

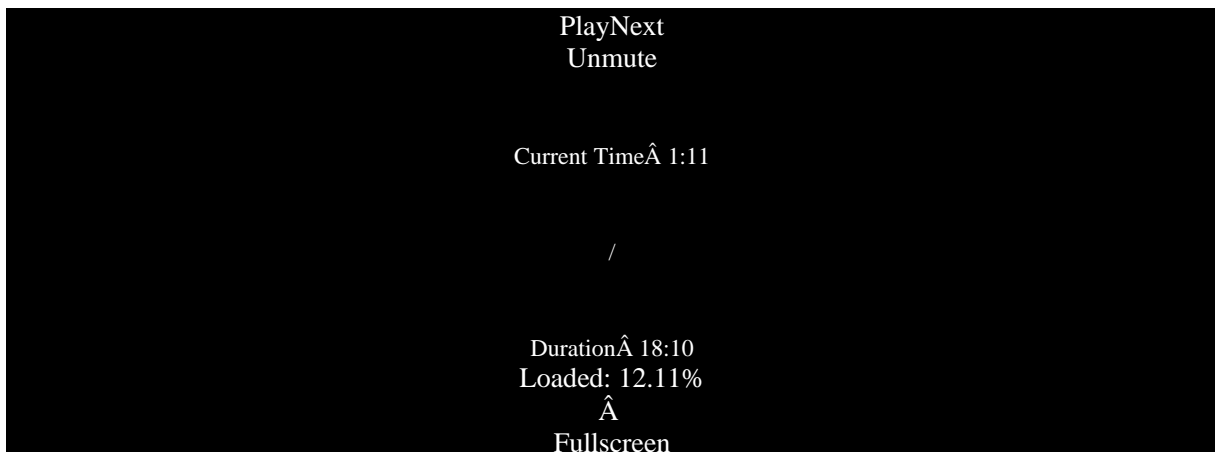
Insertion Sort Algorithm

In this article, we will discuss the Insertion sort Algorithm. The working procedure of insertion sort is also simple. This article will be very helpful and interesting to students as they might face insertion sort as a question in their examinations. So, it is important to discuss the topic.

Insertion sort works similar to the sorting of playing cards in hands. It is assumed that the first card is already sorted in the card game, and then we select an unsorted card. If the selected unsorted card is greater than the first card, it will be placed at the right side; otherwise, it will be placed at the left side. Similarly, all unsorted cards are taken and put in their exact place.

The same approach is applied in insertion sort. The idea behind the insertion sort is that first take one element, iterate it through the sorted array. Although it is simple to use, it is not appropriate for large data sets as the time complexity of insertion sort in the average case and worst case is $O(n^2)$, where n is the number of items. Insertion sort is less efficient than the other sorting algorithms like heap sort, quick sort, merge sort, etc.

Insertion sort has various advantages such as -



- Simple implementation
- Efficient for small data sets
- Adaptive, i.e., it is appropriate for data sets that are already substantially sorted.

Now, let's see the algorithm of insertion sort.

Algorithm

The simple steps of achieving the insertion sort are listed as follows -

Step 1 - If the element is the first element, assume that it is already sorted. Return 1.

Step2 - Pick the next element, and store it separately in a **key**.

Step3 - Now, compare the **key** with all elements in the sorted array.

Step 4 - If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.

Step 5 - Insert the value.

Step 6 - Repeat until the array is sorted.

Working of Insertion sort Algorithm

Now, let's see the working of the insertion sort Algorithm.

To understand the working of the insertion sort algorithm, let's take an unsorted array. It will be easier to understand the insertion sort via an example.

Let the elements of array are -

12	31	25	8	32	17
----	----	----	---	----	----

Initially, the first two elements are compared in insertion sort.

12	31	25	8	32	17
----	----	----	---	----	----

Here, 31 is greater than 12. That means both elements are already in ascending order. So, for now, 12 is stored in a sorted sub-array.

12	31	25	8	32	17
----	----	----	---	----	----

Now, move to the next two elements and compare them.

