Agenda

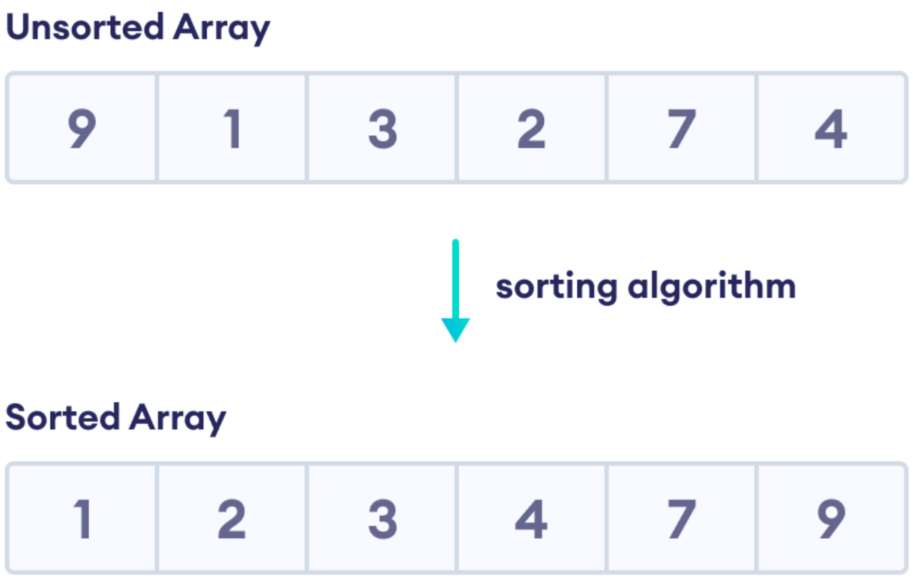
1. What is Sorting Algorithm?
2. Types of Sorting Algorithm.
3. Explanation of Bubble Sort,Selection Sort, Insertion Sort & Merge Sort
4. Visualisation of Sorting Algorithms
5. Complexity of Sorting Algorithms
6. Stability of Sorting Algorithms
7. Divide & Conquer Algorithm

What is Sorting Algorithm?

Sorting Algorithms are methods of reorganising a large number of items into some specific order such as highest to lowest, or vice-versa, or even some alphabetical order.

These algorithms take an input list, processes it (i.e, performs some operations on it) and produce the sorted list.

The most common example we experience every day is sorting clothes or other items on an e-commerce website either by lowest-price to highest, or li by popularity, or some other order.



Here, we are sorting the array in ascending order.

There are various sorting algorithms that can be used to complete this operation. And, we can use any algorithm based on the requirement.

Types of Sorting Algorithm.

1. Bubble Sort
2. Selection Sort
3. Insertion Sort
4. Merge Sort
5. Quick Sort
6. Counting Sort
7. Radix Sort
8. Bucket Sort
9. Heap Sort
10. Shell Sort

Note : This class will cover till Merge sort, in next class Quick, Counting & Radix sort will be covered. Last 3 topics will be covered later in advanced DS
chapter.

Bubble Sort

Bubble sort is a sorting algorithm that compares two adjacent elements and swaps them until they are not in the intended order.

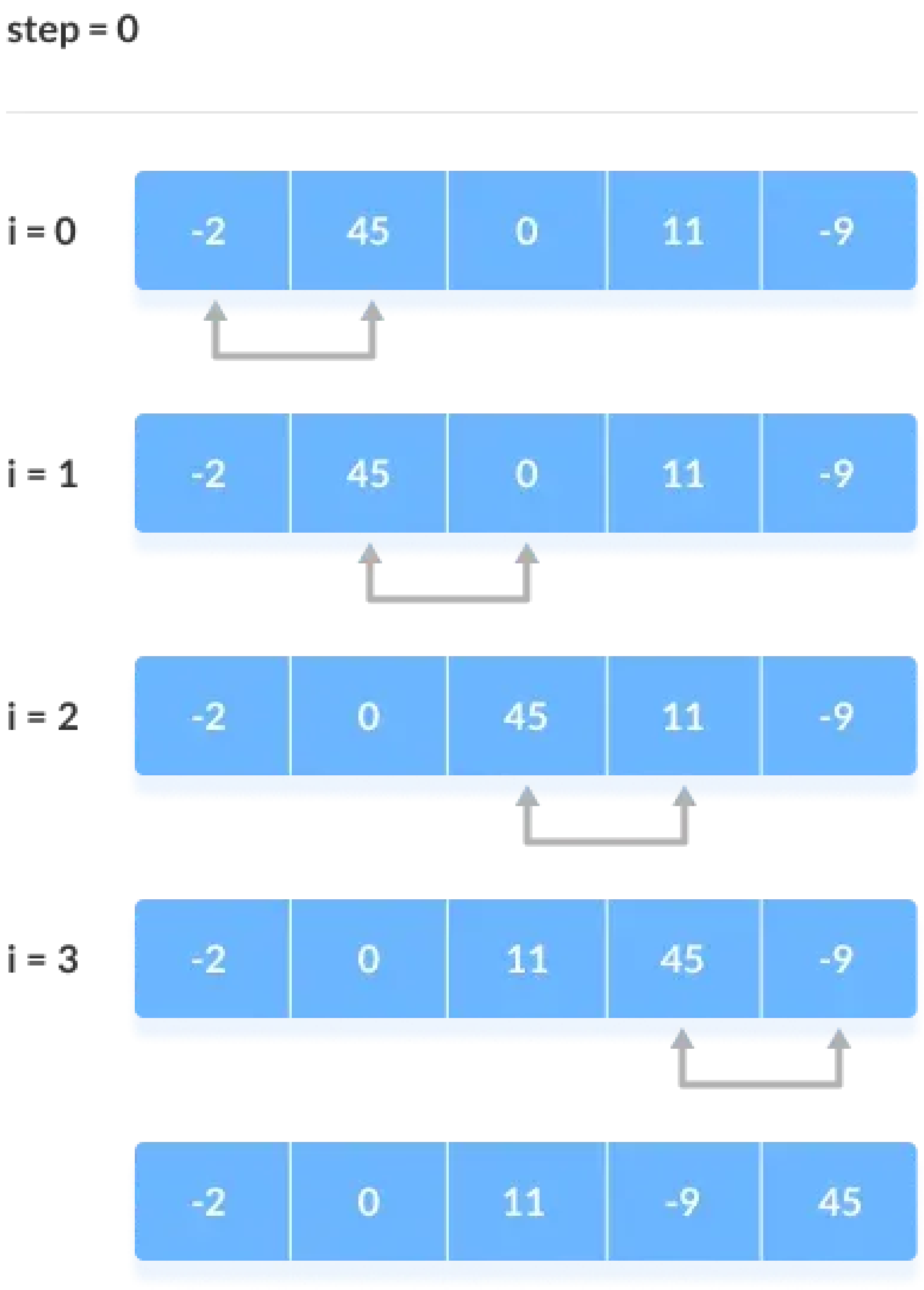
Just like the movement of air bubbles in the water that rise up to the surface, each element of the array move to the end in each iteration. Therefore, it
called a bubble sort.

Working of Bubble Sort

Suppose we are trying to sort the elements in **ascending order**.

1. First Iteration (Compare and Swap)

1. Starting from the first index, compare the first and the second elements.
2. If the first element is greater than the second element, they are swapped.
3. Now, compare the second and the third elements. Swap them if they are not in order.
4. The above process goes on until the last element.

