Sorting Algorithms

Sorting: Arranging data in a specific order (ascending or descending)

Applications of Sorting Algorithms

- Arranging student records by marks or names
- Sorting products by price, rating, or popularity in e-commerce apps
- Organizing files and folders by name, size, or date modified
- Displaying leaderboard rankings in games

Example:
Imagine a baseball team is lined up on the field, as shown in Figure 3.1. You want to arrange the players in order of increasing height (with the shortest player on the left) for the team picture. How would you go about this sorting process?

FIGURE 3.1 The unordered baseball team.

FIGURE 3.2 The ordered baseball team.

Bubble Sort

A simple comparison-based sorting algorithm

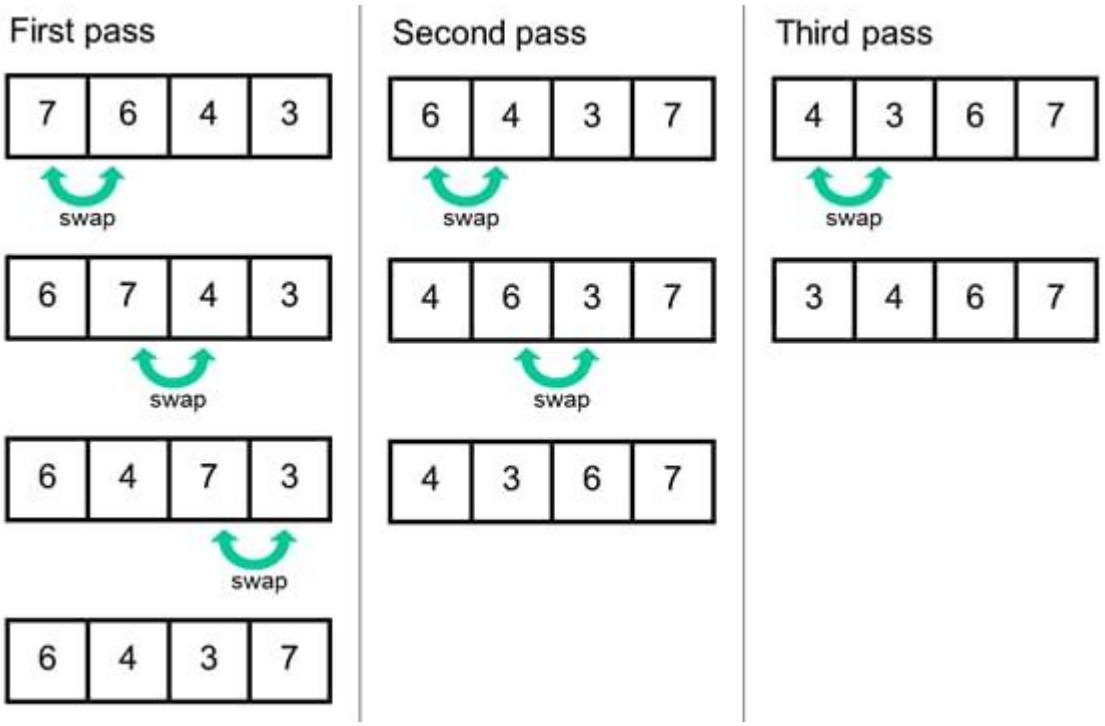
It compares **adjacent elements**, and swaps them if they are in the wrong order.

The process is repeated until the list is sorted

How It Works:

- Start at the beginning of the array.
- Compare each pair of adjacent elements.
- Swap them if they are in the wrong order.
- After each pass, the largest element is placed at the end.
- Repeat until the array is sorted

Example: Here is an array `[7, 6, 4, 3]` being sorted using **Bubble Sort**.



