Sorting algorithms:

I) **Selection Sort** is a comparison-based sorting algorithm. It sorts an array by repeatedly selecting the **smallest (or largest)** element from the unsorted portion and swapping it with the first unsorted element. This process continues until the entire array is sorted.

1. First we find the smallest element and swap it with the first element. This way we get the smallest element at its correct position.
2. Then we find the smallest among remaining elements (or second smallest) and move it to its correct position by swapping.
3. We keep doing this until we get all elements moved to correct position.

```
void selectionSort(vector<int> &arr) {

    int n = arr.size();

    for (int i = 0; i < n - 1; ++i) {

        // Assume the current position hold the minimum element

        int min_idx = i;

        // Iterate through the unsorted portion to find the actual minimum

        for (int j = i + 1; j < n; ++j) {

            if (arr[j] < arr[min_idx]) {

                // Update min_idx if a smaller element is found

                min_idx = j;

            }

        }

        // Move minimum element to its  correct position

        swap(arr[i], arr[min_idx]);

    }

}
```

Time Complexity = o(n^2)
Space Complexity = o(1)

ii) **Bubble Sort** is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity are quite high.

- We sort the array using multiple passes. After the first pass, the maximum element goes to end (its correct position). Same way, after second pass, the second largest element goes to second last position and so on.
- In every pass, we process only those elements that have already not moved to correct position. After k passes, the largest k elements must have been moved to the last k positions.
- In a pass, we consider remaining elements and compare all adjacent and swap if larger element is before a smaller element. If we keep doing this, we get the largest (among the remaining elements) at its correct position.

```
void bubbleSort(vector<int>& arr) {
    int n = arr.size();
    bool swapped;

    for (int i = 0; i < n - 1; i++) {
        swapped = false;
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }

        // If no two elements were swapped, then break
        if (!swapped)
            break;
    }
}
```

Time Complexity = o(n^2)
Space Complexity = o(1)

iii) **Insertion sort** is a simple sorting algorithm that works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list. It is like sorting playing cards in your hands. You split the cards into two groups: the sorted cards and the unsorted cards. Then, you pick a card from the unsorted group and put it in the right place in the sorted group.

- We start with second element of the array as first element in the array is assumed to be sorted.
- Compare second element with the first element and check if the second element is smaller then swap them.
- Move to the third element and compare it with the first two elements and put at its correct position
- Repeat until the entire array is sorted.

```
void insertionSort(int arr[], int n)
{
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int 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;
    }
}
```

Time Complexity = o(n^2)
Space Complexity = o(1)

iv) **Merge sort** is a popular sorting algorithm known for its efficiency and stability. It follows the **divide-and-conquer** approach to sort a given array of elements.
Here’s a step-by-step explanation of how merge sort works:

1. **Divide:** Divide the list or array recursively into two halves until it can no more be divided.
2. **Conquer:** Each subarray is sorted individually using the merge sort algorithm.
3. **Merge:** The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.

```
void merge(vector<int>& arr, int left,

                int mid, int right)

{

    int n1 = mid - left + 1;

    int n2 = right - mid;

    // Create temp vectors

    vector<int> L(n1), R(n2);

    // Copy data to temp vectors L[] and R[]

    for (int i = 0; i < n1; i++)

        L[i] = arr[left + i];

    for (int j = 0; j < n2; j++)

        R[j] = arr[mid + 1 + j];

    int i = 0, j = 0;

    int k = left;

    // Merge the temp vectors back into arr[left..right]

    while (i < n1 && j < n2) {

        if (L[i] <= R[j]) {

            arr[k] = L[i];

            i++;

        }

        else {

            arr[k] = R[j];

            j++;

        }

        k++;

    }
    // Copy the remaining elements of L[],

    // if there are any

    while (i < n1) {

        arr[k] = L[i];

        i++;

        k++;

    }

    // Copy the remaining elements of R[], if there are any

    while (j < n2) {

        arr[k] = R[j];

        j++;

        k++;

    }

}

// begin is for left index and end is right index of the sub-array of arr to be sorted

void mergeSort(vector<int>& arr, int left, int right)

{

    if (left >= right)

        return;

    int mid = left + (right - left) / 2;

    mergeSort(arr, left, mid);

    mergeSort(arr, mid + 1, right);

    merge(arr, left, mid, right);

}
```

Time Complexity = O(n log n)
Space Complexity = O(n)

v) **Quick Sort** works on the principle of **divide and conquer**, breaking down the problem into smaller sub-problems.
There are mainly three steps in the algorithm:

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.

```
int partition(vector<int>& arr, int low, int high) {

     // Choose the pivot

   int pivot = arr[high];

     // Index of smaller element and indicates the right position of pivot found so far

   int i = low - 1;

   // Traverse arr[;ow..high] and move all smallest elements on left side. Elements from low to

   // i are smaller after every iteration

   for (int j = low; j <= high - 1; j++) {

      if (arr[j] < pivot) {

         i++;

         swap(arr[i], arr[j]);

      }

   }

    // Move pivot after smaller elements and return its position

   swap(arr[i + 1], arr[high]);

   return i + 1;

}

// The QuickSort function implementation

void quickSort(vector<int>& arr, int low, int high) {

     if (low < high) {

           // pi is the partition return index of pivot

      int pi = partition(arr, low, high);
```

```
        // Recursion calls for smaller elements

        // and greater or equals elements

        quickSort(arr, low, pi - 1);

        quickSort(arr, pi + 1, high);

    }

}
```

Time Complexity = O(n log n) for average case in worst-case = O(n^2)
Space Complexity = O(n)
Not Stable

vi) **Counting Sort** is a **non-comparison-based** sorting algorithm. It is particularly efficient when the range of input values is small compared to the number of elements to be sorted. The basic idea behind **Counting Sort** is to count the **frequency** of each distinct element in the input array and use that information to place the elements in their correct sorted positions.

Counting Sort Algorithm:

- Declare an auxiliary array **countArray[]** of size **max(inputArray[])+1** and initialize it with **0**.
- Traverse array **inputArray[]** and map each element of **inputArray[]** as an index of **countArray[]** array, i.e., execute **countArray[inputArray[i]]++** for **0 <= i < N**.
- Calculate the prefix sum at every index of array **inputArray**[].
- Create an array **outputArray[]** of size **N**.
- Traverse array **inputArray[]** from end and update **outputArray[countArray[inputArray[i]] – 1] = inputArray[i]**. Also, update **countArray[inputArray[i]] = countArray[inputArray[i]]- –** .

```
vector<int> countSort(vector<int>& inputArray)
{

   int N = inputArray.size();
   // Finding the maximum element of array inputArray[].
   int M = 0;
   for (int i = 0; i < N; i++)
      M = max(M, inputArray[i]);
   // Initializing countArray[] with 0
   vector<int> countArray(M + 1, 0);
   // Mapping each element of inputArray[] as an index of countArray[] array
   for (int i = 0; i < N; i++) countArray[inputArray[i]]++;
   // Calculating prefix sum at every index of array countArray[]
   for (int i = 1; i <= M; i++) countArray[i] += countArray[i - 1];
   // Creating outputArray[] from countArray[] array
   vector<int> outputArray(N);
   for (int i = N - 1; i >= 0; i--)
   {
      outputArray[countArray[inputArray[i]] - 1]= inputArray[i];
      countArray[inputArray[i]]--;
   }
   return outputArray;
}
```

Time complexity = O(N+M)
Space complexity = O(N+M) where N = output array size and M = count array size

vii) **Radix Sort** is a linear sorting algorithm that sorts elements by processing them digit by digit. It is an efficient sorting algorithm for integers or strings with fixed-size keys.
Rather than comparing elements directly, Radix Sort distributes the elements into buckets based on each digit’s value. By repeatedly sorting the elements by their significant digits, from the least significant to the most significant, Radix Sort achieves the final sorted order.

Radix Sort Algorithm

The key idea behind Radix Sort is to exploit the concept of place value. It assumes that sorting numbers digit by digit will eventually result in a fully sorted list. Radix Sort can be performed using different variations, such as Least Significant Digit (LSD) Radix Sort or Most Significant Digit (MSD) Radix Sort.

```
int getMax(int arr[], int n)
{
    int mx = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > mx)
            mx = arr[i];
    return mx;
}

// A function to do counting sort of arr[] according to the digit represented by exp.
void countSort(int arr[], int n, int exp)
{
    // Output array
    int output[n];
    int i, count[10] = { 0 };
    // Store count of occurrences in count[]
    for (i = 0; i < n; i++)
        count[(arr[i] / exp) % 10]++;
    // Change count[i] so that count[i] now contains actual position of this digit in output[]
    for (i = 1; i < 10; i++)
        count[i] += count[i - 1];
    // Build the output array
    for (i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }
// Copy the output array to arr[], so that arr[] now contains sorted numbers according to current digit
    for (i = 0; i < n; i++)
        arr[i] = output[i];
}

// The main function to that sorts arr[ of size n using Radix Sort
void radixsort(int arr[], int n)
{
    // Find the maximum number to know number of digits
    int m = getMax(arr, n);
    // Do counting sort for every digit. Note that instead of passing digit
    // number, exp is passed. exp is 10^i
    // where i is current digit number
    for (int exp = 1; m / exp > 0; exp *= 10)
        countSort(arr, n, exp);
}
```

Time Complexity = O((n+b)*d)
Space Complexity = O(n+b)   where n is the number of elements and b = base of the exponent used and d = digit of the number

viii)