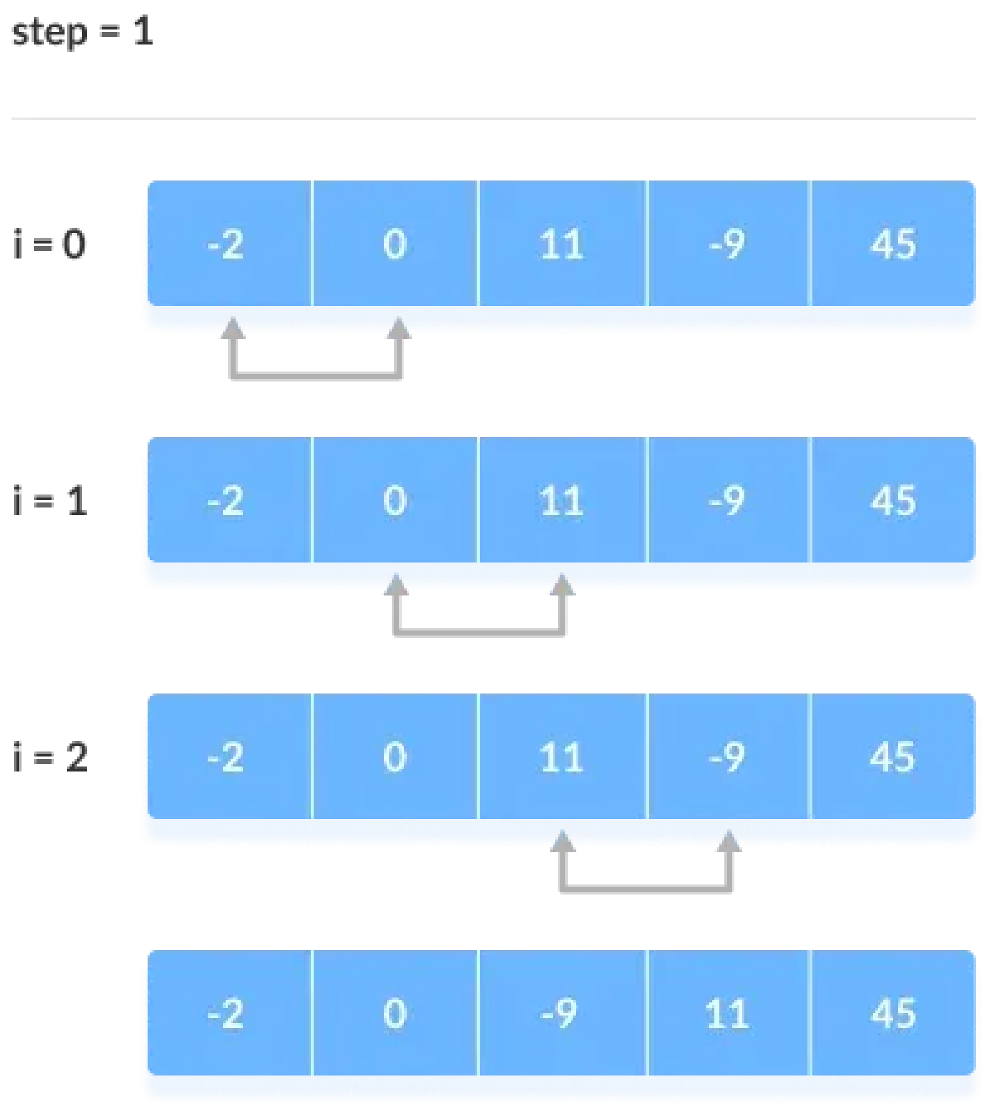


Compare the Adjacent Elements

2. Remaining Iteration

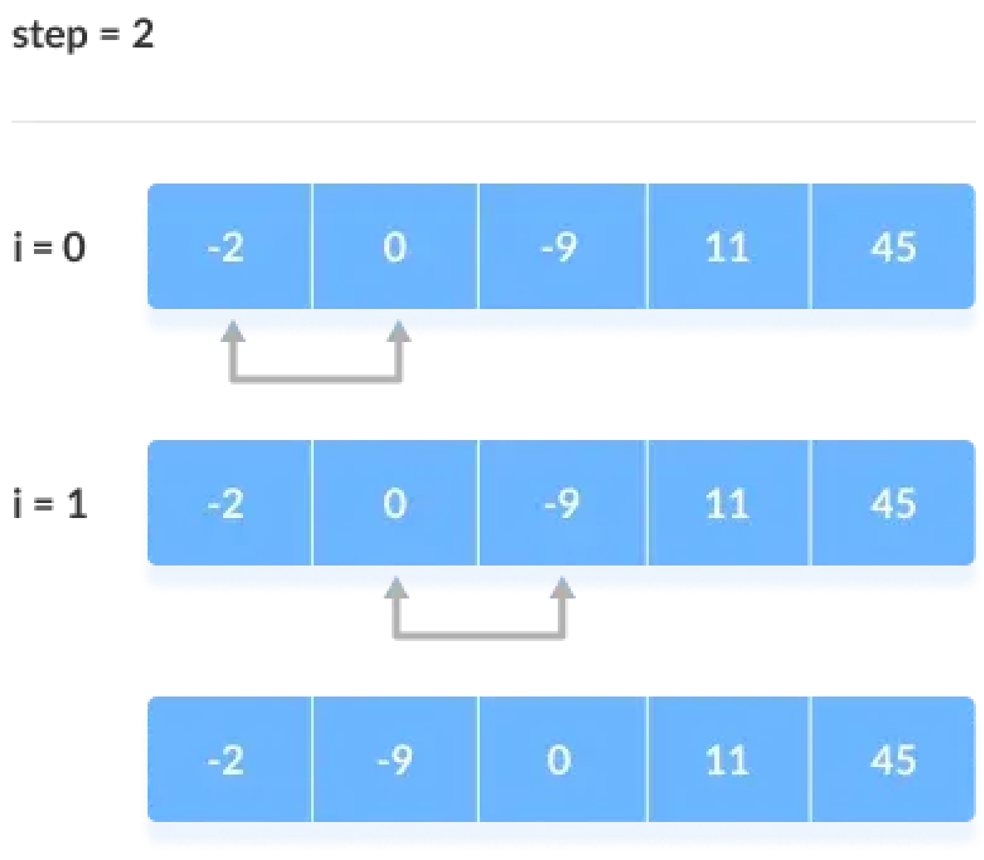
The same process goes on for the remaining iterations.

After each iteration, the largest element among the unsorted elements is placed at the end.



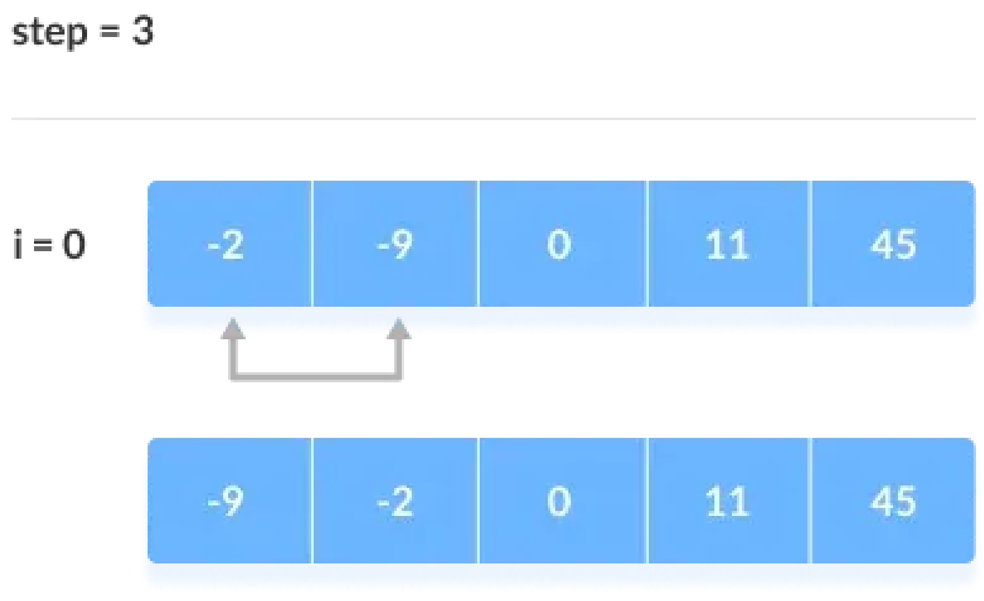
Put the largest element at the end

In each iteration, the comparison takes place up to the last unsorted element.



Compare the adjacent elements

The array is sorted when all the unsorted elements are placed at their correct positions.



The array is sorted if all elements are kept in the right order

Bubble Sort Algorithm

```
bubbleSort(array)
  for i <- 1 to indexOfLastUnsortedElement-1
    if leftElement > rightElement
```

```
      swap leftElement and rightElement
end bubbleSort
```

Bubble Sort Code in Javascript

```
// Bubble sort Implementation using Javascript


// Creating the bblSort function
 function bblSort(arr){

 for(var i = 0; i < arr.length; i++){

   // Last i elements are already in place
   for(var j = 0; j < ( arr.length - i -1 ); j++){

     // Checking if the item at present iteration
     // is greater than the next iteration
     if(arr[j] > arr[j+1]){

       // If the condition is true then swap them
       var temp = arr[j]
       arr[j] = arr[j + 1]
       arr[j+1] = temp
     }
   }
 }
 // Print the sorted array
 console.log(arr);
}


// This is our unsorted array
var arr = [234, 43, 55, 63,  5, 6, 235, 547];


// Now pass this array to the bblSort() function
bblSort(arr);
```

Optimised Bubble Sort Algorithm

In the above algorithm, all the comparisons are made even if the array is already sorted.

This increases the execution time.

To solve this, we can introduce an extra variable swapped. The value of swapped is set true if swapping of elements occurs. Otherwise, it is set **false**.

After an iteration, if there is no swapping, the value of swapped will be **false**. This means elements are already sorted and there is no need to perfor further iterations.

This will reduce the execution time and helps to optimise the bubble sort.

Algorithm for optimised bubble sort is

```
bubbleSort(array)
  swapped <- false
  for i <- 1 to indexOfLastUnsortedElement-1
    if leftElement > rightElement
      swap leftElement and rightElement
      swapped <- true
end bubbleSort
```

Optimised Bubble Sort in JavaScript

```
// Optimized implementation of bubble sort Algorithm

function bubbleSort(arr){
```

```
var i, j;
var len = arr.length;

var isSwapped = false;

for(i =0; i < len; i++){

    isSwapped = false;

    for(j = 0; j < len; j++){
        if(arr[j] > arr[j + 1]){
        var temp = arr[j]
        arr[j] = arr[j+1];
        arr[j+1] = temp;
        isSwapped = true;
        }
    }

    // IF no two elements were swapped by inner loop, then break

    if(!isSwapped){
    break;
    }
}

// Print the array
console.log(arr)
}

var arr = [243, 45, 23, 356, 3, 5346, 35, 5];

// calling the bubbleSort Function
bubbleSort(arr)
```

Bubble Sort Complexity

Time Complexity	Aa
Best	O(n)
Worst	O(n^2)
Average	O(n^2)
Space Complexity	O(1)
Stability	Yes

Complexity in Detail

Bubble Sort compares the adjacent elements.

Cycle	Number of Comparisons
1st	(n-1)
2nd	(n-2)
3rd	(n-3)
....	......
last	1

Hence, the number of comparisons is

```
(n-1) + (n-2) + (n-3) +.....+ 1 = n(n-1)/2
```

nearly equals to `n^2`

Hence, **Complexity:** O(n^2)

Also, if we observe the code, bubble sort requires two loops. Hence, the complexity is `n*n = n^2`

1. Time Complexities

- **Worst Case Complexity:** `O(n^2)` If we want to sort in ascending order and the array is in descending order then the worst case occurs.
- **Best Case Complexity:** `O(n)` If the array is already sorted, then there is no need for sorting.
- **Average Case Complexity:** `O(n^2)` It occurs when the elements of the array are in jumbled order (neither ascending nor descending).