Best Case (Already Sorted): **O(n)**

Worst Case (Reversed Order): **O(n²)**

```
Algorithm: Sequential-Bubble-Sort (A)
fori ← 1 to length [A] do
for j ← length [A] down-to i +1 do
   if A[A] < A[j-1] then
      Exchange A[j] ↔ A[j-1]
```

Java Code for Bubble Sort

```java
public class BubbleSort {
    public static void main(String args[]) {
        int n = 5;
        int[] arr = {67, 44, 82, 17, 20}; //initialize an array
        System.out.print("Array before Sorting: ");
        for(int i = 0; i<n; i++)
            System.out.print(arr[i] + " ");
        System.out.println();
        for(int i = 0; i<n; i++) {
            int swaps = 0; //flag to detect any swap is there or not
            for(int j = 0; j<n-i-1; j++) {
                if(arr[j] > arr[j+1]) { //when the current item is bigger than next
                    int temp;
                    temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
                    swaps = 1; //set swap flag
                }
            }
            if(swaps == 0)
                break;
        }
        System.out.print("Array After Sorting: ");
        for(int i = 0; i<n; i++)
            System.out.print(arr[i] + " ");
        System.out.println();
    }
}
```

Question:

You are given the following array:
`[29, 10, 14, 37, 14]`

1. Sort the array using **Bubble Sort**.
2. How many **swaps** are made in total?

Selection Sort

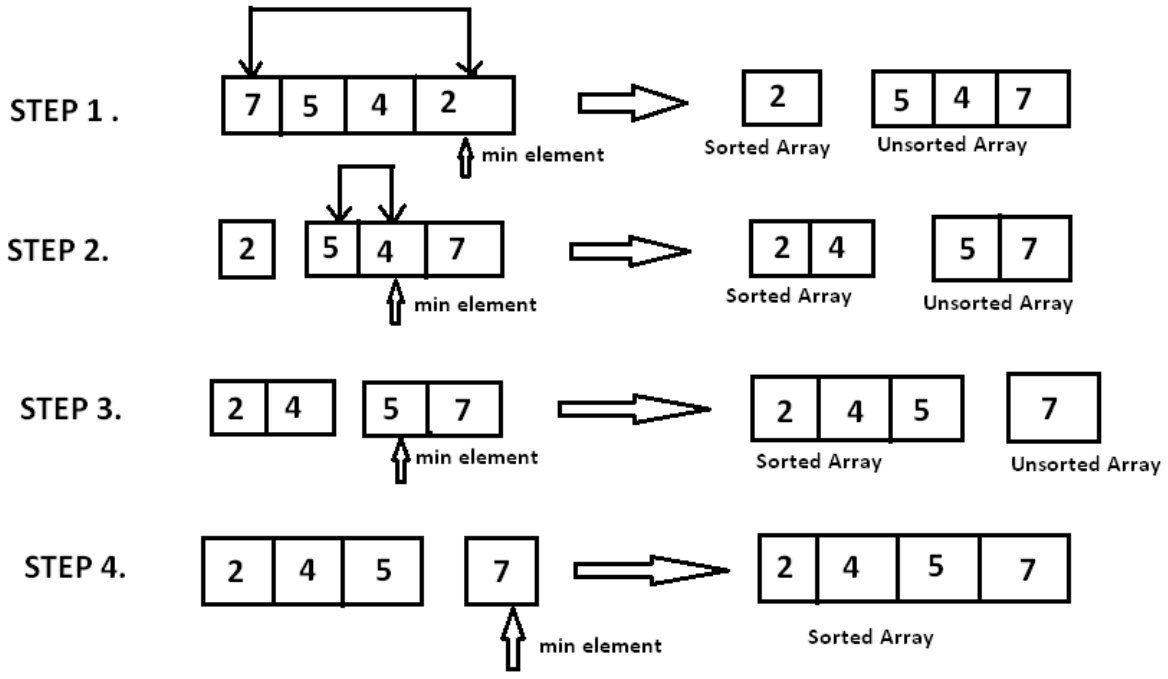
Selection Sort works by **repeatedly selecting the smallest (or largest)** element from the unsorted portion and moving it to the correct position.

It divides the list into a **sorted and an unsorted part**, growing the sorted portion one element at a time.

How It Works:

- Start with the first element as the minimum. Scan the unsorted part to find the actual minimum.
- Swap it with the first unsorted element.
- Move the boundary of the sorted portion forward by one.
- Repeat until the entire array is sorted.

Example: Here is an array `[7, 5, 4, 2]` being sorted using **Selection Sort**.



