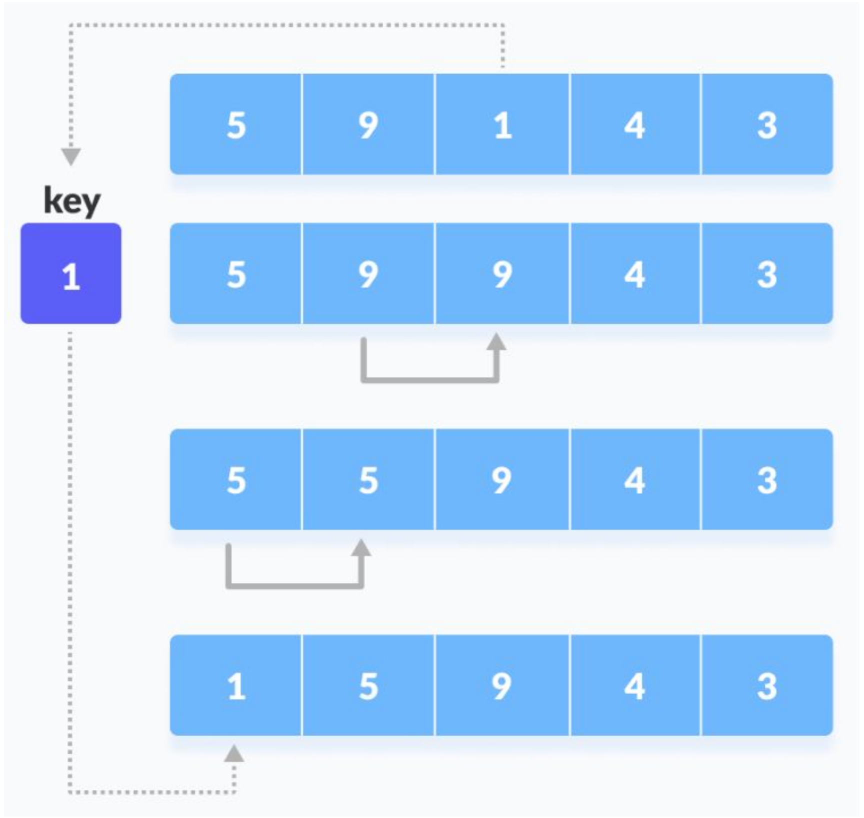


This algorithm assumes that the element at position 0 is sorted. This element becomes part of the logical sorted partition.

The element at position 1 is stored in an extra variable **key**.

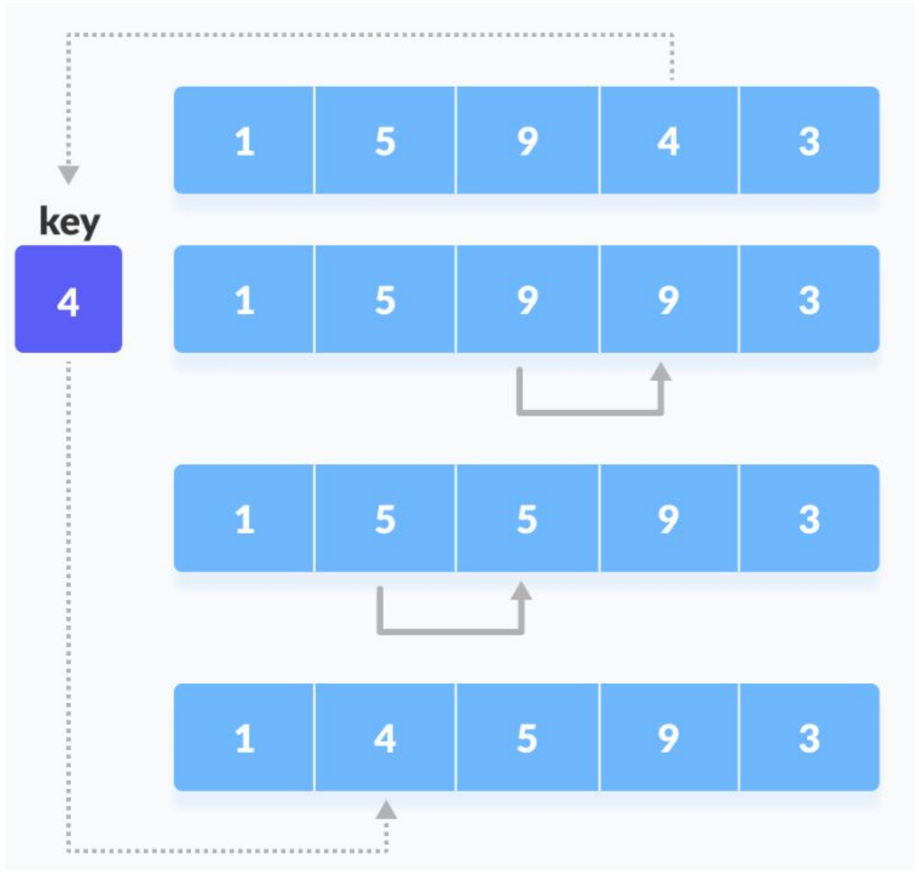
Compare **key** with the first element (sorted partition). If the first element is greater than **key**, then **key** is placed in front of the first element.

Insertion Sort - Step 2 (Pass 2)

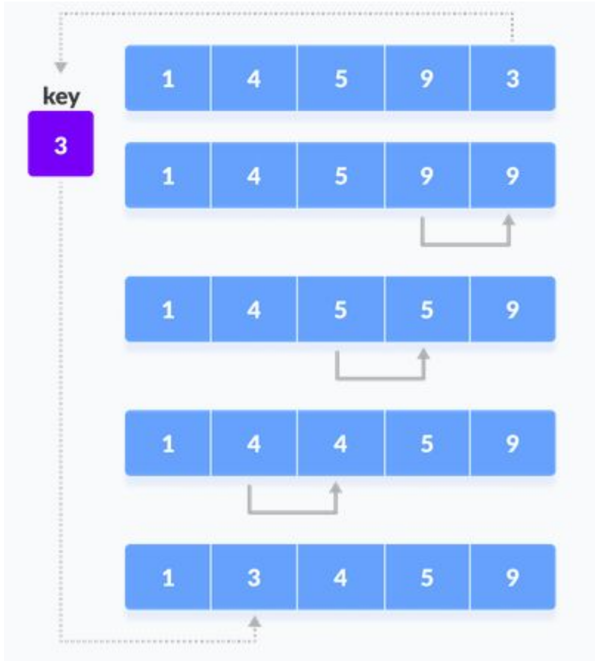


Now, let's take the third element and compare it with the elements in the logical sorted partition (elements on the left). Placed it just behind the element smaller than it. If there is no element smaller than it, then place it at the beginning of the array.

Insertion Sort - Step 3 (Pass 3)



Insertion Sort - Step 4 (Pass 4)



Insertion Sort Applications

This algorithm is used if:

- The array has a small number of elements
- There are only a few elements left to be sorted

Insertion Sort Advantages

- It can start the sorting process even if the complete data set isn’t available.
- Insertion sort makes fewer comparisons compared to the Bubble Sort.

Insertion Sort Disadvantages

- ? time complexity.
- Work slowly on large datasets.