2. Space Complexity

- Space complexity is `O(1)` because an extra variable is used for swapping.
- In the **optimised bubble sort algorithm**, two extra variables are used. Hence, the space complexity will be `O(2)` .

Bubble Sort Applications

Bubble sort is used if

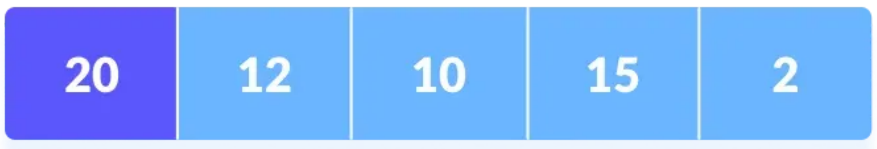
- complexity does not matter
- short and simple code is preferred

Selection Sort Algorithm

Selection sort is a sorting algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning the unsorted list.

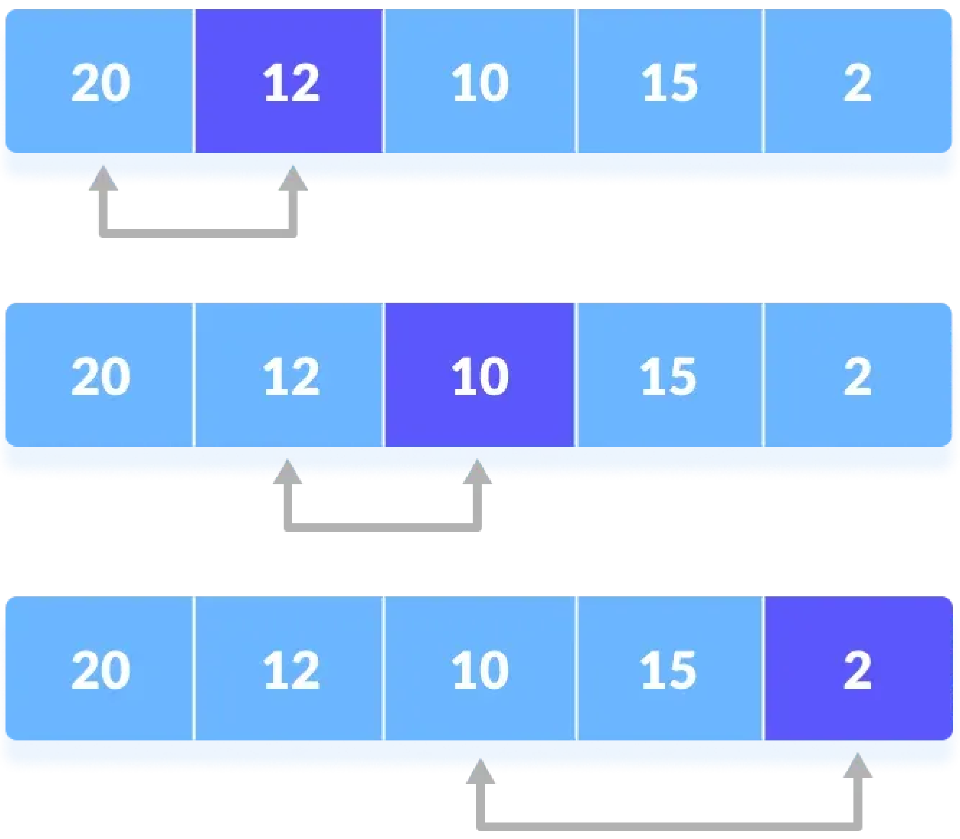
Working of Selection Sort

1. Set the first element as `minimum` .



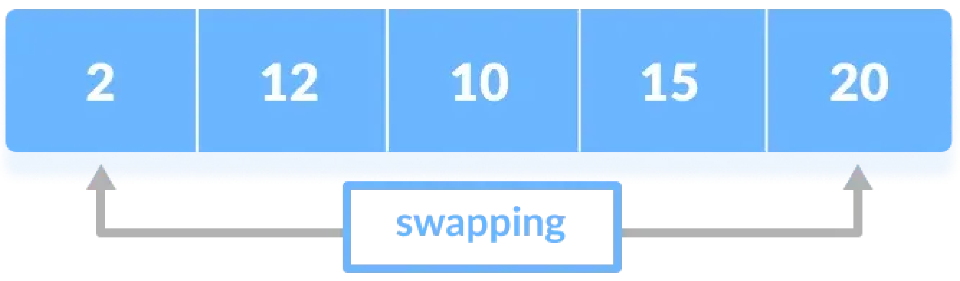
Select first element as minimum

2. 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. The process goes on until the last element.



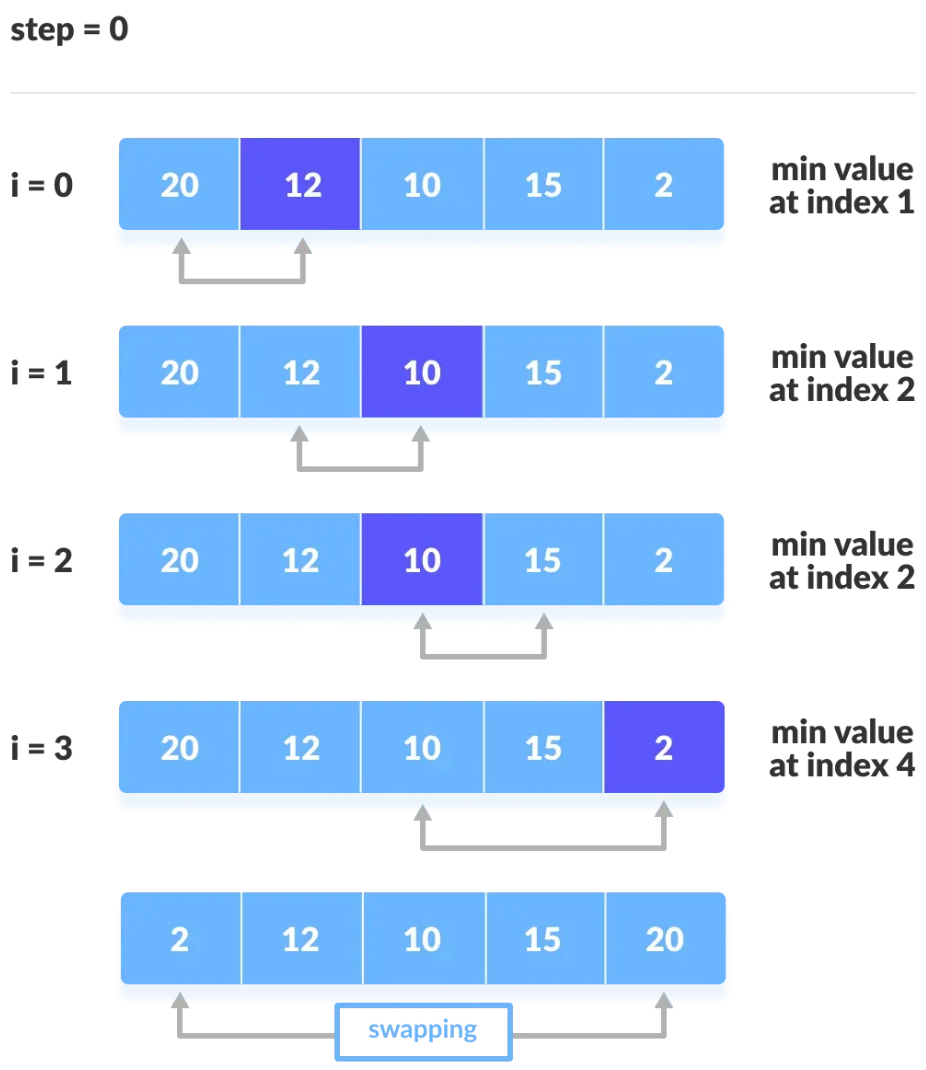
Compare minimum with the remaining elements

