**Q1. What is the difference between in-place and out-place sorting algorithms?**

**In Place:-**

1. The input is transformed via an in-place process without the need for additional memory. The input is often replaced with the output as the algorithm runs. Thus no additional space is required for this procedure.
2. An in-place algorithm could need a small quantity of extra RAM to function. However, the amount of RAM used must be fixed and not based on the input size.
3. Many sorting algorithms, including insertion sort, selection sort, rapid sort, bubble sort, heap sort, etc., rearrange the input into sorted order in place. The extra space needed by each technique to rearrange the input array elements is constant.
4. An algorithm is typically classified as either an in-place algorithm or an out-of-place algorithm based on the explicit storage it allots. Since the call stack takes up more space, it is strongly disputed whether or not a recursive algorithm should be considered in place. Although the call stack strictly disqualifies recursive algorithms from the category of in-place algorithms, methods that merely require O(log(n)) more space are nonetheless regarded as in-place.
5. In-place algorithms are typically utilized in a memory-constrained embedded system. They significantly minimize the amount of space needed, although occasionally, the complexity of the algorithm's duration increases.

**Out Place:-**

1. An algorithm not in use is referred to as being out of place or not in use. The additional storage needed by an out-of-place method depends on the input size, as opposed to an in-place approach.
2. As it takes O(n) additional space for merging, the standard merge sort method illustrates an out-of-place algorithm. Although the merger can be done in place, it makes the sorting process more time-consuming.

**Q2. Implement Insertion sort in both (in-place and out-place) manner.**

// Code for implementing using c++.

#include <bits/stdc++.h>

using namespace std;

// Function for in-place insertion sort.

void insertionSort\_ip(int arr[], int n){

int i;

int key;

int j;

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

key = arr[i];

j = i - 1;

// Any elements in array[0..i-1] that are greater than the key should be moved forward from where they are currently.

while (j >= 0 && arr[j] > key){

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

j = j - 1;

}

arr[j + 1] = key;

}

}

// Function for out-place insertion sort.

void insertionSort\_op(int arr[], int newarr[], int n){

int i;

int key;

int j;

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

key = arr[i];

j = i - 1;

// Any elements in array[0..i-1] that are greater than the key should be moved forward from where they are currently.

while (j >= 0 && arr[j] > key){

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

j = j - 1;

}

newarr[j + 1] = key;

}

}

// A function to print an array of given size n.

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

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

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

}

}

int main(){

int n;

cin>>n;

int arr[n];

int newarr[n];

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

cin>>arr[i];

}

insertionSort\_ip(arr,n);

cout<<"array after sorting using in-place insertion: ";

print(arr,n);

cout<<endl;

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

newarr[i]=arr[i];

}

cout<<"array after sorting using out-place insertion: ";

insertionSort\_op(arr,newarr,n);

print(newarr,n);

return 0;

}

**Q3. Suggest some practical examples of using in-place and out-place techniques.**

**In Place:-**

Regardless of the input size, an in-place algorithm requires only a fixed number of integers for the auxiliary variables i j and key. This is accomplished by reading the elements from both ends of the array and swapping them.

**Out Place:-**

In order to replicate the contents of the new array back into the original one, a new array of the same type and size must first be created. It should then be filled with elements from the original array in reverse order. This is not in place since this implementation takes O(n) more space.