Best Case: **O(n²)**

Worst Case: **O(n²)**

```
Algorithm: Selection-Sort (A)
fori← 1 to n-1 do
   min j ←i;
   min x ← A[i]
   for j ←i + 1 to n do
      if A[j] < min x then
         min j ← j
         min x ← A[j]
   A[min j] ← A [i]
   A[i] ← min x
```

Java Code for Selection Sort

```java
import java.io.*;
public class SelectionSort {
    public static void main(String args[]) {
        int n = 5;
        int[] arr = {12, 19, 55, 2, 16}; //initialize an array
        System.out.print("Array before Sorting: ");
        for(int i = 0; i<n; i++)
            System.out.print(arr[i] + " ");
        System.out.println();
        int imin;
        for(int i = 0; i<n-1; i++) {
            imin = i; //get index of minimum data
            for(int j = i+1; j<n; j++)
                if(arr[j] < arr[imin])
                    imin = j;

            //placing in correct position
            int temp;
            temp = arr[i];
            arr[i] = arr[imin];
            arr[imin] = temp;
        }
        System.out.print("Array After Sorting: ");
        for(int i = 0; i<n; i++)
            System.out.print(arr[i] + " ");
        System.out.println();
    }
}
```

Question:

Sort the array `[29, 20, 73, 34, 64]` using **Selection Sort**.

1. Show the array after each **selection and swap**.
2. How many **total comparisons and swaps** were made?

Insertion Sort

Insertion Sort works by building a sorted sequence one element at a time.

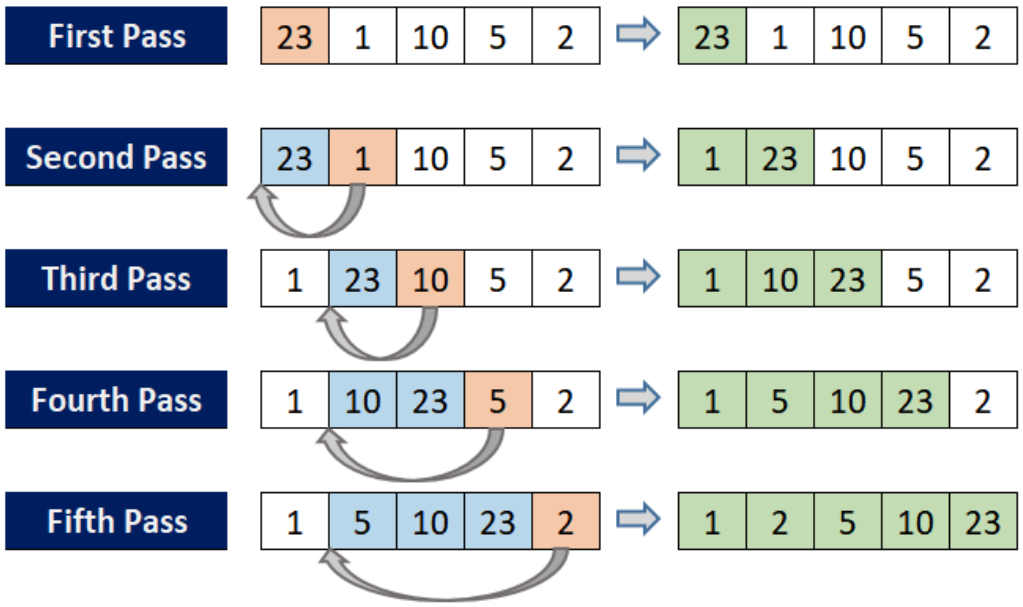
It starts with the **second element**, comparing it with the previous elements, and inserting it into its correct position.

This process repeats until the entire list is sorted.

How It Works:

- Consider the first element as sorted.
- Pick the next element and compare it with the sorted portion.
- Shift larger elements to the right to make space.
- Insert the element in its correct position.
- Repeat until all elements are sorted.

Example: Here is an array `[23, 1, 10, 5, 2]` being sorted using **Insertion Sort**.



Best Case (Already Sorted): **O(n)**

Worst Case (Reversed Order): **O(n²)**

```
Algorithm: Insertion-Sort(A)
for j = 2 to A.length
    key = A[j]
    i = j  1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i -1
    A[i + 1] = key
```

Java Code for Insertion Sort

```java
import java.io.*;
public class InsertionSort {
    public static void main(String args[]) {
        int n = 5;
        int[] arr = {67, 44, 82, 17, 20}; //initialize an array
        System.out.print("Array before Sorting: ");
        for(int i = 0; i<n; i++)
            System.out.print(arr[i] + " ");
        System.out.println();
        for(int i = 1; i<n; i++) {
            int key = arr[i];//take value
            int j = i;
            while(j > 0 && arr[j-1]>key) {
                arr[j] = arr[j-1];
                j--;
            }
            arr[j] = key; //insert in right place
        }
        System.out.print("Array After Sorting: ");
        for(int i = 0; i<n; i++)
            System.out.print(arr[i] + " ");
        System.out.println();
    }
}
```

Question:

Sort the array `[22, 11, 99, 88, 9, 7, 42]` using **Insertion Sort**.