3. After each iteration, `minimum` is placed in the front of the unsorted list.

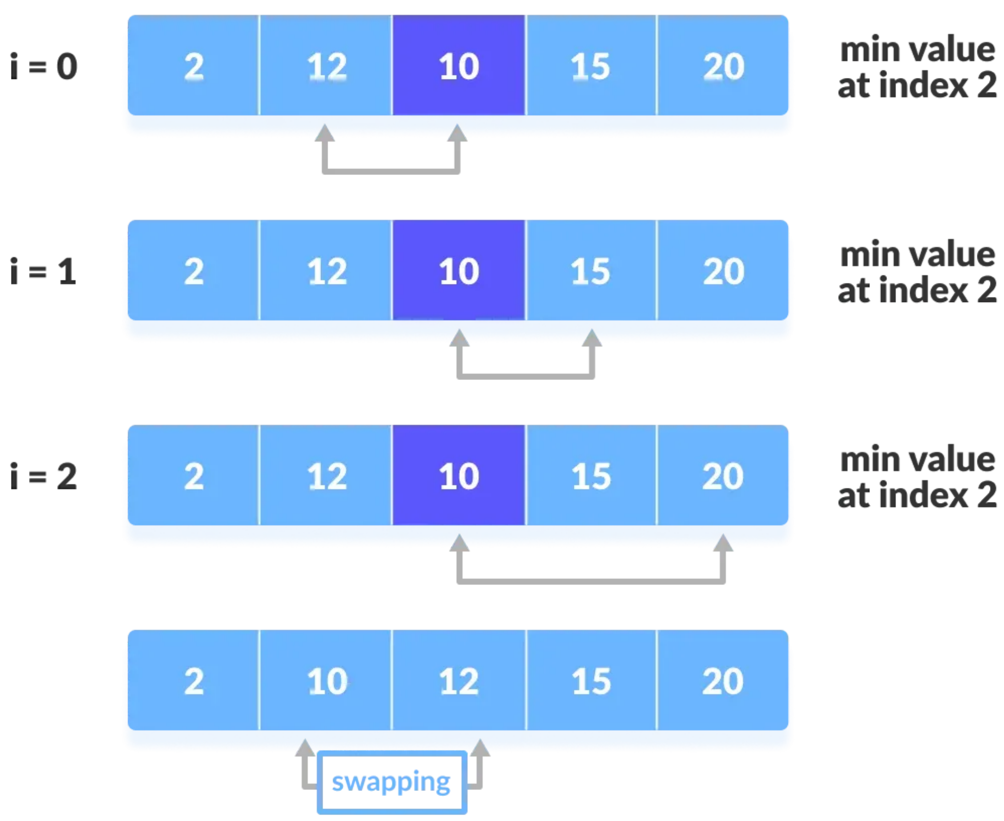


Swap the first with minimum

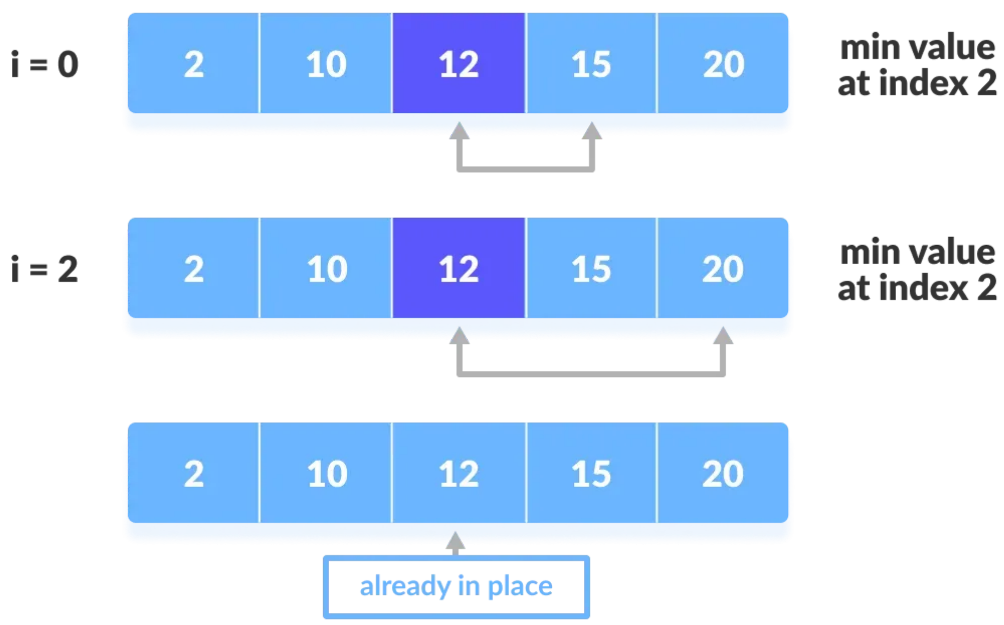
4. For each iteration, indexing starts from the first unsorted element. Step 1 to 3 are repeated until all the elements are placed at their correct positions.

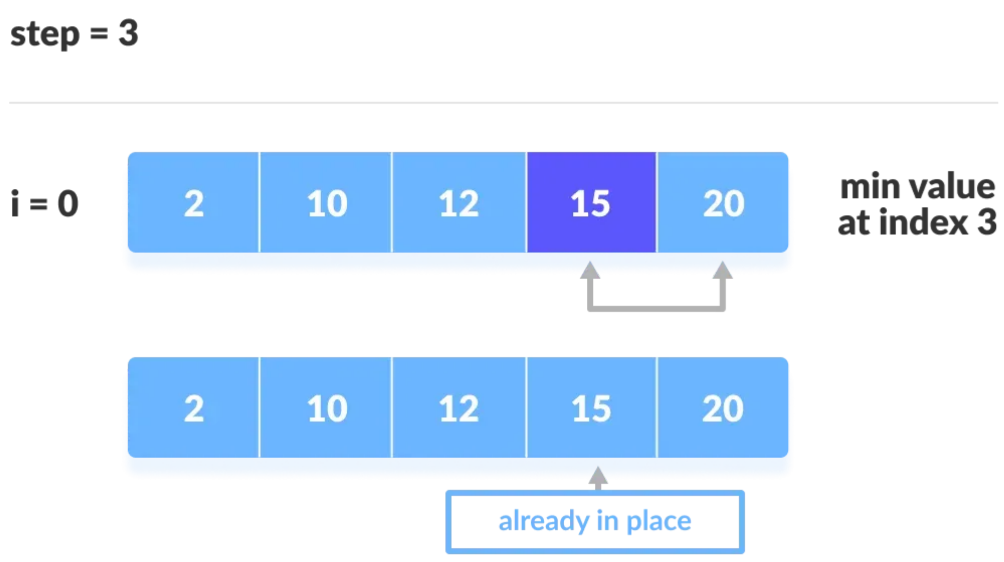

step = 0
i = 0
20
12
10
15
2
min value at index 1
i = 1
20
12
10
15
2
min value at index 2
i = 2
20
12
10
15
2
min value at index 2
i = 3
20
12
10
15
2
min value at index 4
2
12
10
15
20
swapping

step = 1


i = 0
2 12 10 15 20
min value at index 2
i = 1
2 12 10 15 20
min value at index 2
i = 2
2 12 10 15 20
min value at index 2
2 10 12 15 20
swapping

step = 2


i = 0
2 10 12 15 20
min value at index 2
i = 2
2 10 12 15 20
min value at index 2
2 10 12 15 20
already in place



Selection Sort Algorithm

```
selectionSort(array, size)
  repeat (size - 1) times
  set the first unsorted element as the minimum
  for each of the unsorted elements
    if element < currentMinimum
      set element as new minimum
  swap minimum with first unsorted position
end selectionSort
```

Selection Sort Code in JavaScript

```
function swap(arr,xp, yp)
{
    var temp = arr[xp];
    arr[xp] = arr[yp];
    arr[yp] = temp;
}

function selectionSort(arr,  n)
{
    var i, j, min_idx;

    // One by one move boundary of unsorted subarray
    for (i = 0; i < n-1; i++)
    {
        // Find the minimum element in unsorted array
        min_idx = i;
        for (j = i + 1; j < n; j++)
        if (arr[j] < arr[min_idx])
            min_idx = j;

        // Swap the found minimum element with the first element
        swap(arr,min_idx, i);
    }
}

var arr = [64, 25, 12, 22, 11];
var n = 5;
selectionSort(arr, n);
```

Selection Sort Complexity

Time Complexity	Aa
Best	O(n2)
Worst	O(n2)
Average	O(n2)
Space Complexity	O(1)
Stability	No

Aa Cycle	Number of Comparison
1st	(n-1)
2nd	(n-2)
3rd	(n-3)
...	...
last	1

Number of comparisons: `(n - 1) + (n - 2) + (n - 3) + ..... + 1 = n(n - 1) / 2` nearly equals to `n^2` .

Complexity = `O(n^2)`

Also, we can analyse the complexity by simply observing the number of loops. There are 2 loops so the complexity is `n*n = n^2` .

Time Complexities:

- **Worst Case Complexity:** `O(n^2)` If we want to sort in ascending order and the array is in descending order then, the worst case occurs.
- **Best Case Complexity:** `O(n^2)` It occurs when the array is already sorted
- **Average Case Complexity:** `O(n^2)` It occurs when the elements of the array are in jumbled order (neither ascending nor descending).

The time complexity of the selection sort is the same in all cases. At every step, you have to find the minimum element and put it in the right place. Th minimum element is not known until the end of the array is not reached.

Space Complexity:

Space complexity is `O(1)` because an extra variable `temp` is used.

Selection Sort Applications

The selection sort is used when

- a small list is to be sorted
- cost of swapping does not matter
- checking of all the elements is compulsory
- cost of writing to a memory matters like in flash memory (number of writes/swaps is `O(n)` as compared to `O(n^2)` of bubble sort)

Insertion Sort Algorithm

Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.

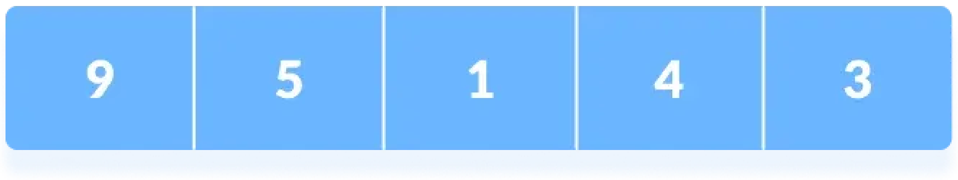
Insertion sort works similarly as we sort cards in our hand in a card game.

We assume that the first card is already sorted then, we select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on th right otherwise, to the left. In the same way, other unsorted cards are taken and put in their right place.

A similar approach is used by insertion sort.