12	31	25	8	32	17
----	----	----	---	----	----

12	31	25	8	32	17
----	----	----	---	----	----

Here, 25 is smaller than 31. So, 31 is not at correct position. Now, swap 31 with 25. Along with swapping, insertion sort will also check it with all elements in the sorted array.

For now, the sorted array has only one element, i.e. 12. So, 25 is greater than 12. Hence, the sorted array remains sorted after swapping.

12	25	31	8	32	17
----	----	----	---	----	----

Now, two elements in the sorted array are 12 and 25. Move forward to the next elements that are 31 and 8.

12	25	31	8	32	17
----	----	----	---	----	----

12	25	31	8	32	17
----	----	----	---	----	----

Both 31 and 8 are not sorted. So, swap them.

12	25	8	31	32	17
----	----	---	----	----	----

After swapping, elements 25 and 8 are unsorted.

12	25	8	31	32	17
----	----	---	----	----	----

So, swap them.

12	8	25	31	32	17
----	---	----	----	----	----

Now, elements 12 and 8 are unsorted.

12	8	25	31	32	17
----	---	----	----	----	----

So, swap them too.

8	12	25	31	32	17
---	----	----	----	----	----

Now, the sorted array has three items that are 8, 12 and 25. Move to the next items that are 31 and 32.

8	12	25	31	32	17
---	----	----	----	----	----

Hence, they are already sorted. Now, the sorted array includes 8, 12, 25 and 31.

8	12	25	31	32	17
---	----	----	----	----	----

Move to the next elements that are 32 and 17.

8	12	25	31	32	17
---	----	----	----	----	----

17 is smaller than 32. So, swap them.

8	12	25	31	17	32
---	----	----	----	----	----

8	12	25	31	17	32
---	----	----	----	----	----

Swapping makes 31 and 17 unsorted. So, swap them too.

8	12	25	17	31	32
---	----	----	----	----	----

8	12	25	17	31	32
---	----	----	----	----	----

Now, swapping makes 25 and 17 unsorted. So, perform swapping again.

8	12	17	25	31	32
---	----	----	----	----	----

Now, the array is completely sorted.