1. Show the array after each **insertion step**.
2. How many **shifts** were made in total?

Merge Sort

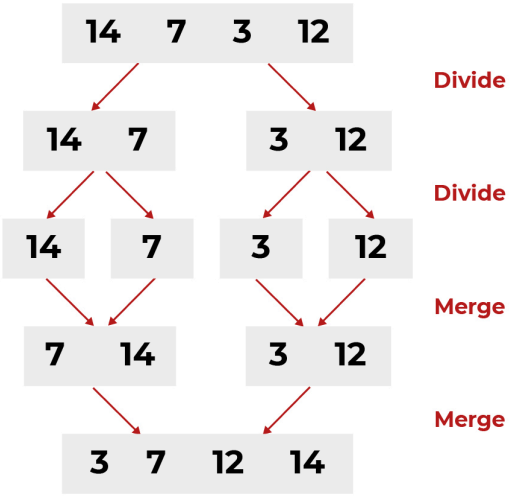
Merge Sort is a powerful **divide-and-conquer** sorting algorithm.

It works by **recursively dividing** the array into two halves.

How It Works:

- Divide the array into two halves.
- Recursively apply merge sort to each half.
- Once the halves are sorted, **merge them** by comparing and arranging elements.
- Repeat this process until the entire array is merged and sorted.

Example: Here is an array `[14, 7, 3, 12]` being sorted using **Merge Sort**.



Best Case: **O(n log n)**

Worst Case: **O(n log n)**

```
procedure mergesort( var a as array )
   if ( n == 1 ) return a
      var l1 as array = a[0] ... a[n/2]
      var l2 as array = a[n/2+1] ... a[n]
      l1 = mergesort( l1 )
      l2 = mergesort( l2 )
      return merge( l1, l2 )
end procedure
procedure merge( var a as array, var b as array )
   var c as array
   while ( a and b have elements )
      if ( a[0] > b[0] )
         add b[0] to the end of c
         remove b[0] from b
      else
         add a[0] to the end of c
         remove a[0] from a
      end if
   end while
   while ( a has elements )
      add a[0] to the end of c
      remove a[0] from a
   end while
   while ( b has elements )
      add b[0] to the end of c
      remove b[0] from b
   end while
   return c
end procedure
```

Java Code for Merge Sort

```
public class MergeSort {
    static int a[] = { 10, 14, 19, 26, 27, 31, 33, 35, 42, 44, 0 };
    static int b[] = new int[a.length];
    static void merging(int low, int mid, int high) {
        int l1, l2, i;
        for(l1 = low, l2 = mid + 1, i = low; l1 <= mid && l2 <= high; i++) {
            if(a[l1] <= a[l2])
                b[i] = a[l1++];
            else
                b[i] = a[l2++];
        }
        while(l1 <= mid)
            b[i++] = a[l1++];
        while(l2 <= high)
            b[i++] = a[l2++];
        for(i = low; i <= high; i++)
            a[i] = b[i];
    }
    static void sort(int low, int high) {
        int mid;
        if(low < high) {
            mid = (low + high) / 2;
            sort(low, mid);
            sort( low: mid+1, high);
            merging(low, mid, high);
        } else {
            return;
        }
    }
```

```
    public static void main(String args[]) {
        int i;
        int n = a.length;
        System.out.println("Array before sorting");
        for(i = 0; i < n; i++)
            System.out.print(a[i] + " ");
        sort( low: 0,  high: n-1);
        System.out.println("\nArray after sorting");
        for(i = 0; i < n; i++)
            System.out.print(a[i]+" ");
    }
}
```

Question:
Sort the array `[38, 27, 43, 3, 9, 82, 10]` using **Merge Sort**.

