Algo	Best case	Average case	Worst case	Space complexity
Bubble	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Quick	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$ (recursion)
Heap	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$
Counting	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$
Radix	$O(nk)$	$O(nk)$	$O(nk)$	$O(n+k)$
Bucket	$O(n+k)$	$O(n+k)$	$O(n^2)$	$O(n)$
Shell	$O(n \log n)$	$O(n(\log n)^2)$	$O(n(\log n)^2)$	$O(1)$

Sorting

Bubble sort :->

def :->

It's the simplest sorting algo. It's works on repetedly swapping of adjacent elements until they are not in intended order. It's called as bubble sort.

Dis-advantage :->

It's not suitable for large data sets. the avarge and worst case complexity of bubble sort is - $O(n^2)$. Where n is numbers of elements.

- Best case is - $O(n)$
- Space complexity -> $O(1)$

Uses :->

- Bubble sort use where-
  -) complexity does not matter.
  - And simple and shortcode is prefered.

Algo :->

```
begin bubbleSort(arr)
  for all array element
    if (arr[i] > arr[i+1])
      swap(arr[i], arr[i+1])
    end if
  end for.
  return arr
end bubble sort.
```

implemention :->

implemention in C :->

```c
//bubble sort implemetion


#include<stdio.h>

void print(int a[], int n)
{
    int i;
    for(i = 0; i < n; i++) {
        printf("%d ", a[i]);
    }
    printf("\n");
}

void bubble(int a[], int n)
{
    int i, j, temp;
    for(i = 0; i < n-1; i++) {  // Outer loop iterates over the entire array
        for(j = 0; j < n-i-1; j++) {  // Inner loop compares adjacent elements
            if(a[j] > a[j+1]) {  // If current element is greater than the next
                // Swap the elements
                temp = a[j];
                a[j] = a[j+1];
                a[j+1] = temp;
            }
        }
    }
}

int main()
{
    int a[5] = {10, 25, 32, 13, 20};
    int n = sizeof(a) / sizeof(a[0]);

    printf("Before sorting:\n");
    print(a, n);

    bubble(a, n);

    printf("After sorting:\n");
    print(a, n);

    return 0;
}

```

2) Merge sort :->

Defination :-> Mege sort is a sorting
tech nique that follows divide and conqer
approach. This will be very helpful and
interesting.

Uses :-> Merge sort is similar to quick sort
algorithm as it uses the divide and conquer
approach to sort elements.

Complexity :->

Time :->

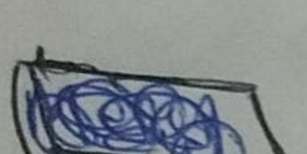
Best case	Avg case	Worst case
O(n log n)	O(n log n)	O(n log n)

all are same.

space :-> O(n),

Algo :->
arr is given array, beg is starting element
and end is last element of array.

MERGE_SORT (arr, beg, end)
if beg < end
set mid = (beg + end)/2.



MERGE_SORT (arr, beg, mid)
MERGE_SORT (arr, mid+1, end)
MERGE (arr, beg, mid, end)
end of if
End MERGE_SORT)

```c
//merge sort implemetion

#include <stdio.h>

void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;  // Size of the left subarray
    int n2 = right - mid;     // Size of the right subarray

    // Create temporary arrays
    int L[n1], R[n2];

    // Copy data to temporary arrays 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];

    // Merge the temporary arrays back into arr[left..right]
    int i = 0;  // Initial index of the first subarray
    int j = 0;  // Initial index of the second subarray
    int k = left;  // Initial index of the merged subarray

    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 any
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Copy the remaining elements of R[], if any
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;  // Avoid overflow for large indices

        // Recursively sort the first and second halves
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        // Merge the sorted halves
        merge(arr, left, mid, right);
    }
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Given array is:\n");
    printArray(arr, n);

    mergeSort(arr, 0, n - 1);

    printf("\nSorted array is:\n");
    printArray(arr, n);

    return 0;
}
```

3) Quick Sort :->

* Defination :-> Quick sort is a sorting technique that uses the divide and conquer approach. It is highly efficient and widely used in practice due to its average-case performance.

* Usage :->

quick sort works by selecting a pivot element, partitioning the array into two parts, and recursively sorting the subarrays.

* complexity :->

- Time :->

Best case	Avg case	Worst case
O(n log n)	O(n log n)	O(n²)

- Space :->
  - Best and average case occur when the pivot divides the array evenly.
  - Worst case occur when the pivot results highly unbalanced partitions.
- Space :-> O(log n) (for recursive calls).

* Algo :->

arr is given array, beg is the starting index, and end is the last index of array.

```
Quick_Sort (arr, beg, end)
    if beg < end
        set pivot = Partition(
arr, beg, end)
Quick_Sort (arr, beg, pivot-1)
Quick_Sort (arr, pivot+1, end)
    End of if
End Quick_Sort.
```

* Partition Function :->

```
Partition (arr, beg, end)
    Set Pivot = arr [end]
    set i = beg - 1
For j = beg to end-1
    if arr [j] <= Pivot
        set i = i+1
        Swap arr[i] and arr[
    End of if
End of for.
Swap arr [i+1] and arr [e
    return i+1
End Partition.
```

```c
//quick sort implemetion

#include <stdio.h>

// Function to partition the array
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // Pivot element (last element)
    int i = low - 1;       // Index of smaller element

    for (int j = low; j < high; j++) {
        // If the current element is smaller than or equal to the pivot
        if (arr[j] <= pivot) {
            i++;
            // Swap arr[i] and arr[j]
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }

    // Swap arr[i + 1] and arr[high] (pivot)
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;

    return (i + 1); // Return the partition index
}

// Function to implement Quick Sort
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // Partition the array and get the pivot index
        int pi = partition(arr, low, high);

        // Recursively sort elements before and after the partition
        quickSort(arr, low, pi - 1);  // Left subarray
        quickSort(arr, pi + 1, high); // Right subarray
    }
}

// Function to print the array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main() {
    int arr[] = {10, 80, 30, 90, 40, 50, 70};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Given array is:\n");
    printArray(arr, n);

    quickSort(arr, 0, n - 1);

    printf("\nSorted array is:\n");
    printArray(arr, n);

    return 0;
}
```