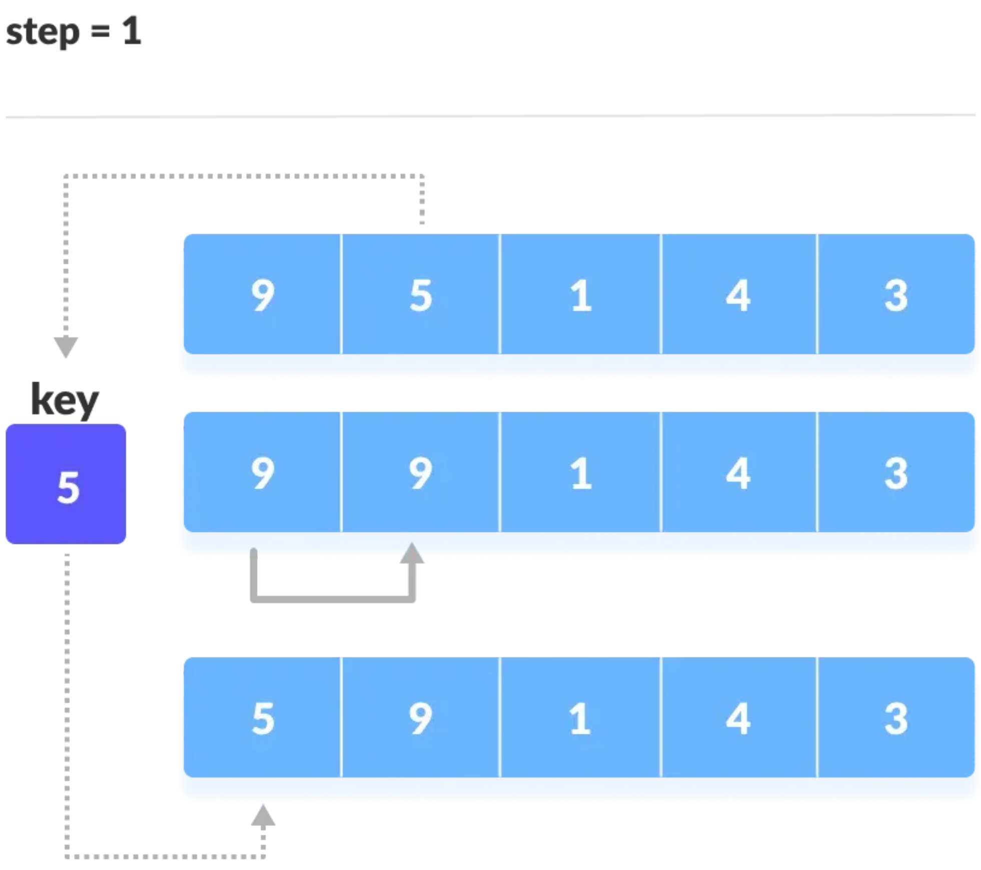
Working of Insertion Sort

Suppose we need to sort the following array.

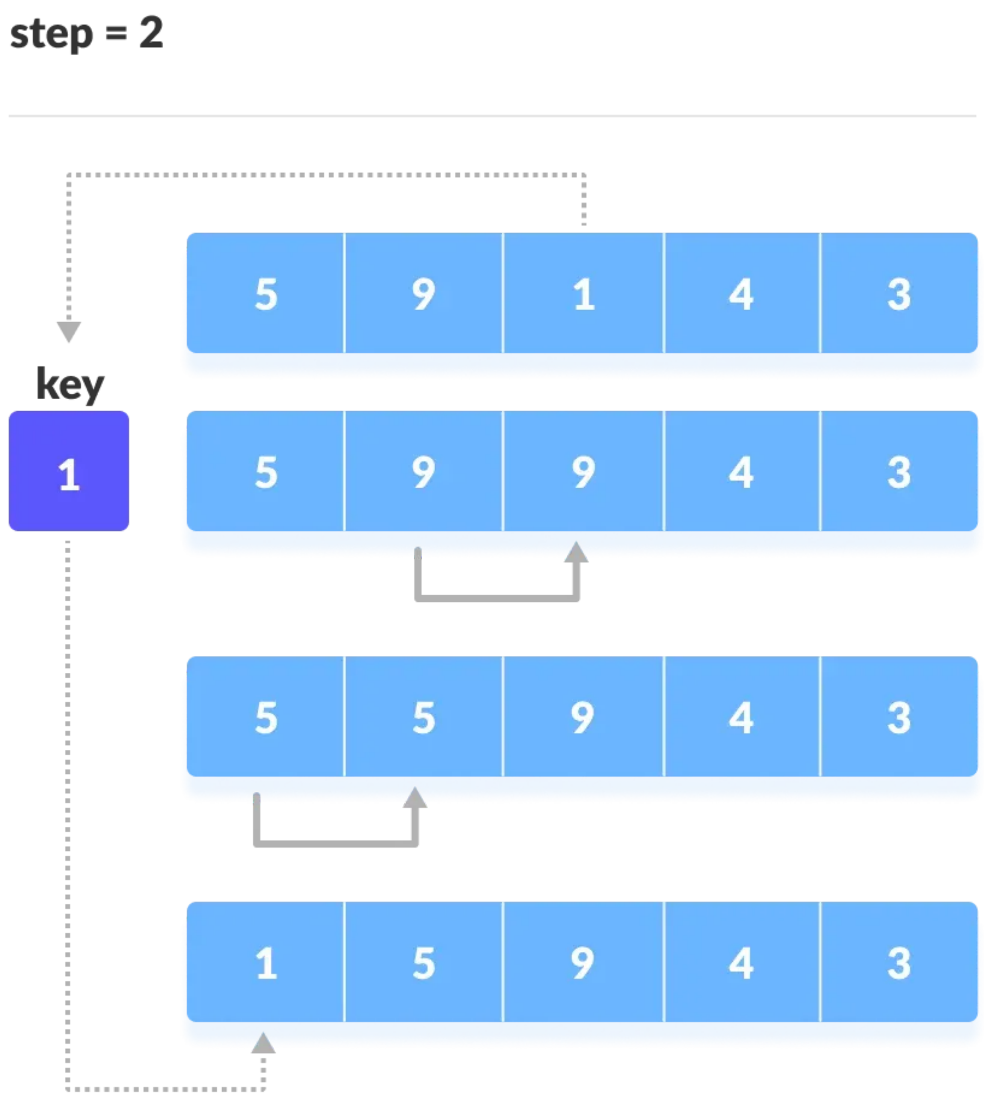


Initial array

1. The first element in the array is assumed to be sorted. Take the second element and store it separately in `key` .Compare `key` with the first element. If the first element is greater than `key` , then is placed in front of the first element.

