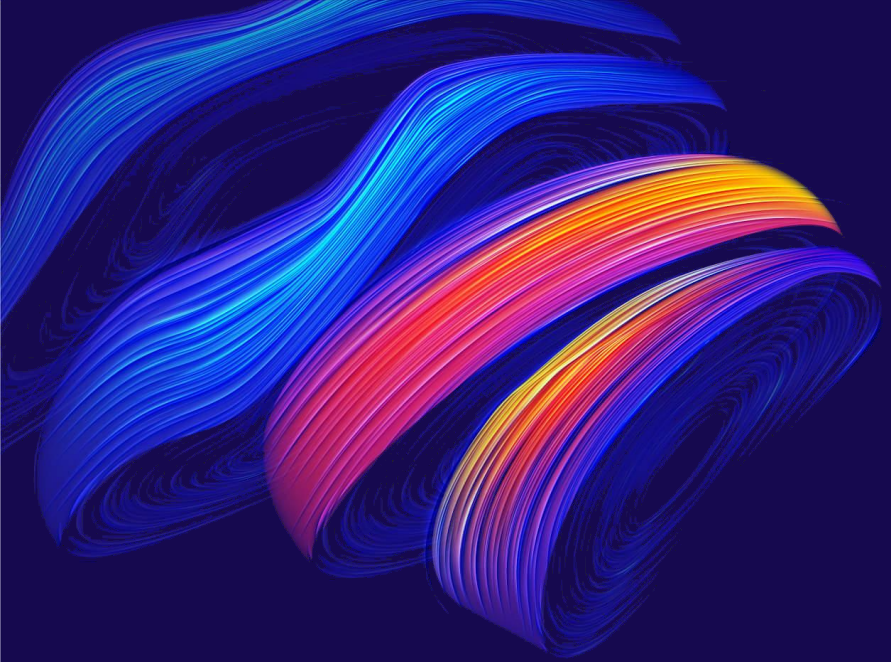


Algorithms and Data Structures Using Java

– Soumya

Searching Algorithms

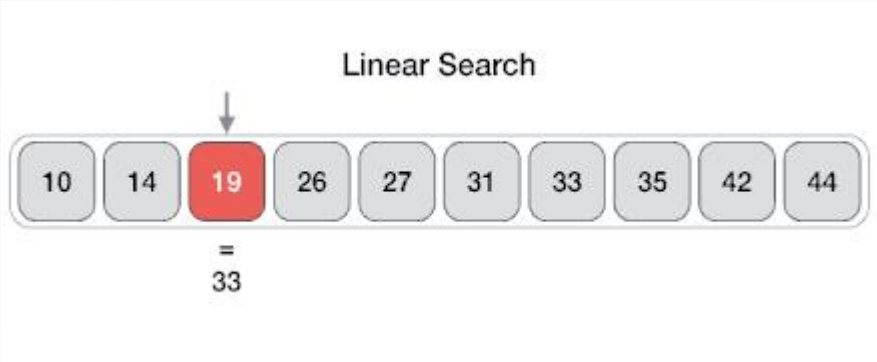
- A search algorithm is an algorithm designed to solve a search problem. Search algorithms work to retrieve information stored within particular data structure, or calculated in the search space of a problem domain, with either discrete or continuous values.
- Although search engines use search algorithms, they belong to the study of information retrieval, not algorithmics.
- The appropriate search algorithm often depends on the data structure being searched, and may also include prior knowledge about the data.
- Search algorithms can be made faster or more efficient by specially constructed database structures, such as search trees, hash maps, and database indexes.

Basic Searching Algorithms

- Linear Search
- Binary Search

Linear Search

- Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one.
- Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.



Algorithm

- Linear Search (Array A, Value x)
  - Step 1: Set i to 1
  - Step 2: if i > n then go to step 7
  - Step 3: if A[i] = x then go to step 6
  - Step 4: Set i to i + 1
  - Step 5: Go to Step 2
  - Step 6: Print Element x Found at index i and go to step 8
  - Step 7: Print element not found
  - Step 8: Exit

```
public class LinearSearch {

    public static int linearSearch(int[] arr, int target) {

        for (int i = 0; i < arr.length; i++) {
            if (arr[i] == target) {
                return i;  // Element found at index i
            }
        }
        return -1;  // Element not found
    }

    public static void main(String[] args) {
        int[] numbers = {10, 20, 30, 40, 50};
        int target = 30;

        int result = linearSearch(numbers, target);

        if (result != -1)
            System.out.println("Element found at index: " + result);
        else
            System.out.println("Element not found.");
    }
}
```

Complexity

- Sequential search is a search algorithm that examines each element of a list one by one until the desired element is found or the end of the list is reached.
- The time complexity of sequential search is O(n), which means that the search time grows linearly with the size of the list.
- This is because the search algorithm has to examine each element of the list until the desired element is found or the end of the list is reached.

