Sorting Algorithms

Prepared by Jakir Hasan

Bubble Sort

This sorting algorithm is a comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order.

Implementation:

```
#include <bits/stdc++.h>
using namespace std;

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

int main()
{
    int arr[10000], n, i, j;
    // n: Size of the array
    cin>> n;

    for (i = 1; i <= n; i++)
        cin>> arr[i];

    bubble_sort(arr, n);
    cout<< "\nSorted array:" << "\n";

    for (i = 1; i <= n; i++)
        cout<< arr[i] << " ";
    cout<< "\n";

    return 0;
}


void bubble_sort(int arr[], int n)
```

```
{
    /* Sorts the array in ascending order */

    int i, j, temp;
    for (i = 1; i <= n; i++)
    {
        for (j = 1; j <= n-1; j++)
        {
            if (arr[j] > arr[j+1])
            {
                // Swap arr[j] with arr[j+1]

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

/*
5
14 33 27 35 10
*/
```

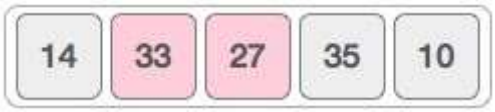
How Bubble Sort Works?

We take an unsorted array for our example. Bubble sort takes $O(n^2)$ time so we're keeping it short and precise.

Bubble sort starts with the very first two elements, comparing them to check which one is greater.

In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.

We find that 27 is smaller than 33 and these two values must be swapped.



The new array should look like this −

Next we compare 33 and 35. We find that both are in already sorted positions.

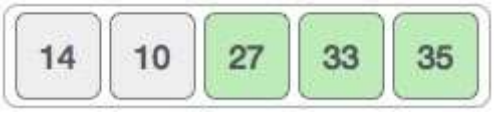
Then we move to the next two values, 35 and 10.

We know then that 10 is smaller 35. Hence they are not sorted.

We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −

To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −

Notice that after each iteration, at least one value moves at the end.



And when there's no swap required, bubble sort learns that an array is completely sorted.

Now we should look into some practical aspects of bubble sort.

@Credit: tutorialpoints.com

Insertion Sort

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'inserted' in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, insertion sort.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$, where n is the number of items.

How Insertion Sort Works?

We take an unsorted array for our example.

Insertion sort compares the first two elements.

It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.

Insertion sort moves ahead and compares 33 with 27.

And finds that 33 is not in the correct position.

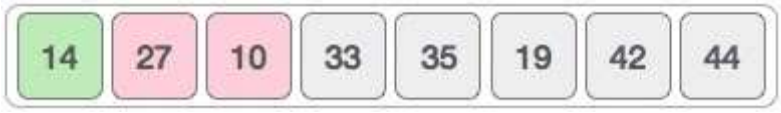
It swaps 33 with 27. It also checks with all the elements of the sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.

These values are not in a sorted order.

So we swap them.

However, swapping makes 27 and 10 unsorted.



Hence, we swap them too.

Again we find 14 and 10 in an unsorted order.

We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.

This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

Pseudocode:

```
i ← 1
while i < length(A)
    j ← i
    while j > 0 and A[j-1] > A[j]
        swap A[j] and A[j-1]
        j ← j - 1
    end while
    i ← i + 1
end while
```

Example:

```
3   7   4   9   5   2   6   1
3*  7   4   9   5   2   6   1
3   7*  4   9   5   2   6   1
3   4*  7   9   5   2   6   1
3   4   7   9*  5   2   6   1
3   4   5*  7   9   2   6   1
2*  3   4   5   7   9   6   1
2   3   4   5   6*  7   9   1
1*  2   3   4   5   6   7   9
```

Implementation:

```
#include <bits/stdc++.h>
using namespace std;

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

int main()
{
    int arr[10000], n, i, j;
    cin>> n;

    for (i = 1; i <= n; i++)
        cin>> arr[i];

    insertion_sort(arr, n);

    cout<< "\nAfter sort:" << "\n";

    for (i = 1; i <= n; i++)
        cout<< arr[i] << " ";
    cout<< "\n";
```

```
    return 0;
}


void insertion_sort(int arr[], int n)
{
    /* Sorts array in ascending order */
    int i, j, temp;

    for (i = 2; i <= n; i++)
    {
        for (j = i; j > 1; j--)
        {
            if (arr[j] < arr[j-1])
            {
                // swap arr[j] with arr[j-1]
                temp = arr[j];
                arr[j] = arr[j-1];
                arr[j-1] = temp;
            }
        }
    }

}
/*
8
3  7  4  9  5  2  6  1
*/
```

@Credit: tutorialspoint.com, wikipedia

Selection Sort

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving the unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst case complexities are of $O(n^2)$, where n is the number of items.

How Selection Sort Works?

Consider the following depicted array as an example.

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

After two iterations, two least values are positioned at the beginning in a sorted manner.

