Bubble Sort:

Algorithm:

1. Traverse the array from the first element to the last element.
2. Compare adjacent elements and swap them if they are in the wrong order.
3. Repeat step 1 and 2 until the array is sorted.

Code:

```
#include <iostream>

using namespace std;

void bubble(int arr[], int n)

{

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

    {

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

        {

            if (arr[j] > arr[j + 1])

            {

                swap(arr[j], arr[j + 1]);

            }

        }

    }

}
```

```
int main()

{

    int n;

    cout << "Enter the size of the array\n";

    cin >> n;

    int arr[n];

    cout << "enter the elements\n";

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

    {

        cin >> arr[i];

    }

    bubble(arr, n);

    cout << "The sorted array is\n";

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

    {

        cout << arr[i] << " ";

    }

    return 0;

}
```

Insertion Sort

Algorithm:Traverse the array from the second element to the last element.

Compare the current element with the sorted subarray on the left.

Shift the elements in the sorted subarray that are greater than the current element to the right.

Insert the current element in its correct position in the sorted subarray.

Repeat step 1 to 4 until the array is sorted.

Code:

```
#include<iostream>

void insertsort(int arr[],int n);

using namespace std;

int main(){

int n;

cout<<"Enter the size of the array\n";

cin>>n;

int arr[n];

cout<<"Enter the values:\n";

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

    cin>>arr[i];

}

insertsort(arr,n);
cout<<"The sorted array is: ";

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

{

    cout<<arr[i]<<" ";

}

return 0;

}

void insertsort(int arr[],int n){

    int temp;

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

    {

        int temp=arr[i];

        int j=i-1;

        while(j>=0 && arr[j]>temp)

        {

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

            j--;

        }

        arr[j+1]=temp;

    }

}
```

Merge Sort:

Algorithm:

1. Divide the array into two halves.
2. Recursively sort the two halves.
3. Merge the sorted halves.

```
#include <iostream>

void mergesort(int arr[], int left, int right);

void merge(int arr[], int lb, int mid, int ub);

using namespace std;

int main()

{

    int n;

    cout << "Enter the size of the array\n";

    cin >> n;

    int arr[n];

    cout << "Enter the values:\n";

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

    {

        cin >> arr[i];

    }

    mergesort(arr, 0, n - 1);
    cout << "The sorted array is: ";

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

    {

        cout << arr[i] << " ";

    }

    return 0;

}

void mergesort(int arr[], int left, int right)

{

    if (left < right)

    {

        int lb = left;

        int ub = right;

        int mid = (lb + ub) / 2;

        mergesort(arr, lb, mid);

        mergesort(arr, mid + 1, ub);

        merge(arr, lb, mid, ub);

    }

}

void merge(int arr[], int lb, int mid, int ub)
```

```
{

    int i = lb;

    int j = mid + 1;

    int k = lb;

    int arrb[ub - lb + 1];

    while (i <= mid && j <= ub)

    {

        if (arr[i] <= arr[j])

        {

            arrb[k] = arr[i];

            i++;

        }

        else

        {

            arrb[k] = arr[j];

            j++;

        }

        k++;

    }

    if (i > mid)
```

```
{

    while (j <= ub)

    {

        arrb[k] = arr[j];

        k++;

        j++;

    }

}

else

{

    while (i <= mid)

    {

        arrb[k] = arr[i];

        k++;

        i++;

    }

}

for (int i = lb; i <= ub; i++)

{

    arr[i] = arrb[i];
    }

}
```

Heap Sort:

Algorithm:

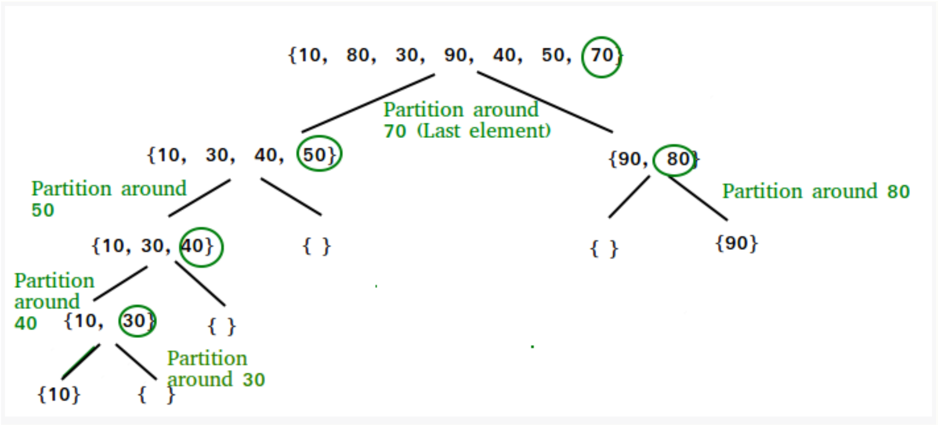
1. Build a max heap from the array.
2. Extract the maximum element from the heap and place it at the end of the array.
3. Heapify the remaining elements in the heap.
4. Repeat step 2 to 3 until the array is sorted.