Binary Search

- Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.
- Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item.
- Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub- array as well until the size of the subarray reduces to zero

Binary Search

- For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example.
- The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.

- First, we shall determine half of the array by using this formula −
- mid = low + (high - low) / 2

Binary Search

Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

- We change our low to mid + 1 and find the new mid value again.

low = mid + 1

mid = low + (high - low) / 2

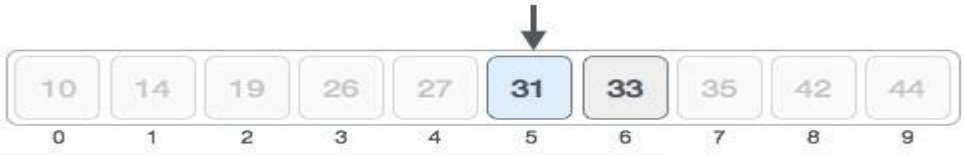
Binary Search

• Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

• The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.

Binary Search

- Hence, we calculate the mid again. This time it is 5.



- We compare the value stored at location 5 with our target value. We find that it is a match.

- We conclude that the target value 31 is stored at location 5.
- Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

```
public class BinarySearch {
    public static int binarySearch(int[] arr, int target) {
        int low = 0;
        int high = arr.length - 1;

        while (low <= high) {
            int mid = (low + high) / 2;

            if (arr[mid] == target)
                return mid; // Element found
            else if (arr[mid] < target)
                low = mid + 1;  // Search right half
            else
                high = mid - 1; // Search left half
        }

        return -1; // Element not found
    }

    public static void main(String[] args) {
        int[] numbers = {10, 20, 30, 40, 50, 60, 70};
        int target = 40;

        int result = binarySearch(numbers, target);

        if (result != -1)
            System.out.println("Element found at index: " + result);
        else
            System.out.println("Element not found.");
    }
}
```

Complexity

- Time Complexity

– The time complexity of binary search is O(log n), where n is the number of elements in the sorted array.

– This means that the number of comparisons required to find an element in the array grows logarithmically with the size of the array.

Sorting Algorithms

- Sorting refers to arranging data in a particular format.
- Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.
- The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner.
- Sorting is also used to represent data in more readable formats.

Sorting Algorithms:Examples

- Since sorting is often capable of reducing the algorithmic complexity of some particular types of problems, it has found its usage in a wide range of fields.
- Dictionary: The dictionary stores words in alphabetical order so that searching for any word becomes easy.
- Telephone Book: The telephone book stores the telephone numbers of people sorted by their names in alphabetical order, so that the names can be searched easily.
- E-commerce application: While shopping through any e-commerce application like amazon or flipkart, etc. , we preferably tend to sort our items based on their price range, popularity, size, etc.