The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process -

Implementation:

```
#include <bits/stdc++.h>
using namespace std;

void selection_sort(int arr[], int n);
```

```
int main()
{
    int arr[10000], n, i, j;
    cin>> n;

    for (i = 1; i <= n; i++)
        cin>> arr[i];

    selection_sort(arr, n);

    cout<< "After sort:" << "\n";

    for (i = 1; i <= n; i++)
        cout<< arr[i] << " ";
    cout<< "\n";

    return 0;
}


void selection_sort(int arr[], int n)
{
    /* Sorts array in ascending order */
    int i, j, temp, k;

    for (i = 1; i <= n; i++)
    {
        for (j = i; j <= n; j++)
        {
            if (arr[j] < arr[i])
            {
                temp = arr[j];
                arr[j] = arr[i];
                arr[i] = temp;
            }

        }

    }
}

/*
```

```
8
3  7  4  9  5  2  6  1
*/
```

@Credit: tutorialspoint.com

Quick Sort

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are O(n2), respectively.

Pseudocode:

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q-1)
    QUICKSORT(A, q+1, r)


PARTITION(A, p, r)
x = A[r]
i = p-1
for j = p to r-1
    if A[j] <= x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i+1] with A[r]
return i+1
```

Implementation:

```
#include <bits/stdc++.h>
using namespace std;

void quick_sort(int arr[], int low, int high);
int partition(int arr[], int low, int high);

int main()
{
    int n, i, arr[10000];
    cin>> n;

    for (i = 1; i <= n; i++)
        cin>> arr[i];

    quick_sort(arr, 1, n);
    cout<< "After sorting:" << "\n";

    for (i = 1; i <= n; i++)
        cout<< arr[i] << " ";
    cout<< "\n";

    return 0;
}


void quick_sort(int arr[], int low, int high)
{
    if (low >= high)
        return;

    int pivot;
    pivot = partition(arr, low, high);

    quick_sort(arr, low, pivot-1);
    quick_sort(arr, pivot+1, high);
}


int partition(int arr[], int low, int high)
{
```

```
    int i, j, pivot, temp, k;
    pivot = arr[high];

    i = low-1;
    for (j = low; j < high; j++)
    {
        if (arr[j] < pivot)
        {
            i += 1;

            // Swapping elements
            temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }

    i += 1;
    temp = arr[i];
    arr[i] = arr[high];
    arr[high] = temp;

    return i;
}

/*
8
2 8 7 1 3 5 6 4
*/
```

@Credit: CLRS, tutorialspoint.com

Merge Sort

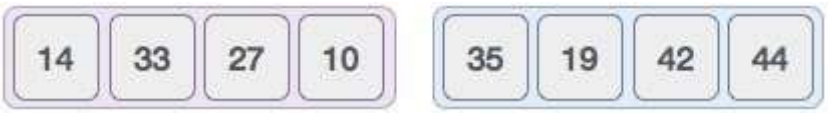
Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(n log n), it is one of the most respected algorithms.

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

How Merge Sort Works?

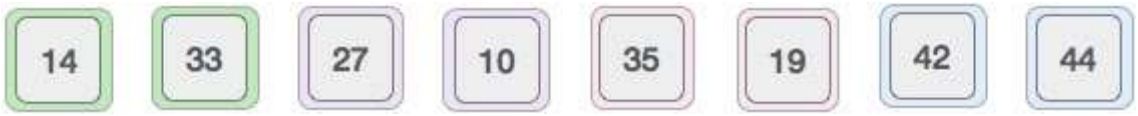
To understand merge sort, we take an unsorted array as the following −

We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.



This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

We further divide these arrays and we achieve atomic value which can no longer be divided.



Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

We first compare the elements for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.

In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.

After the final merging, the list should look like this −

Implementation:

```
#include <bits/stdc++.h>
using namespace std;

void merge_sort(int arr[], int low, int high);
void merge(int arr[], int low, int mid, int high);

int main()
{
    int n, arr[100000], i;
    cin>> n;

    for (i = 0; i < n; i++)
    {
        cin>> arr[i];
    }

    merge_sort(arr, 0, n-1);

    for (i = 0; i < n; i++)
        cout<< arr[i] << " ";
    cout<< "\n";

    return 0;
}


void merge_sort(int arr[], int low, int high)
{
    int mid;
```

```
    if (low < high)
    {
        mid = (low + high)/2;
        merge_sort(arr, low, mid);
        merge_sort(arr, mid+1, high);
        merge(arr, low, mid, high);
    }
}


void merge(int arr[], int low, int mid, int high)
{
    int i = low, j = mid + 1, k = 0;
    int temp[high-low+1];

    while (i <= mid && j <= high)
    {
        if (arr[i] < arr[j])
        {
            temp[k] = arr[i];
            i += 1;
        }
        else
        {
            temp[k] = arr[j];
            j += 1;
        }
        k += 1;
    }

    while (i <= mid)
    {
        temp[k] = arr[i];
        i += 1;
        k += 1;
    }

    while (j <= high)
    {
        temp[k] = arr[j];
        j += 1;
        k += 1;
    }
```

```
    k = 0;
    for (i = low; i <= high; i++)
    {
        arr[i] = temp[k];
        k += 1;
    }
}

/*
5
1 4 3 2 5
*/
```