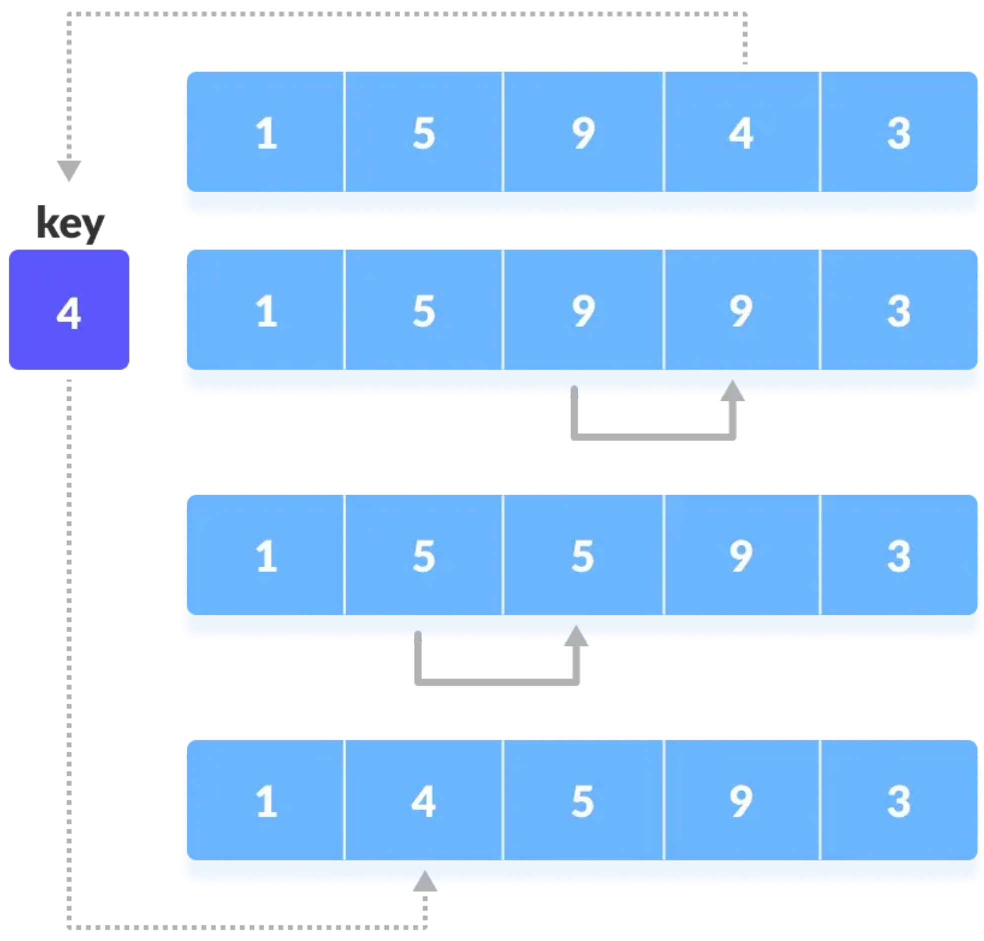


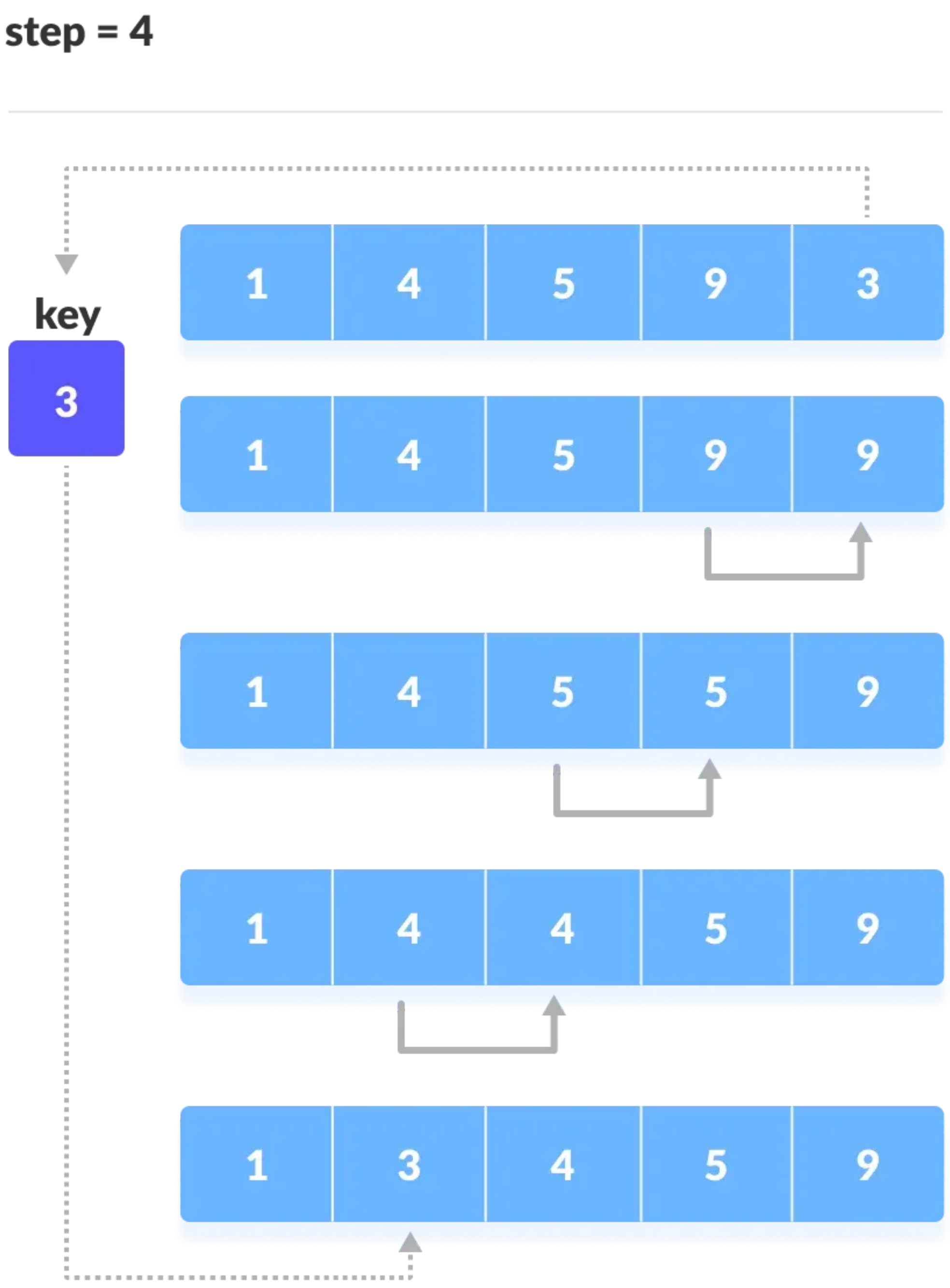
2. Now, the first two elements are sorted.Take the third element and compare it with the elements on the left of it. Placed it just behind the element smaller than it. If there is no element smaller than it, then place it at the beginning of the array.



3. Similarly, place every unsorted element at its correct position.

step = 3





Insertion Sort Algorithm

```
insertionSort(array)
  mark first element as sorted
  for each unsorted element X
    'extract' the element X
    for j <- lastSortedIndex down to 0
      if current element j > X
        move sorted element to the right by 1
    break loop and insert X here
end insertionSort
```

Insertion Sort in JavaScript

```
// Javascript program for insertion sort

// Function to sort an array using insertion sort
function insertionSort(arr, n)
{
    let i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i - 1;

        /* Move elements of arr[0..i-1], that are
        greater than key, to one position ahead
        of their current position */
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
let arr = [12, 11, 13, 5, 6 ];
let n = arr.length;
insertionSort(arr, n);
```

Insertion Sort Complexity

Time Complexity	Aa
Best	O(n)
Worst	O(n^2)
Average	O(n^2)
Space Complexity	O(1)
Stability	Yes

Time Complexities

- **Worst Case Complexity:** `O(n^2)` Suppose, an array is in ascending order, and you want to sort it in descending order. In this case, worst case complexity occurs.Each element has to be compared with each of the other elements so, for every nth element, `(n-1)` number of comparisons are made.Thus, the total number of comparisons = `n*(n-1) ~ n^2`
- **Best Case Complexity:** `O(n)` When the array is already sorted, the outer loop runs for `n` number of times whereas the inner loop does not run at all. So, there are only `n` number of comparisons. Thus, complexity is linear.
- **Average Case Complexity:** `O(n^2)` It occurs when the elements of an array are in jumbled order (neither ascending nor descending).

Space Complexity

Space complexity is `O(1)` because an extra variable `key` is used.

Insertion Sort Applications

The insertion sort is used when:

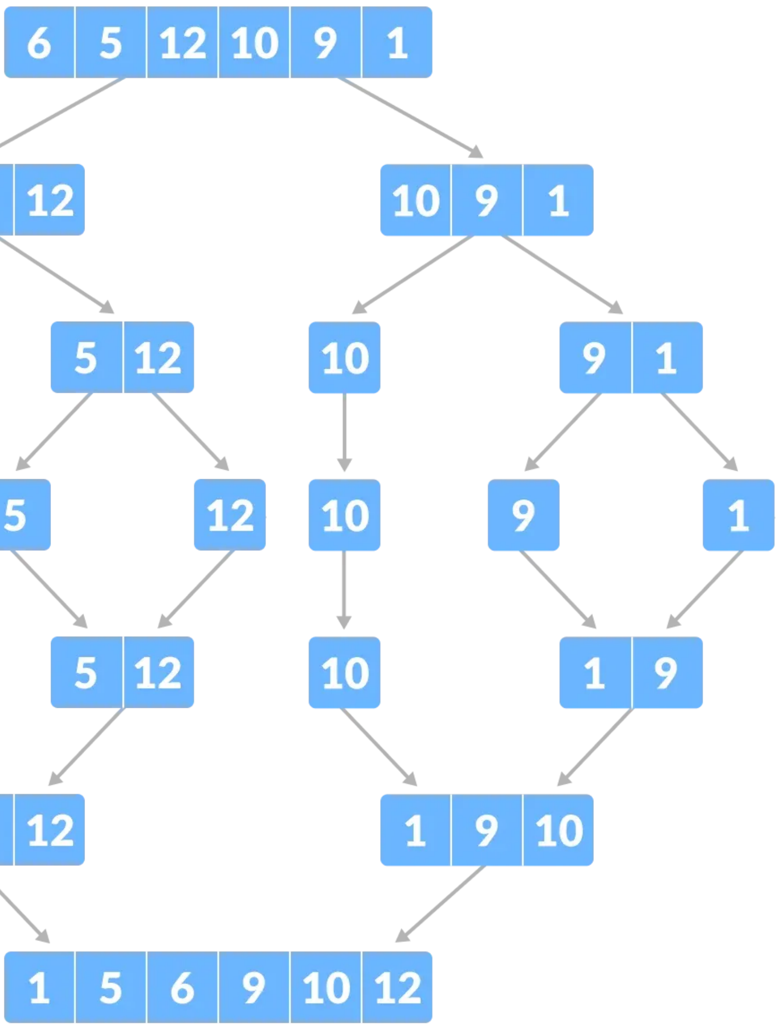
- the array is has a small number of elements

- there are only a few elements left to be sorted

Merge Sort

Merge Sort is one of the most popular sorting algorithms that is based on the principle of Divide and Conquer Algorithm.

Here, a problem is divided into multiple sub-problems. Each sub-problem is solved individually. Finally, sub-problems are combined to form the fin solution.



Divide and Conquer Strategy

Using the **Divide and Conquer** technique, we divide a problem into subproblems. When the solution to each subproblem is ready, we 'combine' tl results from the subproblems to solve the main problem.

Suppose we had to sort an array A. A subproblem would be to sort a sub-section of this array starting at index p and ending at index r, denoted as A[p..r

Divide

If q is the half-way point between p and r, then we can split the subarray A[p..r] into two arrays A[p..q] and A[q+1, r].

Conquer

In the conquer step, we try to sort both the subarrays A[p..q] and A[q+1, r]. If we haven't yet reached the base case, we again divide both these subarray and try to sort them.

Combine

When the conquer step reaches the base step and we get two sorted subarrays A[p..q] and A[q+1, r] for array A[p..r], we combine the results by creatir a sorted array A[p..r] from two sorted subarrays A[p..q] and A[q+1, r].

MergeSort Algorithm

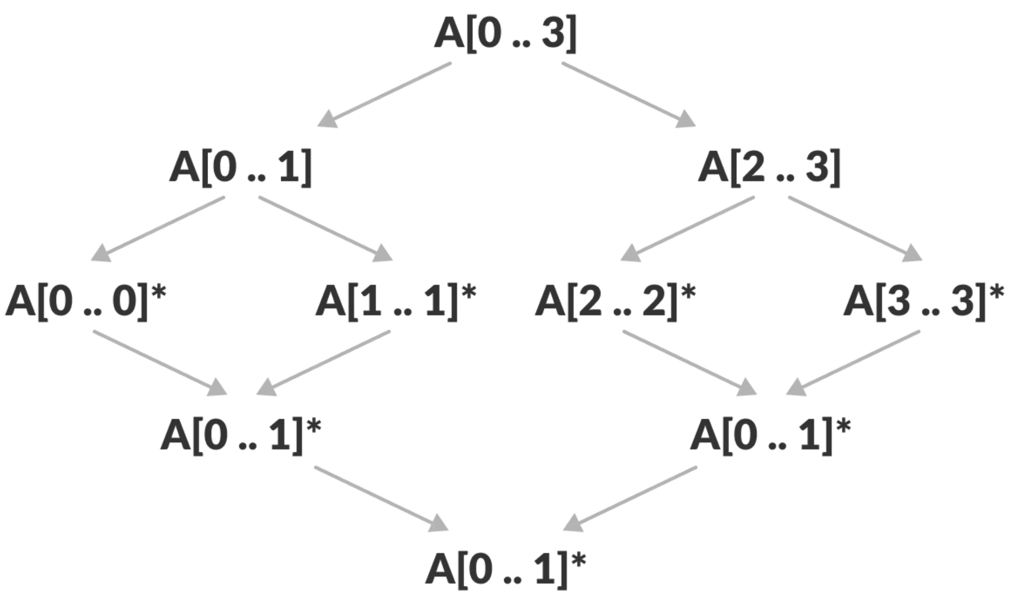
The MergeSort function repeatedly divides the array into two halves until we reach a stage where we try to perform MergeSort on a subarray of size i.e. p == r.