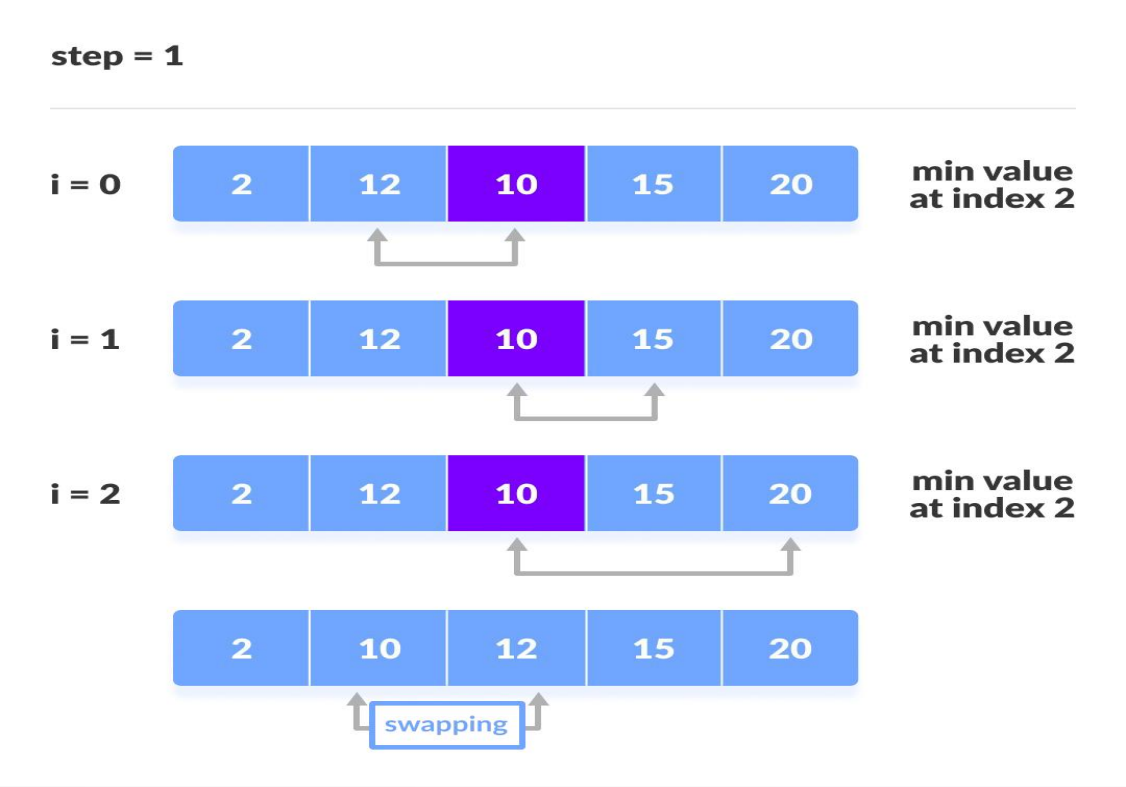
Sorting Algorithms

- Selection Sort
- Insertion Sort
- Bubble Sort
- Quick Sort
- Heap Sort
- Merge Sort

Selection Sort

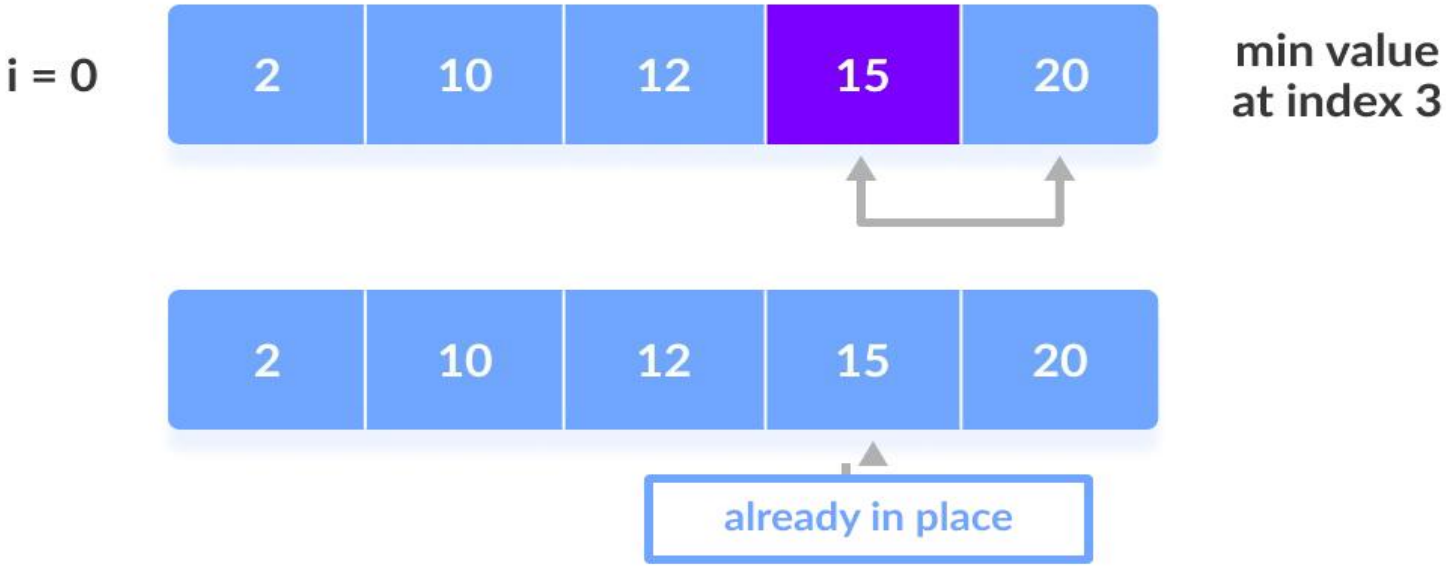
- Selection sort is a sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list.
- Algorithm
  - Start with the first unsorted list and find the smallest (or largest) element in the remaining unsorted list.
  - Swap the first unsorted list with the smallest (or largest) element.
  - Repeat steps 1 and 2 until all elements are sorted.

Selection Sort



Selection Sort

step = 3


i = 0
2
10
12
15
20
min value
at index 3
2
10
12
15
20
already in place

```
class SelectionSort
{
    void selectionSort(int arr[])
    {
        int size = arr.length;

        // loop to iterate over the entire array
        for (int i = 0; i < size - 1; i++)
        {
            // set minIndex equal to the first unsorted element
            int minIndex = i;

          //iterate over unsorted sublist
            for (int j = i+1; j < size; j++)
      //helps in finding the minimum element
                if (arr[j] < arr[minIndex])
                    minIndex = j;

 // swapping the minimum element with the element at minIndex to place it at its proper position
            int temp = arr[minIndex];
            arr[minIndex] = arr[i];
            arr[i] = temp;
        }
    }
}
```

```
// method to print array
    void printArray(int arr[]) {
        for (int value : arr) {
            System.out.print(value + " ");
        }
        System.out.println();
    }

    // main method
    public static void main(String[] args) {
        SelectionSort sorter = new SelectionSort();
        int[] arr = {34, 55, 12, 22, 11};

        System.out.println("Before sorting:");
        sorter.printArray(arr);

        sorter.selectionSort(arr);

        System.out.println("After sorting:");
        sorter.printArray(arr);
    }
}
```

Complexity

- The time complexity of selection sort is O(n^2) in the worst case, average case, and best case.
- This is because the algorithm needs to compare each element in the array to every other element in the array to find the smallest (or largest) element.

