`BUBBLE SORT:` It just compare and swap the adjacent elements in a repetation until the array elements are in the correct order[ascending or descending].

`PSEUDOCODE :`

```
BUBBLE_SORT(arr,n)
for (i = 0; i < n - 1; i++){
        for (j = 0; j < n - i - 1; j++){
            if (arr[j] > arr[j + 1]){
                swap(arr[j], arr[j + 1]);
        }
    }
}
```

`CODE:`

```
    int s,e;
    // s = start
    // e = end
    int temp;
    for(s = 0; s<n-1; s++){ // 5,4,3,2,1 => 5
        for(e = 0; e<(n-1)-s; e++){
            if(arr[e] > arr[e+1]){ // 5 , 4
                //swapping
                temp = arr[e];
                arr[e] = arr[e+1];
                arr[e+1] = temp;
            }
        }
    }
```

`COMPLEXITY:`

- Worst, Average case performance is **O(N^2)**
- Best case is `O(N)`
- **Bubble sort is 70% faster than Selection sort.**

`INSERTION SORT:` Insert an element from unsorted array to it's current position in sorted array.

`PSEUDOCODE :`

```
INSERTION_SORT(arr,n):
    for (i = 1; i < n; i++){
        key = arr[i];
        j = i - 1;
        while (j >= 0 && arr[j] > key){
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
```

CODE:

```
void insertionSort(int arr[], int n){
    int i, key, j;
    for(i = 1; i<n; i++){
        key = arr[i]; // 8
        j = i-1; // j = 0, arr[j] = 2
        while(j>=0 && arr[j] > key){ // arr[0] = 2 > 8
            arr[j+1] = arr[j];
            j = j-1;
        }
        arr[j+1] = key; // arr[0+1] = arr[1] = key = 8 => arr[1] = 8
    }
}
```

COMPLEXITY:

- Worst, Average case performance is **O(N^2)**
- Best case is `O(n)`
- **Insertion sort is 20% faster than Selection sort.**
- **Insertion sort is 500% faster than Bubble sort**, because there are less swaps while sorting elements in array, less writes done, less cache misses (possibly, works more on recently accessed items/ caches) compared to bubble sort.

SELECTION SORT: Find the smallest/minimum element in an unsorted array and swap it with the element at first position.

PSEUDOCODE :

```
SELECTION_SORT(arr,n):
    for (i = 0; i < n - 1; i++){
        min_idx = i;
        for (j = i + 1; j < n; j++){
            if (arr[j] < arr[min_idx]){
                min_idx = j;
            }
        }
        swap(arr[min_idx], arr[i]);
    }
```

`CODE:` *

```
void selectionSort(int arr[], int n){
    int i, j, min_idx;
    for(i = 0; i<n-1; i++){
        min_idx = i;
        for(j = i+1; j<n; j++){
            if(arr[j] < arr[min_idx]){
                min_idx = j;
            }
        }
        swap(arr[min_idx], arr[i]);
    }
}
```

`COMPLEXITY:`

- Worst, Average case performance is **O(N^2)**
- Best case is **O(N^2)**
- **Selection sort is 100% faster than Bubble sort.**
- The greater number of comparisons the slow your program will run.
- So, never use Selection Sort.

`MERGE SORT:` This sorting algorithm is based on the Divide and Conquer Algorithm which simply means it divides the problem in a smaller subdivisions from which we can easily solve that problems and then we merge the divided problems into one.

`PSEUDOCODE :`

```
MERGE_SORT(arr, l, r):
    if (l < r){
        m = l + (r - l) / 2;
        MERGE_SORT(arr, l, m);
        MERGE_SORT(arr, m + 1, r);
        MERGE(arr, l, m, r);
    }

MERGE(arr, l, m, r):
    n1 = m - l + 1;
    n2 = r - m;
    L[n1], R[n2];
    for (i = 0; i < n1; i++){
        L[i] = arr[l + i];
    }
    for (j = 0; j < n2; j++){
        R[j] = arr[m + 1 + j];
    }
    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2){
        if (L[i] <= R[j]){
            arr[k] = L[i];
            i++;
        }
        else{
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1){
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2){
        arr[k] = R[j];
        j++;
        k++;
    }
```

CODE:

```
void merge(int arr[], int l, int mid, int r){
    int i,j,k;
    int n1 = mid-l+1;
    int n2 = r-mid;

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

    /*Copy data to the temporary arrays*/
    for(i=0;i<n1;i++){
        L[i] = arr[l+i];
    }
    for(j=0;j<n2;j++){
        R[j] = arr[mid+1+j];
    }

    // Merge the temp arrays back into arr[1...r]
    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of 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 there are any
    while(i<n1){
        arr[k] = L[i];
        i++;
        j++;
    }

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


void MergeSort(int arr[], int l, int r){
    if(l<r){
        int mid = l+ (r-l)/2;
        // int mid = (l+r)/2;
        MergeSort(arr, l, mid);
        MergeSort(arr, mid+1, r);
        merge(arr, l, mid, r);
    }
}
```

`COMPLEXITY:`

- Worst, Average case performance is **O(NlogN)**
- Best case is **O(NlogN)**
- **Merge sort is 100% faster than Bubble sort.**
- **Merge sort is 100% faster than Selection sort.**
- **Merge sort is 100% faster than Insertion sort.**
- **Merge sort is 100% faster than Quick sort.**
- **Merge sort is 100% faster than Heap sort.**

`How merge function works?`

At last two arrays are left, so merge them we need two pointers first pointer dedicating for 1st array and 2nd pointer to the 2nd array. This will help us to compare the elements of both arrays and merge them in sorted order. We can't directly put the elements of 2nd array in the main array because it will be in unsorted order. So, we need to compare the elements of both arrays and put the smaller element in the main array. We will do this *until we reach the end of any of the array. After that, we will put the remaining elements of the array which is not finished.*

`QUICK SORT:` This sorting algorithm is based on the Divide and Conquer Algorithm which simply means it divides the problem in a smaller subdivisions from which we can easily solve that problems and then we merge the divided problems into one.

In Quick sort we have a special element which is called `PIVOT`. We will take the last element of the array as a pivot. We will compare the elements of the array with the pivot and put the smaller elements on the left side of the pivot and the greater elements on the right side of the pivot. After that, we will do the same thing for the left and right subarrays. We will do this until we reach the end of the array.

`PIVOT` can be any element in the array. For making everything easy-peasy we take the last or the first element.

- But, it is not a good practice to take the first element as a pivot because it can make the worst case performance of the Quick sort. So, we take the last element as the pivot.
- For the **middle element** pivot we need to find the middle element of the array. We can do this by using the formula `mid = l + (r-l)/2`. After that, we will swap the middle element with the last element of the array. After that, we will take the last element as the pivot.
- **Randomised Pivot** is the best way to choose the pivot. We will choose the pivot randomly from the array. We can do this by using the formula `rand() % (r-l+1) + l`. After that, we will swap the pivot with the last element of the array. After that, we will take the last element as the pivot.

Quick Sort in a easy way: robedwardssd.yt

SOME OTHER SORTINGS: [😃]

- **DUTCH NATIONAL FLAG SORTING** *[Used for the problem in which we have to sort the array of 0, 1, 2]*
- **Heap Sort** *[Used for the problem in which we have to sort the array in descending order]*

You can watch any chhapri YT or blog for it.