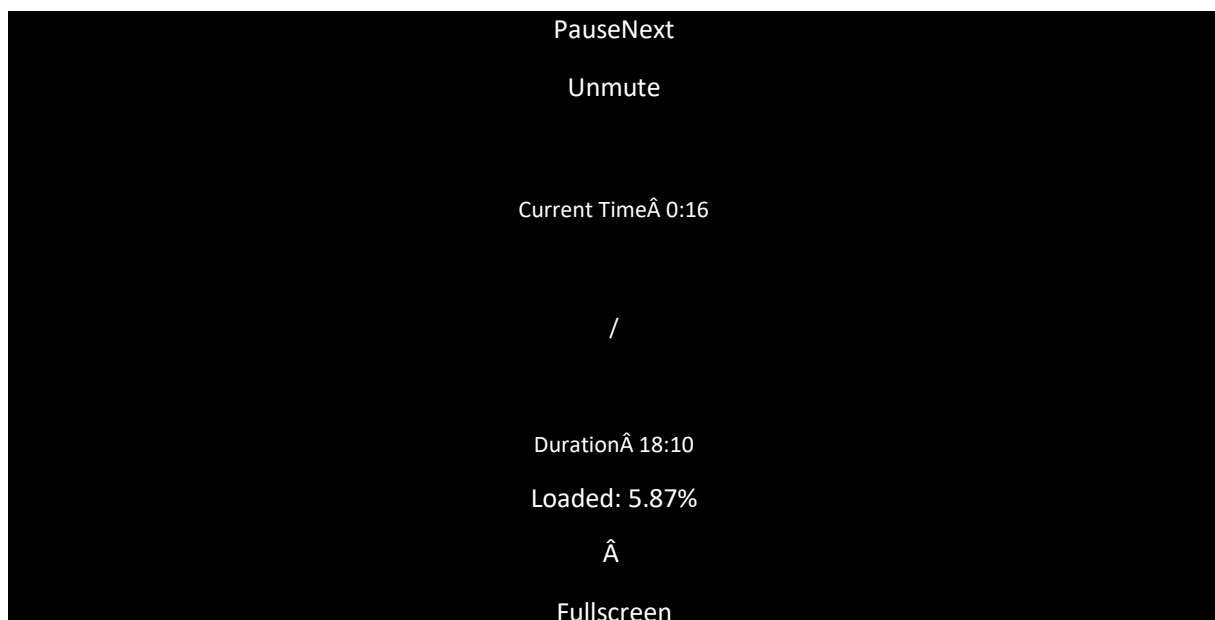
Shell Sort Algorithm

In this article, we will discuss the shell sort algorithm. Shell sort is the generalization of insertion sort, which overcomes the drawbacks of insertion sort by comparing elements separated by a gap of several positions.

It is a sorting algorithm that is an extended version of insertion sort. Shell sort has improved the average time complexity of insertion sort. As similar to insertion sort, it is a comparison-based and in-place sorting algorithm. Shell sort is efficient for medium-sized data sets.

In insertion sort, at a time, elements can be moved ahead by one position only. To move an element to a far-away position, many movements are required that increase the algorithm's execution time. But shell sort overcomes this drawback of insertion sort. It allows the movement and swapping of far-away elements as well.

This algorithm first sorts the elements that are far away from each other, then it subsequently reduces the gap between them. This gap is called as **interval**. This interval can be calculated by using the **Knuth's** formula given below -



1. $h = h * 3 + 1$
2. where, 'h' is the interval having initial value 1.

Now, let's see the algorithm of shell sort.

Algorithm

The simple steps of achieving the shell sort are listed as follows -

1. ShellSort(a, n) // 'a' is the given array, 'n' is the size of array
2. for (interval = n/2; interval > 0; interval /= 2)
3. for (i = interval; i < n; i += 1)
4. temp = a[i];

5. for ($j = i; j \geq \text{interval} \ \&\& \ a[j - \text{interval}] > \text{temp}; j -= \text{interval}$)
6. $a[j] = a[j - \text{interval}]$;
7. $a[j] = \text{temp}$;
8. End ShellSort

Working of Shell sort Algorithm

Now, let's see the working of the shell sort Algorithm.

To understand the working of the shell sort algorithm, let's take an unsorted array. It will be easier to understand the shell sort via an example.

Let the elements of array are -

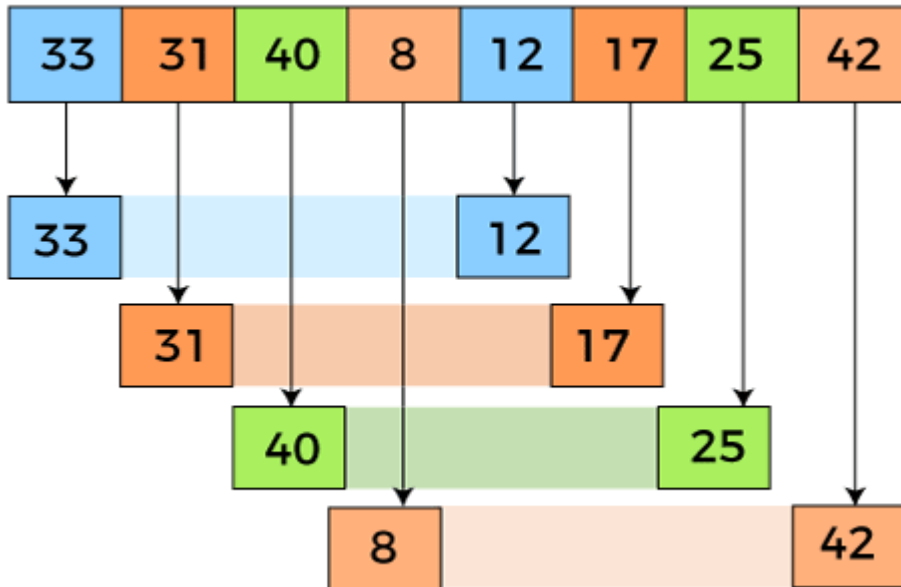
33	31	40	8	12	17	25	42
----	----	----	---	----	----	----	----

We will use the original sequence of shell sort, i.e., $N/2, N/4, \dots, 1$ as the intervals.