Complexity

- Advantages

– Selection sort is a simple and easy-to-implement sorting algorithm. It is also a stable sorting algorithm, which means that it preserves the original order of equal elements in the array.

- Disadvantages

– Selection sort is a very inefficient sorting algorithm, with a time complexity of O(n^2). It is not recommended for sorting large arrays.

Insertion Sort

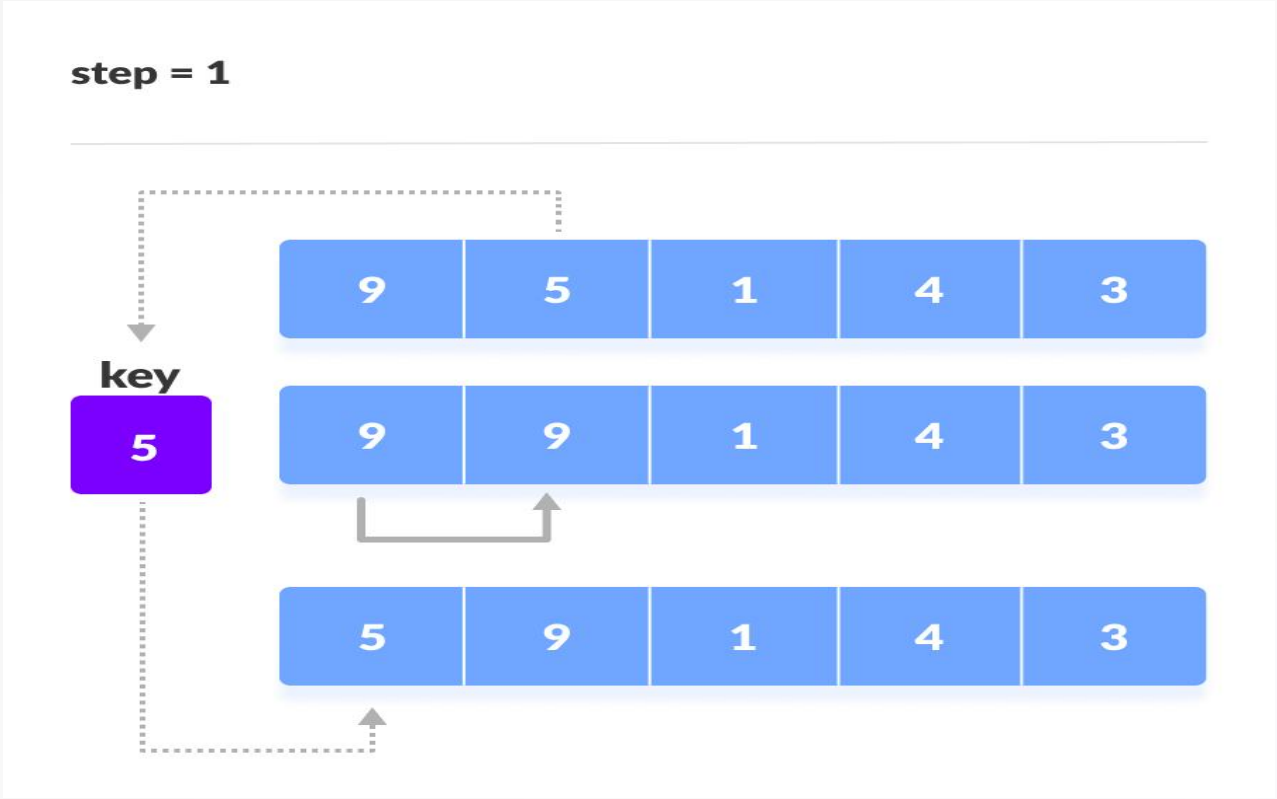
- Insertion sort is a simple sorting algorithm that works by repeatedly inserting an unsorted element into its sorted position in an array.
- The algorithm starts by comparing the unsorted element to the first element in the array.
- If the unsorted element is less than or equal to the first element, it is inserted at the beginning of the array.
- Otherwise, the algorithm continues comparing the unsorted element to the remaining elements in the array until it finds a sorted position for the element.