After that, the merge function comes into play and combines the sorted arrays into larger arrays until the whole array is merged.

```
MergeSort(A, p, r):
    if p > r
        return
    q = (p+r)/2
    mergeSort(A, p, q)
    mergeSort(A, q+1, r)
    merge(A, p, q, r)
```

To sort an entire array, we need to call `MergeSort(A, 0, length(A)-1)` .

As shown in the image below, the merge sort algorithm recursively divides the array into halves until we reach the base case of array with 1 elemer After that, the merge function picks up the sorted sub-arrays and merges them to gradually sort the entire array.



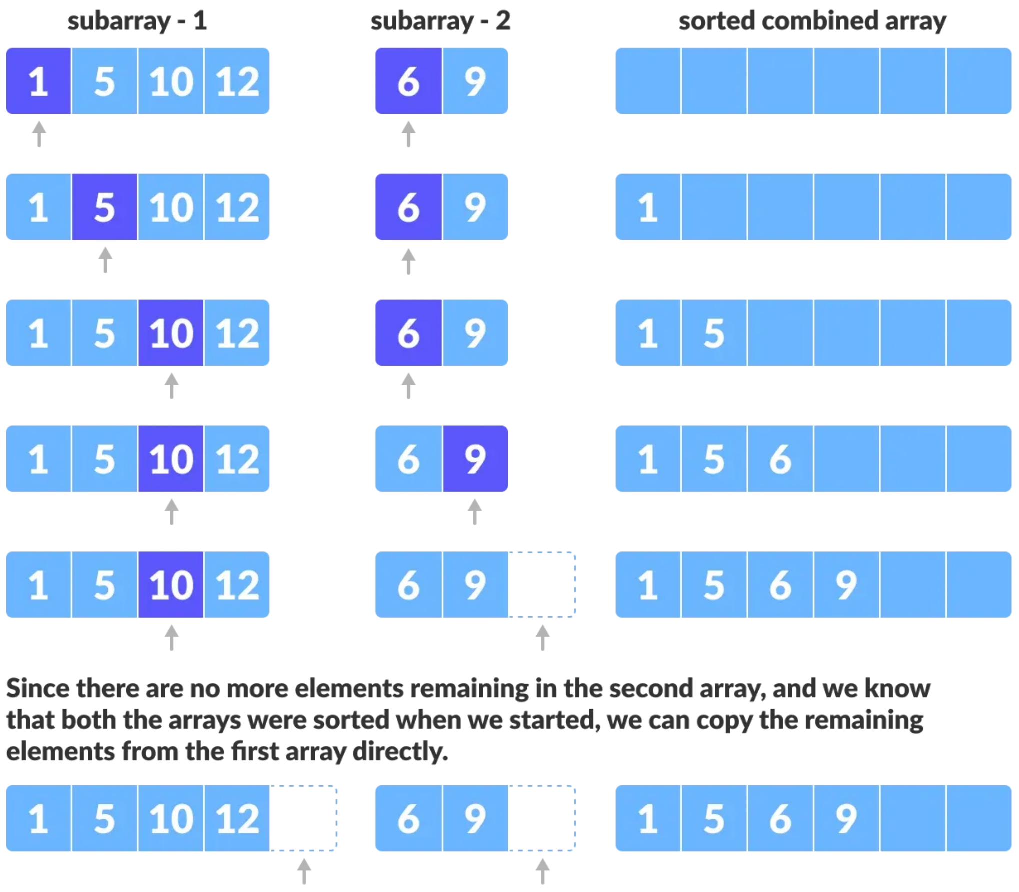
The merge Step of Merge Sort

Every recursive algorithm is dependent on a base case and the ability to combine the results from base cases. Merge sort is no different. The mo important part of the merge sort algorithm is, you guessed it, merge step.

The merge step is the solution to the simple problem of merging two sorted lists(arrays) to build one large sorted list(array).

The algorithm maintains three pointers, one for each of the two arrays and one for maintaining the current index of the final sorted array.

```
Have we reached the end of any of the arrays?
    No:
        Compare current elements of both arrays
        Copy smaller element into sorted array
        Move pointer of element containing smaller element
    Yes:
        Copy all remaining elements of non-empty array
```



Since there are no more elements remaining in the second array, and we know that both the arrays were sorted when we started, we can copy the remaining elements from the first array directly.

Writing the Code for Merge Algorithm

To implement merge sort using JavaScript, you need to first create a function that merges two arrays. Obviously, this function will accept two arrays, ar it needs to sort the two arrays correctly starting from the smallest element.

Let’s first create the function and sort the arrays as follows:

```
function merge(left, right) {
  let sortedArr = []; // the sorted elements will go here

  while (left.length && right.length) {
    // insert the smallest element to the sortedArr
    if (left[0] < right[0]) {
      sortedArr.push(left.shift());
    } else {
      sortedArr.push(right.shift());
    }
  }
}
```

When the `while` loop above is finished, you would have the elements of two arrays sorted inside the `sortedArr` variable. But since the two array length can be uneven, you may have one last element available in either `left` or `right` array:

Take a look at the following `merge()` simulation:

```
merge([3, 4, 7], [2, 5]);

function merge(left, right) {
  let sortedArr = []; // the sorted elements will go here

  while (left.length && right.length) {
    // insert the smallest element to the sortedArr
    if (left[0] < right[0]) {
      sortedArr.push(left.shift());
    } else {
      sortedArr.push(right.shift());
    }
  }

  console.log(sortedArr); // [2, 3, 4, 5]
  console.log(left); // [7]
  console.log(right); // []
}
```

As you can see above, the `left` element still has the number `7` because there's no element to compare with. To solve this problem, you can simp use the spread operator and put the leftover element into the mix:

```
function merge(left, right) {
  let sortedArr = []; // the sorted elements will go here

  while (left.length && right.length) {
    // insert the smallest element to the sortedArr
    if (left[0] < right[0]) {
      sortedArr.push(left.shift());
    } else {
      sortedArr.push(right.shift());
    }
  }

  // use spread operator and create a new array, combining the three arrays
  return [...sortedArr, ...left, ...right];
}
```

With that, you have the `merge()` function for the merge sort ready. Now you only need a function that splits an array into `1` size.

Here's an example of the `mergeSort()` function:

```
function mergeSort(arr) {
  const half = arr.length / 2;

  // the base case is array length <=1
  if (arr.length <= 1) {
    return arr;
  }

  const left = arr.splice(0, half); // the first half of the array
  const right = arr;
  return merge(mergeSort(left), mergeSort(right));
}
```

The `mergeSort()` function will first split the given array parameter in half until the array length is one or smaller. The arrays will then be passed the `merge()` function, which will start merging the arrays until all elements are merged.

And that's how you implement a merge sort with JavaScript.

Merge Sort Complexity

Time Complexity	Aa
Best	O(n*log n)
Worst	O(n*log n)
Average	O(n*log n)
Space Complexity	O(n)
Stability	Yes

Time Complexity

Best Case Complexity: O(n*log n)

Worst Case Complexity: O(n*log n)

Average Case Complexity: O(n*log n)

Space Complexity

The space complexity of merge sort is O(n).

Merge Sort Applications

- Inversion count problem
- External sorting
- E-commerce applications

Visualisation of Sorting Algorithms

Click Here

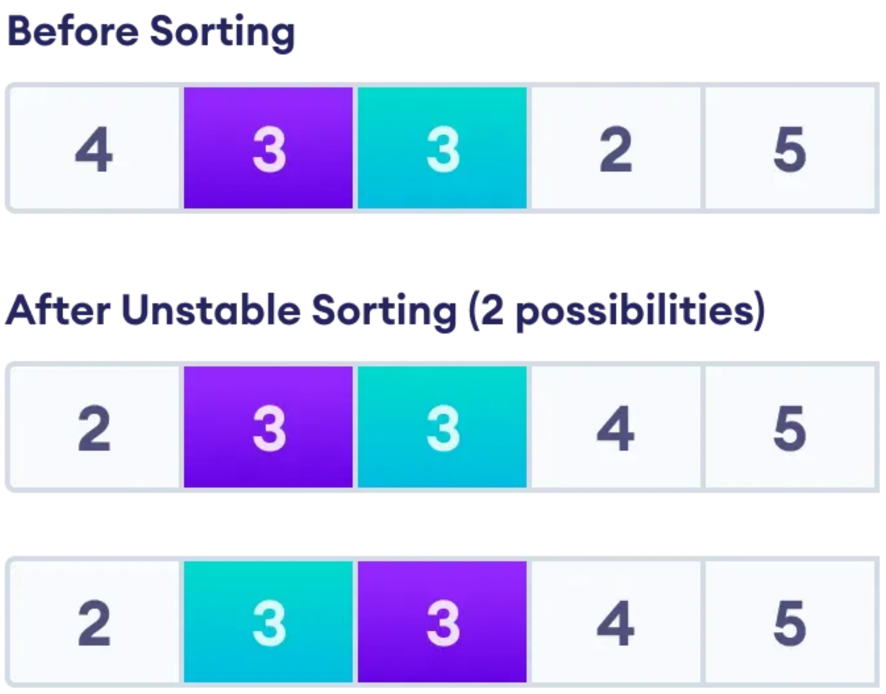
Complexity of Sorting Algorithms

Sorting Algorithm	Time Complexity - Best	Time Complexity - Worst	Time Complexity - Average	Space Complexity
Bubble Sort	n	n^2	n^2	1
Selection Sort	n^2	n^2	n^2	1
Insertion Sort	n	n^2	n^2	1
Merge Sort	nlog n	nlog n	nlog n	n