1. Show the array as it is **divided** step by step.
2. Show the **merge steps** and how sorted subarrays are formed.

Quick Sort

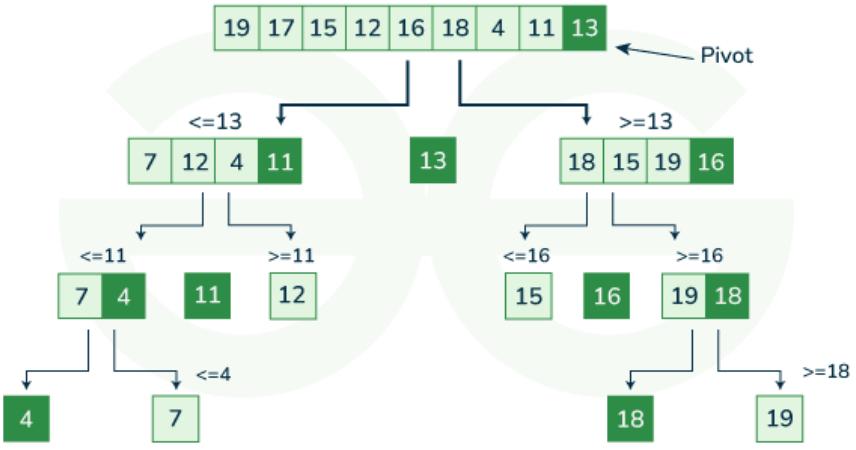
Quick Sort is a highly efficient **divide-and-conquer** algorithm.

It works by selecting a **pivot** element.

How It Works:

- Choose a **pivot** element
- **Partition** the array: place smaller elements on the left, larger on the right.
- Recursively apply quick sort to the left and right subarrays.
- Combine the results – the array becomes fully sorted.

Example: Here is an array `[19, 17, 15, 12, 16, 18, 4, 11, 13]` being sorted using **Quick Sort**.



Best Case: **O(n log n)**

Worst Case: **O(n²)**

```
function partitionFunc(left, right, pivot)
   leftPointer = left
   rightPointer = right - 1

   while True do
      while A[++leftPointer] < pivot do
      //do-nothing
      end while

      while rightPointer > 0 && A[--rightPointer] > pivot do
         //do-nothing
      end while

      if leftPointer >= rightPointer
         break
      else
         swap leftPointer,rightPointer
      end if
   end while

   swap leftPointer,right
   return leftPointer
end function
```

Java Code for Quick Sort

```
import java.util.Arrays;
public class QuickSort {
    int[] intArray = {4,6,3,2,1,9,7};

    void swap(int num1, int num2) {
        int temp = intArray[num1];
        intArray[num1] = intArray[num2];
        intArray[num2] = temp;
    }
    int partition(int left, int right, int pivot) {
        int leftPointer = left - 1;
        int rightPointer = right;

        while (true) {
            while (intArray[++leftPointer] < pivot) {
                // do nothing
            }
            while (rightPointer > 0 && intArray[--rightPointer] > pivot) {
                // do nothing
            }
```

```
            if (leftPointer >= rightPointer) {
                break;
            } else {
                swap(leftPointer, rightPointer);
            }
        }
        swap(leftPointer, right);

        // System.out.println("Updated Array: ");
        return leftPointer;
    }
    void quickSort(int left, int right) {
        if (right - left <= 0) {
            return;
        } else {
            int pivot = intArray[right];
            int partitionPoint = partition(left, right, pivot);
            quickSort(left, right: partitionPoint - 1);
            quickSort( left: partitionPoint + 1, right);
        }
    }
```

```
    public static void main(String[] args) {
        QuickSort sort = new QuickSort();
        int max = sort.intArray.length;
        System.out.println("Contents of the array :");
        System.out.println(Arrays.toString(sort.intArray));

        sort.quickSort( left: 0, right: max - 1);
        System.out.println("Contents of the array after sorting :");
        System.out.println(Arrays.toString(sort.intArray));
    }
}
```

Question:

Sort the array `[45, 23, 78, 10, 89, 34]` using **Quick Sort**.