Insertion Sort

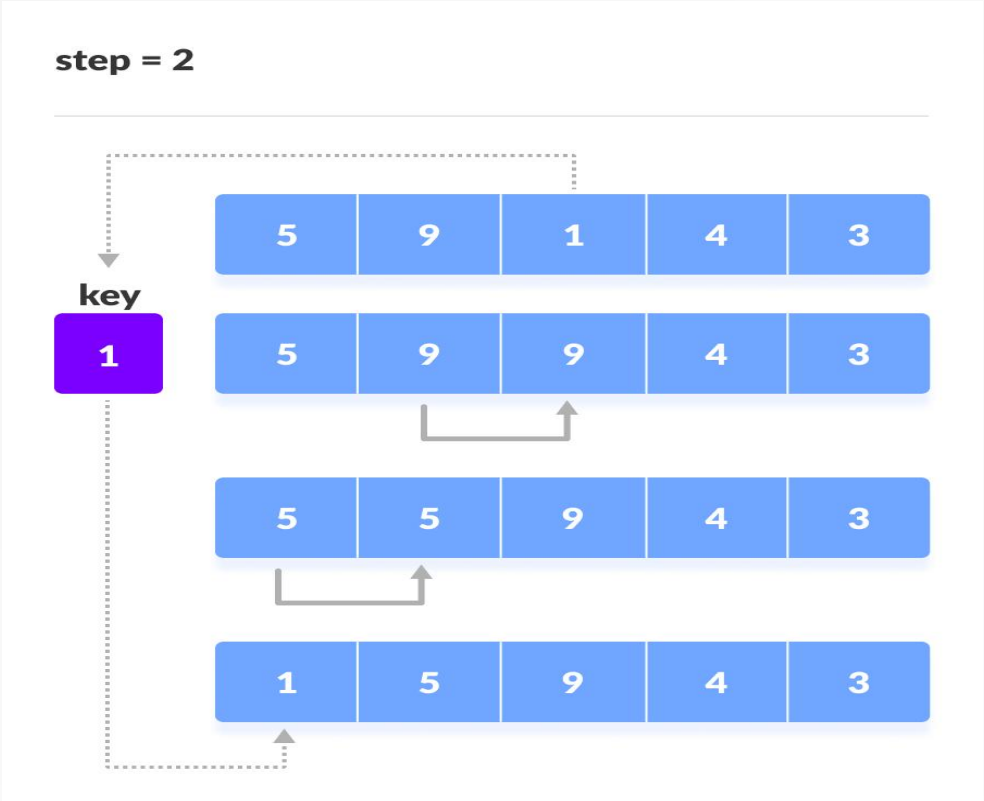
- insertionSort(array)
- mark first element as sorted
- for each unsorted element X
- 'extract' the element X
- for j <- lastSortedIndex down to 0
- if current element j > X
- move sorted element to the right by 1
- break loop and insert X here
- end insertionSort