Code:

```
#include <iostream>

using namespace std;

void heapify(int arr[], int n, int i)

{

    int largest = i;

    int l = i * 2 + 1;

    int r = i * 2 + 2;

    if (l <n && arr[l] > arr[largest])

    {

        largest = l;

    }

    if (r < n && arr[r] > arr[largest])
    {

        largest = r;

    }

    if (largest != i)

    {

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

        heapify(arr, n, largest);

    }

}

void heapsort(int arr[], int n)

{



    for (int i = n / 2-1; i >= 0; i--)

    {

        heapify(arr, n, i);

    }

    for (int i = n-1 ; i >= 0; i--)

    {

        swap(arr[i], arr[0]);

        heapify(arr, i , 0);
    }
}
int main()
{
    int n;
    cout << "Enter the size of array\n";
    cin >> n;
    int arr[n];
    cout << "Enter the elements of the array\n";
    for (int i = 0; i < n; i++)
    {
        cin >> arr[i];
    }
    heapsort(arr, n);
    cout << "The sorted array is\n";
    for (int i = 0; i < n; i++)
    {
        cout << arr[i] << " ";
    }
    return 0; }
```

Quick Sort:

1. Choose a pivot element from the array.
2. Partition the array into two subarrays: one with elements less than the pivot and one with elements greater than or equal to the pivot.
3. Recursively apply quicksort to the two subarrays.
4. Concatenate the sorted subarrays and the pivot element in order to form a new sorted array.



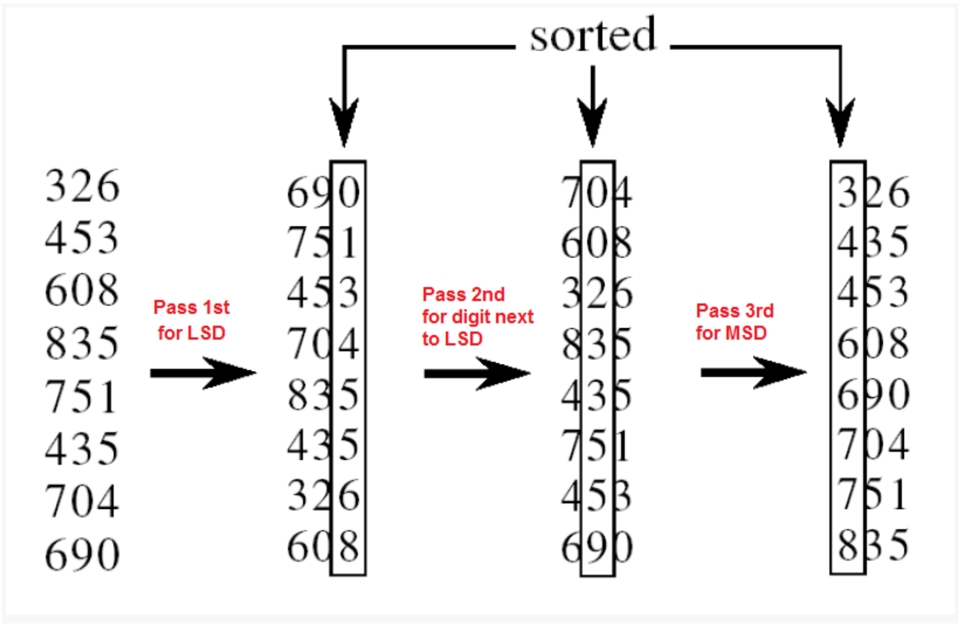
Count Sort:

Algorithm:

1. Count the frequency of each element in the array.
2. Calculate the cumulative sum of the counts.
3. Use the cumulative sum to determine the position of each element in the output array.
4. Copy the elements to the output array in their correct positions.