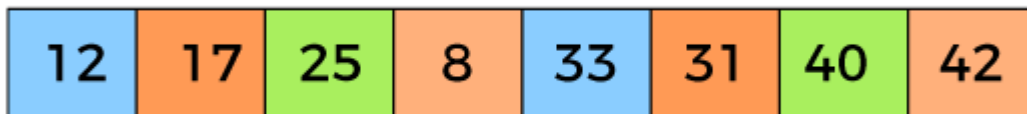
In the first loop, n is equal to 8 (size of the array), so the elements are lying at the interval of 4 ($n/2 = 4$). Elements will be compared and swapped if they are not in order.

Here, in the first loop, the element at the 0th position will be compared with the element at 4th position. If the 0th element is greater, it will be swapped with the element at 4th position. Otherwise, it remains the same. This process will continue for the remaining elements.

At the interval of 4, the sublists are {33, 12}, {31, 17}, {40, 25}, {8, 42}.

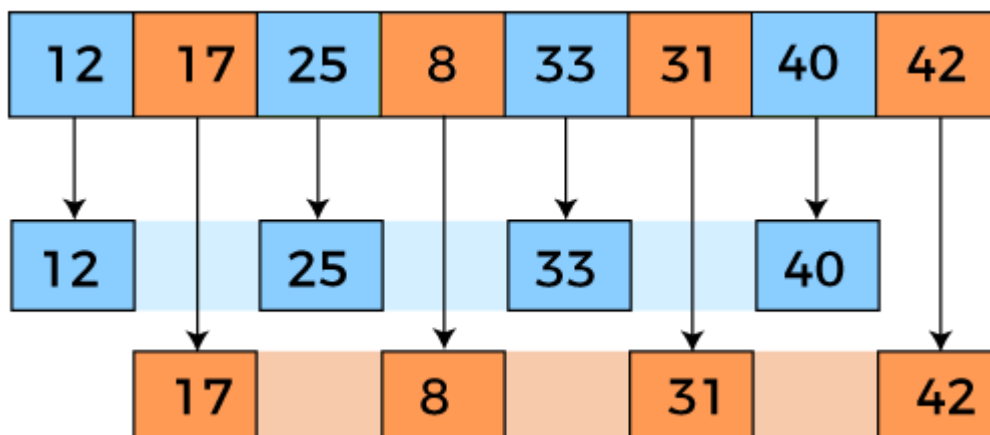


Now, we have to compare the values in every sub-list. After comparing, we have to swap them if required in the original array. After comparing and swapping, the updated array will look as follows -



In the second loop, elements are lying at the interval of 2 ($n/4 = 2$), where $n = 8$.

Now, we are taking the interval of 2 to sort the rest of the array. With an interval of 2, two sublists will be generated - {12, 25, 33, 40}, and {17, 8, 31, 42}.



Now, we again have to compare the values in every sub-list. After comparing, we have to swap them if required in the original array. After comparing and swapping, the updated array will look as follows -

12	8	25	17	33	31	40	42
----	---	----	----	----	----	----	----

In the third loop, elements are lying at the interval of 1 ($n/8 = 1$), where $n = 8$. At last, we use the interval of value 1 to sort the rest of the array elements. In this step, shell sort uses insertion sort to sort the array elements.

12	8	25	17	33	31	40	42
12	8	25	17	33	31	40	42
8	12	25	17	33	31	40	42
8	12	25	17	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	31	33	40	42
8	12	17	25	31	33	40	42
8	12	17	25	31	33	40	42

Now, the array is sorted in ascending order.

Shell sort complexity

Now, let's see the time complexity of Shell sort in the best case, average case, and worst case. We will also see the space complexity of the Shell sort.

1. Time Complexity

Case	Time Complexity
Best Case	$O(n \cdot \log n)$
Average Case	$O(n \cdot \log(n)^2)$
Worst Case	$O(n^2)$