1. Use the **last element as pivot**.
2. Show the **partitioning steps** and pivot placement after each round.

Shell Sort

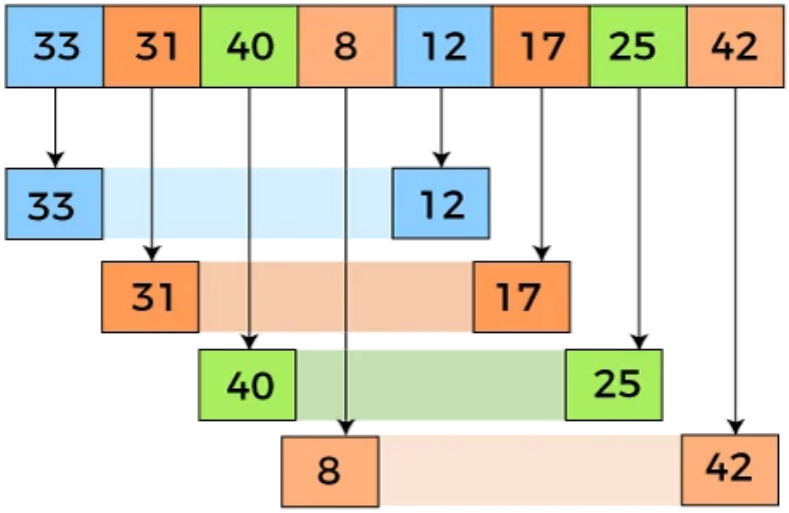
Shell Sort is an optimized version of Insertion Sort

It works by sorting elements that are a certain **gap** distance apart and then gradually reducing the gap.

How It Works:

- Start with a large **gap** between elements (e.g., n/2).
- Sort elements at that gap distance using Insertion Sort.
- Reduce the gap and repeat the process.
- Continue until the gap is 1 — final pass is like regular Insertion Sort.

Example: Here is an array `[33, 31, 40, 8, 12, 17, 25, 42]` being sorted using **Shell Sort**.



Best Case: **O(n log n)** (depends on gap sequence)

Worst Case: **O(n²)** (for some gap sequences)

```
procedure shellSort()
   A : array of items

   /* calculate interval*/
   while interval < A.length /3 do:
      interval = interval * 3 + 1
   end while

   while interval > 0 do:
      for outer = interval; outer < A.length; outer ++ do:

         /* select value to be inserted */
         valueToInsert = A[outer]
         inner = outer;

            /*shift element towards right*/
            while inner > interval -1 &&  A[inner - interval]
            >= valueToInsert do:
               A[inner] = A[inner - interval]
               inner = inner  interval
            end while

         /* insert the number at hole position */
         A[inner] = valueToInsert
         end for

   /* calculate interval*/
   interval = (interval -1) /3;
   end while
end procedure
```

Java Code for Shell Sort

```
import java.io.*;
import java.util.*;
public class ShellSort {
    public static void main(String args[]) {
        int n = 5;
        int[] arr = {33, 45, 62, 12, 98}; //initialize an array
        System.out.print("Array before Sorting: ");
        for(int i = 0; i<n; i++)
            System.out.print(arr[i] + " ");
        System.out.println();
        int gap;
        for(gap = n/2; gap > 0; gap = gap / 2) { //initially gap = n/2, decreasing by gap /2
            for(int j = gap; j<n; j++) {
                for(int k = j-gap; k>=0; k -= gap) {
                    if(arr[k+gap] >= arr[k])
                        break;
                    else {
                        int temp;
                        temp = arr[k+gap];
                        arr[k+gap] = arr[k];
                        arr[k] = temp;
                    }
                }
            }
        }
        System.out.print("Array After Sorting: ");
        for(int i = 0; i<n; i++)
            System.out.print(arr[i] + " ");
        System.out.println();}
}
```

Question:
Sort the array `[64, 34, 25, 12, 22, 11, 90]` using **Shell Sort**.

1. Use **gap sequence: n/2, n/4, ..., 1**.
2. Show the array after each gap-based pass.
3. How is this more efficient than regular Insertion Sort for this input?