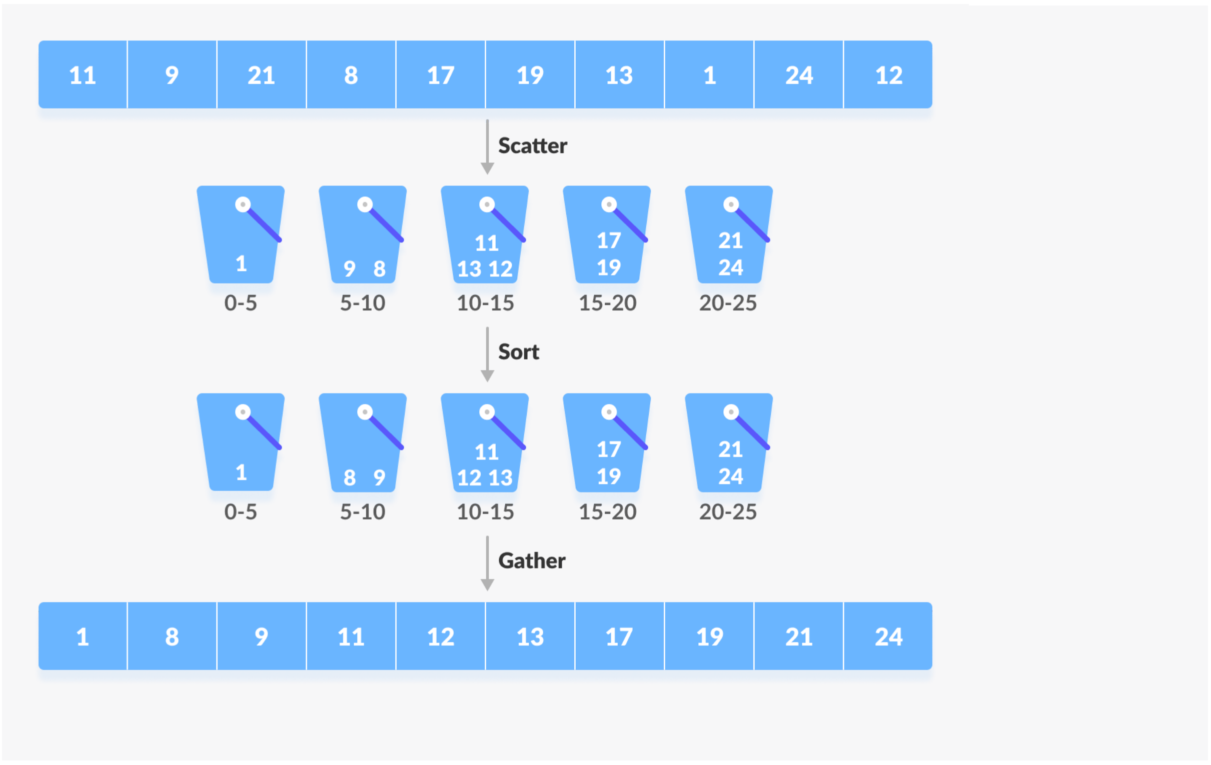
Radix Sort

1. Determine the maximum number of digits in the largest element of the array.
2. For each digit position, create 10 buckets (0-9).
3. Iterate through the array and place each element in the appropriate bucket based on its digit value at the current position.
4. Concatenate the elements in each bucket in order to form a new array.
5. Repeat steps 3-4 for each digit position.



Bucket Sort

6. Create n empty buckets.
7. Iterate through the array and place each element in the appropriate bucket based on its value.
8. Sort each bucket.
9. Concatenate the elements in each bucket in order to form a new array.



Divide and conquer is a problem-solving technique used in computer science and mathematics. It is a recursive algorithmic approach that breaks down a complex problem into smaller, more manageable subproblems, solves those subproblems, and then combines their solutions to solve the original problem. This technique is widely used to solve a variety of problems efficiently and is a fundamental concept in algorithm design.
Here's how divide and conquer is applied in the context of merge sort:

> 1. Divide: The first step in merge sort is to divide the unsorted array into two equal halves, recursively, until you have subarrays with only one or zero elements. Each division step breaks the problem down into smaller subproblems. This is done until we reach the base case, which is a single element or an empty array (both of which are trivially sorted).
>
> 2. Conquer: Once you have the subarrays with one or zero elements, they are considered sorted by definition. This is the "conquer" step, as you have solved the simplest form of the problem. It's essential to note that the conquer step involves doing nothing because a single element or an empty array is already considered sorted.
>
> 3. Combine (Merge): After all the subarrays are sorted, you begin the "combine" step, which involves merging the subarrays to create larger sorted subarrays. This is where the name "merge sort" comes from. The merging process combines two smaller sorted subarrays into a larger sorted subarray. This is done in a way that the elements are compared and arranged in ascending order, ensuring that the final merged array is sorted.
>
> 4. Continue Merging: The merge step is repeated recursively until you've combined all the subarrays back into the original array. Each merge operation combines two smaller, sorted subarrays into a larger, sorted subarray, maintaining the overall sorted order.

The main idea behind merge sort is that by dividing the problem into smaller and smaller pieces, eventually, you reach the base case where the problem is trivially solved. Once these small subproblems are sorted, you can efficiently combine them into larger sorted subarrays. This recursive combination ensures that the original array is sorted correctly.