Insertion Sort

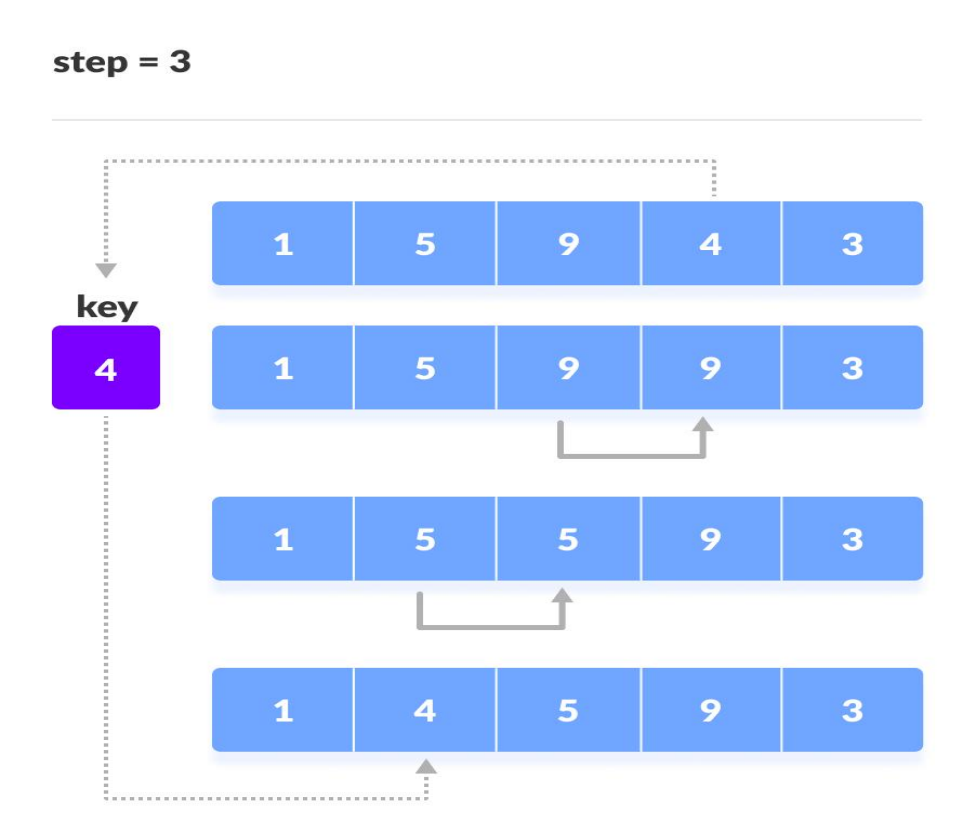
Insertion Sort



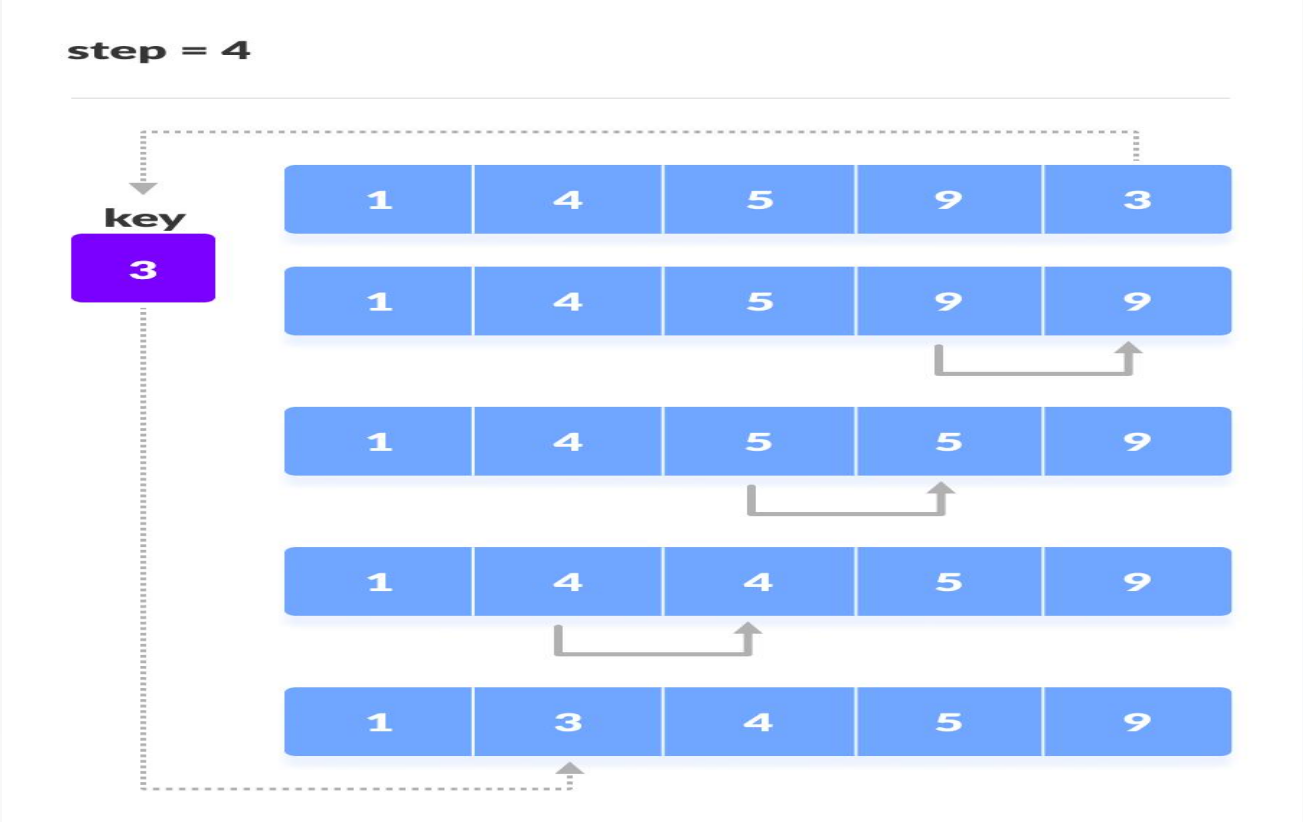
Insertion Sort



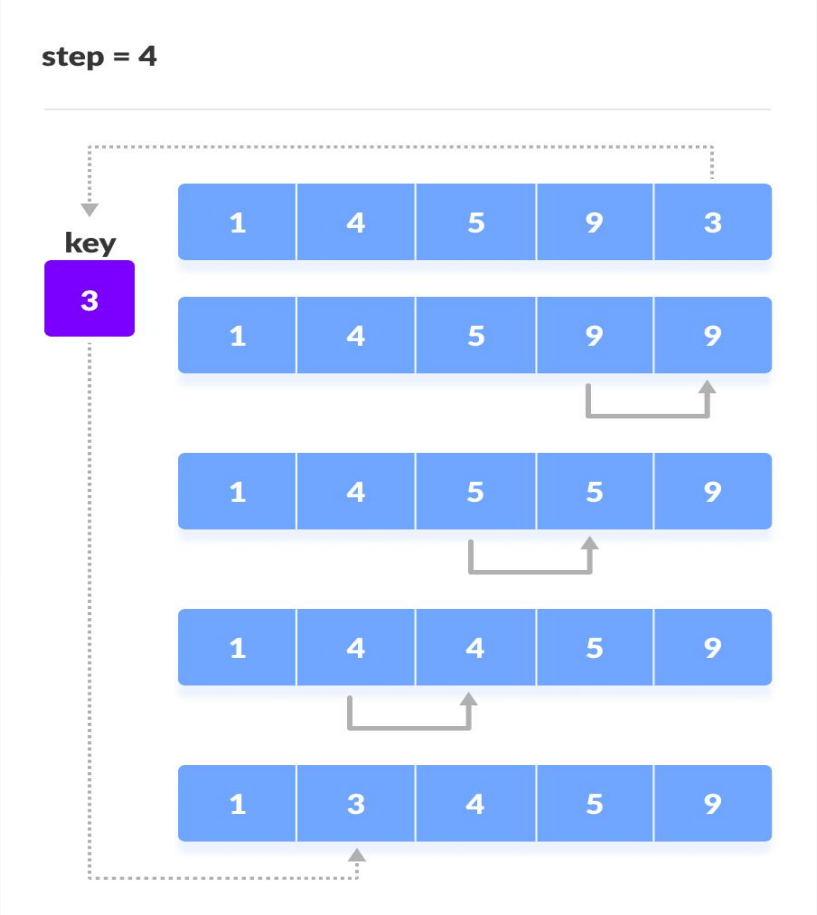
Insertion Sort



Insertion Sort



Insertion Sort



Bubble Sort

- Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order.
- This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n2)where n is the number of items.

Bubble Sort: Working

- We take an unsorted array for our example. Bubble sort takes O(n2) time so we're keeping it short and precise.

- Bubble sort starts with very first two elements, comparing them to check which one is greater.

- In this case, value 33 is greater than 14, so it is already in sorted locations.

Bubble Sort: Working

- Next, we compare 33 with 27.

- We find that 27 is smaller than 33 and these two values must be swapped.

- The new array should look like this −

Bubble Sort: Working

• Next we compare 33 and 35. We find that both are in already sorted positions.

• Then we move to the next two values, 35 and 10.

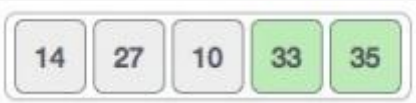
• We know then that 10 is smaller 35. Hence they are not sorted.

Bubble Sort: Working