Radix Sort

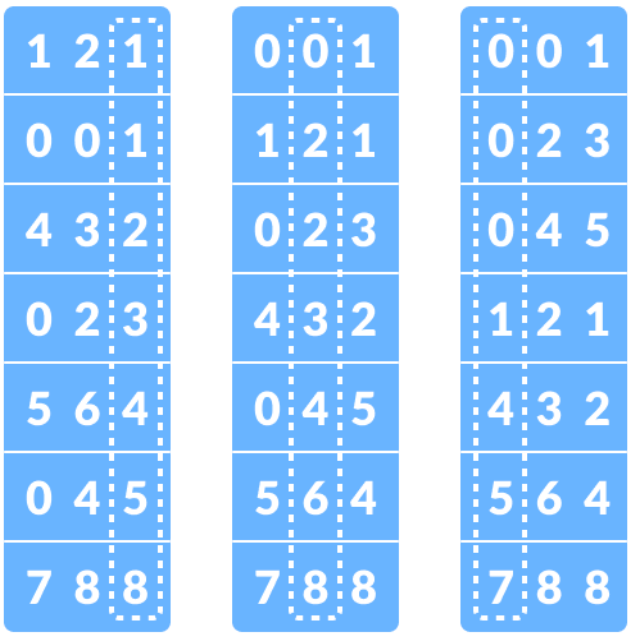
Radix Sort is a **non-comparison-based** sorting algorithm

It works by sorting numbers digit by digit, starting from the **least significant digit (LSD)** to the **most significant digit (MSD)**.

How It Works:

- Find the maximum number to determine the number of digits.
- Sort the numbers **based on each digit**, starting from the least significant.
- Use a **stable sort** (like Counting Sort) at each digit level.
- Repeat the process for each digit place until the entire list is sorted.

Example:



Best Case: **O(nk)**

Worst Case: **O(nk)**
(n = number of elements, k = number of digits in the largest number)

```
Algorithm: RadixSort(a[], n):

   // Find the maximum element of the list
   max = a[0]
   for (i=1 to n-1):
      if (a[i]>max):
         max=a[i]

   // applying counting sort for each digit
   //the input list
   For (pos=1 to max/pos>0):
      countSort(a, n, pos)
      pos=pos*10
```

Java Code for Radix Sort

```
import java.io.*;
public class Main {
    static void countsort(int a[], int n, int pos) {
        int output[] = new int[n + 1];
        int max = (a[0] / pos) % 10;
        for (int i = 1; i < n; i++) {
            if (((a[i] / pos) % 10) > max)
                max = a[i];
        }
        int count[] = new int[max + 1];
        for (int i = 0; i < max; ++i)
            count[i] = 0;
        for (int i = 0; i < n; i++)
            count[(a[i] / pos) % 10]++;
        for (int i = 1; i < 10; i++)
            count[i] += count[i - 1];
        for (int i = n - 1; i >= 0; i--) {
            output[count[(a[i] / pos) % 10] - 1] = a[i];
            count[(a[i] / pos) % 10]--;
        }
        for (int i = 0; i < n; i++)
            a[i] = output[i];
    }
```

```java
    static void radixsort(int a[], int n) {
        int max = a[0];
        for (int i = 1; i < n; i++)
            if (a[i] > max)
                max = a[i];
        for (int pos = 1; max / pos > 0; pos *= 10)
            countsort(a, n, pos);
    }
    public static void main(String args[]) {
        int a[] = {236, 15, 333, 27, 9, 108, 76, 498};
        int n = a.length;
        System.out.println("Before sorting array elements are: ");
        for (int i = 0; i < n; ++i)
            System.out.print(a[i] + " ");
        radixsort(a, n);
        System.out.println("\nAfter sorting array elements are: ");
        for (int i = 0; i < n; ++i)
            System.out.print(a[i]+ " ");
    }
}
```

Question:
Sort the array `[170, 45, 75, 90, 802, 24, 2, 66]` using **Radix Sort**.

1. Show how the array is sorted **digit by digit** (LSD first).
2. What stable sort would you use for each digit position?
3. How many **passes** are required for this array?

