Introduction to Sorting

Sorting any sequence means to arrange the elements of that sequence according to some specific criterion.

For Example, the array arr[] = {5, 4, 2, 1, 3} after *sorting in increasing order* will be: arr[] = {1, 2, 3, 4, 5}. The same array after *sorting in descending order* will be: arr[] = {5, 4, 3, 2, 1}.

In-Place Sorting: An in-place sorting algorithm uses constant extra space for producing the output (modifies the given array only). It sorts the list only by modifying the order of the elements within the list.

In this tutorial, we will see three of such in-place sorting algorithms, namely:

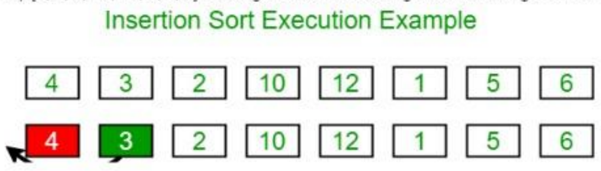
- Insertion Sort
- Selection Sort
- Bubble Sort

Insertion Sort

Insertion Sort is an In-Place sorting algorithm. This algorithm works in a similar way of sorting a deck of playing cards.

The idea is to start iterating from the second element of array till last element and for every element insert at its correct position in the subarray before it.

In the below image you can see, how the array [4, 3, 2, 10, 12, 1, 5, 6] is being sorted in increasing order following the insertion sort algorithm.



Algorithm:

```
Step 1: If the current element is 1st element of array,
        it is already sorted.
Step 2: Pick next element
Step 3: Compare the current element will all elements
        in the sorted sub-array before it.
Step 4: Shift all of the elements in the sub-array before
        the current element which are greater than the current
        element by one place and insert the current element
        at the new empty space.
Step 5: Repeat step 2-3 until the entire array is sorted.
```

Another Example:
arr[] = {**12**, 11, 13, 5, 6}

Let us loop for i = 1 (second element of the array) to 4 (Size of input array - 1).

- *i = 1*, Since 11 is smaller than 12, move 12 and insert 11 before 12.
  11, 12, 13, 5, 6
- *i = 2*, 13 will remain at its position as all elements in A[0..I-1] are smaller than 13
  11, 12, 13, 5, 6
- *i = 3*, 5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.
  5, 11, 12, 13, 6
- *i = 4*, 6 will move to position after 5, and elements from 11 to 13 will move one position ahead of their current position.
  5, 6, 11, 12, 13

Function Implementation:

```c

/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        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), where N is the size of the array.

Bubble Sort

Bubble Sort is also an in-place sorting algorithm. This is the simplest sorting algorithm and it works on the principle that:

> In one iteration if we swap all adjacent elements of an array such that after swap the first element is less than the second element then at the end of the iteration, the first element of the array will be the minimum element.

Bubble-Sort algorithm simply repeats the above steps N-1 times, where N is the size of the array.

Example: Consider the array, arr[] = {5, 1, 4, 2, 8}.

- **First Pass:** (**5 1** 4 2 8) --> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
  (1 **5 4** 2 8) --> (1 **4 5** 2 8), Swap since 5 > 4
  (1 4 **5 2** 8) --> (1 4 **2 5** 8), Swap since 5 > 2
  (1 4 2 **5 8**) --> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.
- **Second Pass:** (**1 4** 2 5 8) --> (**1 4** 2 5 8)
  (1 **4 2** 5 8) --> (1 **2 4** 5 8), Swap since 4 > 2
  (1 2 **4 5** 8) --> (1 2 **4 5** 8)
  (1 2 4 **5 8**) --> (1 2 4 **5 8**)
  Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.
- **Third Pass:** (**1 2** 4 5 8) --> (**1 2** 4 5 8)
  (1 **2 4** 5 8) --> (1 **2 4** 5 8)
  (1 2 **4 5** 8) --> (1 2 **4 5** 8)
  (1 2 4 **5 8**) --> (1 2 4 **5 8**)

Function Implementation:

```

// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
   int i, j;
   for (i = 0; i < n-1; i++)

       // Last i elements are already in place
       for (j = 0; j < n-i-1; j++)
           if (arr[j] > arr[j+1])
              swap(&arr[j], &arr[j+1]);
}

```

Note: The above solution can be further optimized by keeping a flag to check if the array is already sorted in the first pass itself and to stop any further iteration.

Time Complexity: O(N^2)

Selection Sort

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

1. The subarray which is already sorted.
2. Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

Following example explains the above steps:

```
arr[] = 64 25 12 22 11.

// Find the minimum element in arr[0...4]
// and place it at beginning
11 25 12 22 64

// Find the minimum element in arr[1...4]
// and place it at beginning of arr[1...4]
11 12 25 22 64

// Find the minimum element in arr[2...4]
// and place it at beginning of arr[2...4]
11 12 22 25 64

// Find the minimum element in arr[3...4]
// and place it at beginning of arr[3...4]
11 12 22 25 64
```

Function Implementation:

```

void selectionSort(int arr[], int n)
{
    int i, j, min_idx;

    // One by one move boundary of unsorted subarray
    for (i = 0; i < n-1; i++)
    {
        // Find the minimum element in unsorted array
        min_idx = i;
        for (j = i+1; j < n; j++)
          if (arr[j] < arr[min_idx])
            min_idx = j;

        // Swap the found minimum element with the first element
        swap(&arr[min_idx], &arr[i]);
    }
}

```

Time Complexity: O(N^2)