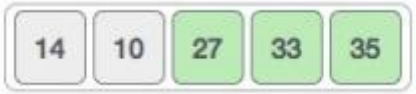
- We swap these values. We find that we have reached the end of the array.

  After one iteration, the array should look like this −

- To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −



- Notice that after each iteration, at least one value moves at the end.



- And when there's no swap required, bubble sorts learns that an array is completely sorted.

Bubble Sort: Working

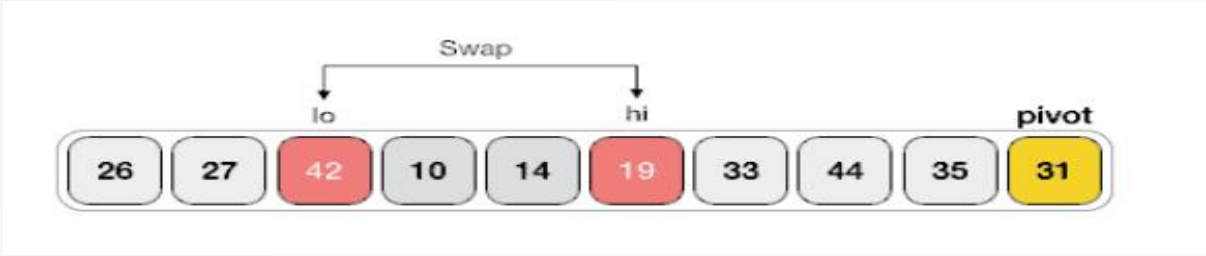
• We assume list is an array of n elements. We further assume that swap function swaps the values of the given array elements.

```
begin BubbleSort(list)

for all elements of list

if list[i] > list[i+1]

swap(list[i], list[i+1])

end if

end for

return list

end BubbleSort
```

Quick Sort

- Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays.
- A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.
- Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays.
- This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are O(n2),respectively.

Quick Sort

- 

- The pivot value divides the list into two parts. And recursively, we find the pivot for each sub-lists until all lists contains only one element.