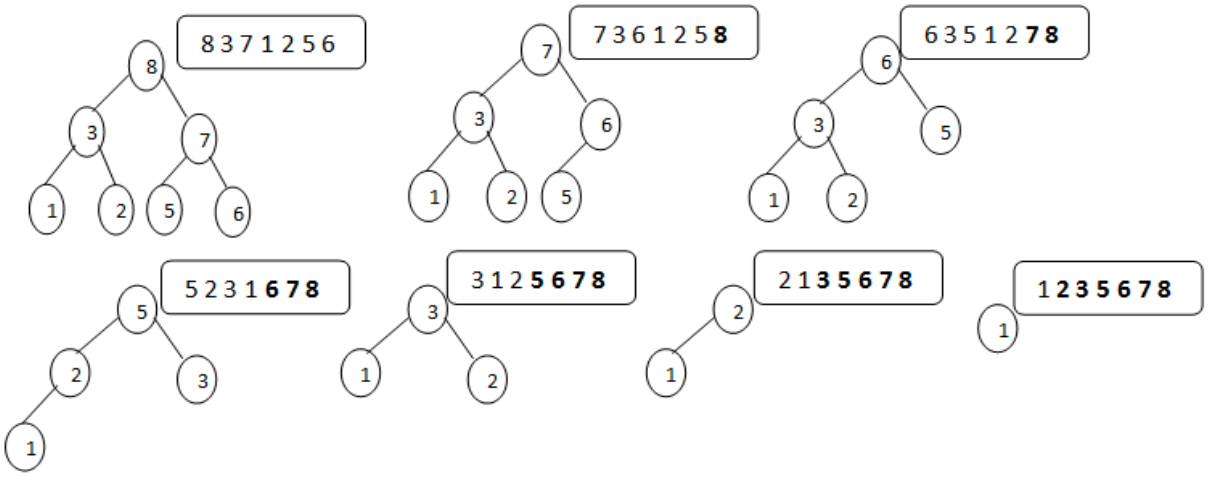
Heap Sort

Heap Sort is a **comparison-based** sorting algorithm that uses a **binary heap** data structure. It first builds a **max-heap** from the input data and then repeatedly removes the largest element (root of the heap), placing it at the end of the array.

How It Works:

- Convert the input array into a **max-heap**.
- Swap the **first (largest)** element with the last.
  Reduce the heap size and **heapify** the root.
- Repeat until all elements are sorted.

Example: Here is an array `[8, 3, 7, 1, 2, 5, 6]` being sorted using **Heap Sort**.



Best Case: **O(n log n)**

Worst Case: **O(n log n)**

Java Code for Heap Sort

```java
import java.io.*;
public class HeapSort {
    static void build_maxheap(int heap[], int n) {
        for (int i = 1; i < n; i++) {
            int c = i;
            do {
                int r = (c - 1) / 2;
                if (heap[r] < heap[c]) { // to create MAX heap array
                    int t = heap[r];
                    heap[r] = heap[c];
                    heap[c] = t;
                }
                c = r;
            } while (c != 0);
        }
        System.out.println("Heap array: ");
        for (int i = 0; i < n; i++) {
            System.out.print(heap[i] + " ");
        }
        heapify(heap, n);
    }
```

```java
static void heapify(int heap[], int n) {
    for (int j = n - 1; j >= 0; j--) {
        int c;
        int temp = heap[0];
        heap[0] = heap[j]; // swap max element with rightmost leaf element
        heap[j] = temp;
        int root = 0;
        do {
            c = 2 * root + 1; // left node of root element
            if ((heap[c] < heap[c + 1]) && c < j-1)
                c++;
            if (heap[root]<heap[c] && c<j) { // again rearrange to max heap array
                temp = heap[root];
                heap[root] = heap[c];
                heap[c] = temp;
            }
            root = c;
        } while (c < j);
    }
    System.out.println("\nThe sorted array is: ");
    for (int i = 0; i < n; i++) {
        System.out.print(heap[i] + " ");
    }
}
```

```java
    public static void main(String args[]) {
        int heap[] = new int[10];
        heap[0] = 4;
        heap[1] = 3;
        heap[2] = 1;
        heap[3] = 0;
        heap[4] = 2;
        int n = 5;
        build_maxheap(heap,n);
    }
}
```

Question:

Sort the array `[12, 11, 13, 5, 6, 7]` using **Heap Sort**.

1. Show how the array is converted into a **max heap**.
2. Show the array after each **extraction and heapify** step.
3. What is the **heap property**, and how does it help sorting?

Complexity Comparison Table

Algorithm	Best Case	Average Case	Worst Case
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$
Merge Sort	O(n log n)	O(n log n)	O(n log n)
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$
Shell Sort	O(n log n)	$O(n (\log n)^2)$	$O(n^2)$
Radix Sort	O(nk)	O(nk)	O(nk)
Heap Sort	O(n log n)	O(n log n)	O(n log n)