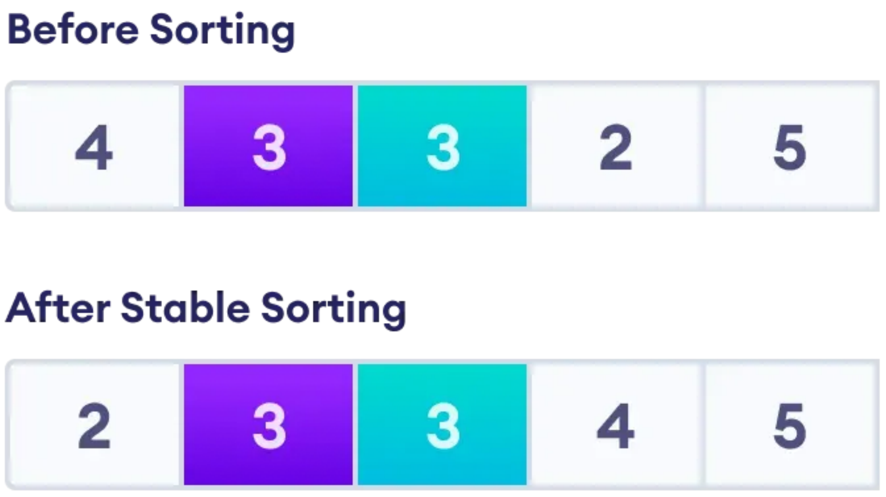
Stability of Sorting Algorithm

A sorting algorithm is considered stable if the two or more items with the same value maintain the same relative positions even after sorting.

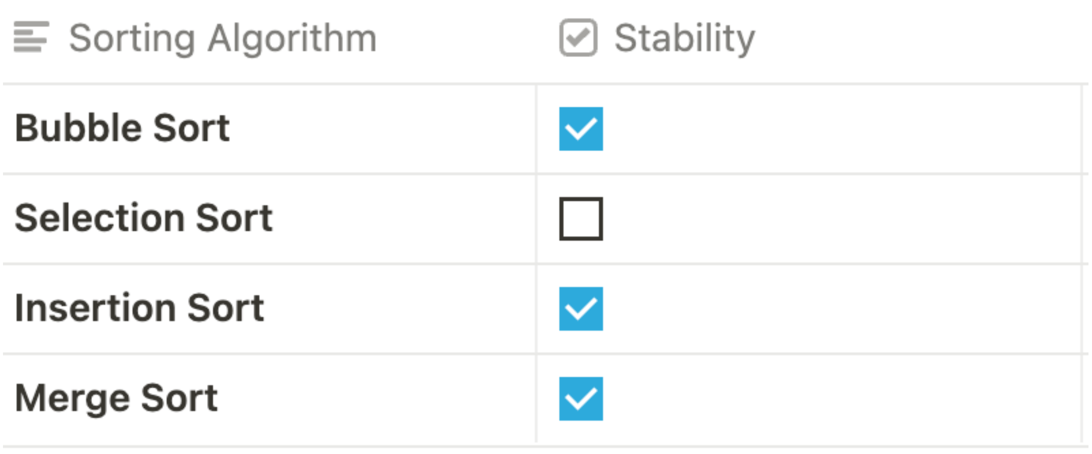
For example, in the image below, there are two items with the same value 3. An unstable sorting algorithm allows two possibilities where the tw
positions of 3 may or may not be maintained.



However, after a stable sorting algorithm, there is always one possibility where the positions are maintained as in the original array.



Here's a table showing the stability of different sorting algorithm.



Sorting Algorithm	Stability
Bubble Sort	☑
Selection Sort	☐
Insertion Sort	☑
Merge Sort	☑

Divide and Conquer Algorithm

A **divide and conquer algorithm** is a strategy of solving a large problem by

1. breaking the problem into smaller sub-problems
2. solving the sub-problems, and
3. combining them to get the desired output.

To use the divide and conquer algorithm, **recursion** is used. Recursion will be discussed in the next class in more detail.

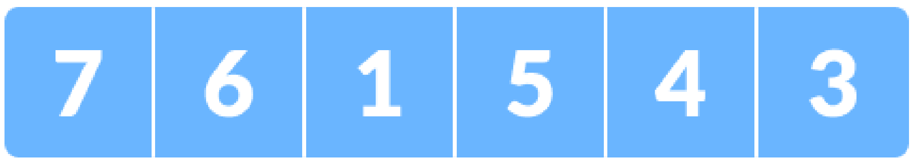
How Divide and Conquer Algorithms Work?

Here are the steps involved:

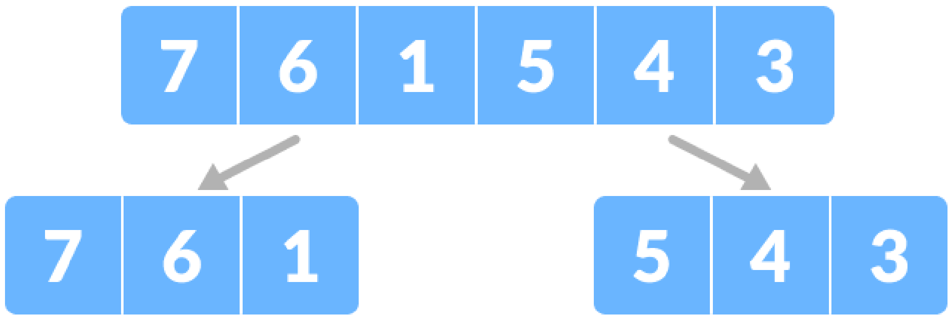
1. **Divide**: Divide the given problem into sub-problems using recursion.
2. **Conquer**: Solve the smaller sub-problems recursively. If the subproblem is small enough, then solve it directly.
3. **Combine:** Combine the solutions of the sub-problems that are part of the recursive process to solve the actual problem.

Let us understand this concept with the help of an example.
Here, we will sort an array using the divide and conquer approach (ie. merge sort).

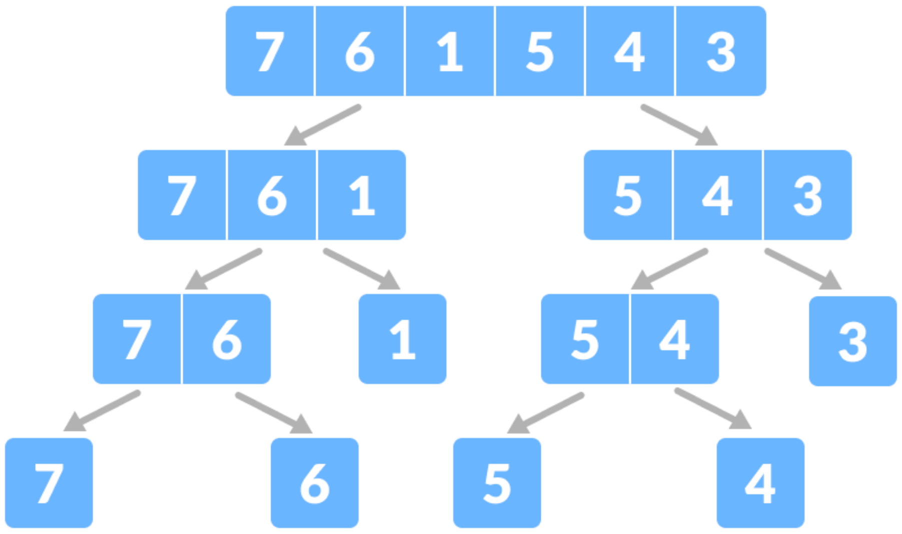
1. Let the given array be:



2. Divide the array into two halves.Again, divide each subpart recursively into two halves until you get individual elements.



Divide the array into two subparts



3. Now, combine the individual elements in a sorted manner. Here, conquer and combine steps go side by side.

Time Complexity

The complexity of the divide and conquer algorithm is calculated using the master theorem.

```
T(n) = aT(n/b) + f(n),
where,
n = size of input
a = number of subproblems in the recursion
n/b = size of each subproblem. All subproblems are assumed to have the same size.
f(n) = cost of the work done outside the recursive call, which includes the cost of dividing the problem and cost of mer
```

Let us take an example to find the time complexity of a recursive problem.

For a merge sort, the equation can be written as:

```
T(n) = aT(n/b) + f(n)
     = 2T(n/2) + O(n)
Where,
a = 2 (each time, a problem is divided into 2 subproblems)
n/b = n/2 (size of each sub problem is half of the input)
f(n) = time taken to divide the problem and merging the subproblems
T(n/2) = O(n log n) (To understand this, please refer to the master theorem.)
```

```
Now, T(n) = 2T(n log n) + O(n)
          ≈ O(n log n)
```

Example of Divide and Conquer Algorithm

- Binary Search
- Merge Sort
- Quick Sort
- Strassen's Matrix multiplication
- Karatsuba Algorithm

All these examples will be discussed in subsequent classes.

Thank You !