Quick Sort

- Step 1 − Choose the highest index value has pivot
- Step 2 − Take two variables to point left and right of the list excluding pivot
- Step 3 − left points to the low index
- Step 4 − right points to the high
- Step 5 − while value at left is less than pivot move right
- Step 6 − while value at right is greater than pivot move left
- Step 7 − if both step 5 and step 6 does not match swap left and right
- Step 8 − if left ≥ right, the point where they met is new pivot

```
QuickSort(arr, low, high)
    if low < high then
        pivot_index = Partition(arr, low, high)
        QuickSort(arr, low, pivot_index - 1)
        QuickSort(arr, pivot_index + 1, high)

Partition(arr, low, high)
    pivot = arr[high]
    i = low - 1
    for j from low to high - 1 do
        if arr[j] <= pivot then
            i = i + 1
            swap arr[i] with arr[j]
    swap arr[i + 1] with arr[high]
    return i + 1
```

Partition Algorithm

The key process in **quickSort** is a **partition().** There are three common algorithms to partition. All these algorithms have O(n) time complexity.

1.Naive Partition: Here we create copy of the array. First put all smaller elements and then all greater. Finally we copy the temporary array back to original array. This requires O(n) extra space.

2.Lomuto Partition: We have used this partition in this article. This is a simple algorithm, we keep track of index of smaller elements and keep swapping. We have used it here in this article because of its simplicity.

3.Hoare's Partition: This is the fastest of all. Here we traverse array from both sides and keep swapping greater element on left with smaller on right while the array is not partitioned.

Quick Sort

1.Choose a Pivot: Select an element from the array as the pivot. The choice of pivot can vary (e.g., first element, last element, random element, or median).

2.Partition the Array: Rearrange the array around the pivot. After partitioning, all elements smaller than the pivot will be on its left, and all elements greater than the pivot will be on its right. The pivot is then in its correct position, and we obtain the index of the pivot.

3.Recursively Call: Recursively apply the same process to the two partitioned sub-arrays (left and right of the pivot).

4.Base Case: The recursion stops when there is only one element left in the sub-array, as a single element is already sorted.

Merge Sort

- Merge sort is a sorting technique based on divide and conquer technique.
- With worst-case time complexity being O(n logn), it is one of the most respected algorithms.
- Merge sort first divides the array into equal halves and then combines them in a sorted manner.

Merge Sort

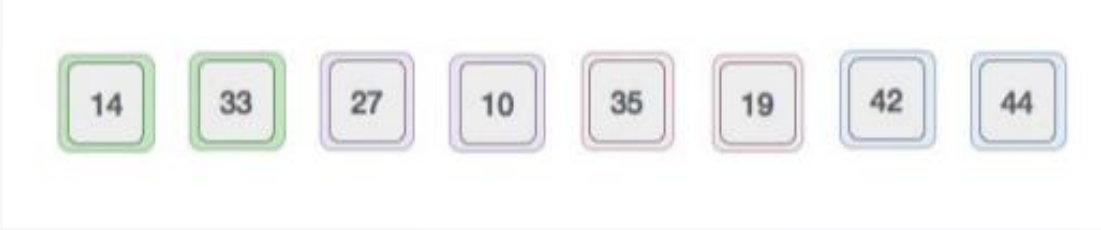
To understand merge sort, we take an unsorted array as the following −

- We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.

Merge Sort

- This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

- We further divide these arrays and we achieve atomic value which can no more be divided.



Merge Sort

- Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.
- We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.

Merge Sort

- In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.

- After the final merging, the list should look like this −

- Now we should learn some programming aspects of merge sorting.

Algorithm

• Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted.

• Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

– Step 1 − if it is only one element in the list it is already sorted, return.

– Step 2 − divide the list recursively into two halves until it can no more be divided.

– Step 3 − merge the smaller lists into new list in sorted order.

Heap Sort

- Heap sort processes the elements by creating the min- heap or max-heap using the elements of the given array.
- Min-heap or max-heap represents the ordering of array in which the root element represents the minimum or maximum element of the array.
- Heap sort basically recursively performs two main operations -

– Build a heap H, using the elements of array.

– Repeatedly delete the root element of the heap formed in 1st phase.

What is Heap?

- A heap is a complete binary tree, and the binary tree is a tree in which the node can have the utmost two children.
- A complete binary tree is a binary tree in which all the levels except the last level, i.e., leaf node, should be completely filled, and all the nodes should be left-justified.

What is Heap Sort?

- Heapsort is a popular and efficient sorting algorithm.
- The concept of heap sort is to eliminate the elements one by one from the heap part of the list, and then insert them into the sorted part of the list.
- Heapsort is the in-place sorting algorithm.

- Build a Heap: First, we arrange the list of numbers into a heap. This makes sure the largest number is at the top of the heap.
- Remove the Top: Then, we remove the top (the largest number) and place it at the end of the list.
- Rebuild the Heap: After removing the top, we rebuild the heap with the remaining numbers.
- Repeat: We keep repeating the process of removing the top and rebuilding the heap until all numbers are sorted.

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)