Bubble Sort Algorithm

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

Bubble Sort Algorithm

Bubble Sort is an elementary sorting algorithm, which works by repeatedly exchanging adjacent elements, if necessary. When no exchanges are required, the file is sorted.

We assume **list** is an array of **n** elements. We further assume that **swap** function swaps the values of the given array elements.

Step 1 − Check if the first element in the input array is greater than the next element in the array.

Step 2 − If it is greater, swap the two elements; otherwise move the pointer forward in the array.

Step 3 − Repeat Step 2 until we reach the end of the array.

Step 4 − Check if the elements are sorted; if not, repeat the same process (Step 1 to Step 3) from the last element of the array to the first.

Step 5 − The final output achieved is the sorted array.

```
Algorithm: Sequential-Bubble-Sort (A)
fori ← 1 to length [A] do
for j ← length [A] down-to i +1 do
   if A[A] < A[j-1] then
      Exchange A[j] ↔ A[j-1]
```

Pseudocode

We observe in algorithm that Bubble Sort compares each pair of array element unless the whole array is completely sorted in an ascending order. This may cause a few complexity issues like what if the array needs no more swapping as all the elements are already ascending.

To ease-out the issue, we use one flag variable **swapped** which will help us see if any swap has happened or not. If no swap has occurred, i.e. the array requires no more processing to be sorted, it will come out of the loop.

Pseudocode of bubble sort algorithm can be written as follows −

```
voidbubbleSort(int numbers[], intarray_size){
   inti, j, temp;
   for (i = (array_size - 1); i>= 0; i--)
   for (j = 1; j <= i; j++)
   if (numbers[j-1] > numbers[j]){
      temp = numbers[j-1];
      numbers[j-1] = numbers[j];
      numbers[j] = temp;
   }
}
```

Analysis

Here, the number of comparisons are

```
1 + 2 + 3 + ... + (n - 1) = n(n - 1)/2 = O(n2)
```

Clearly, the graph shows the $\mathbf{n^2}$ nature of the bubble sort.

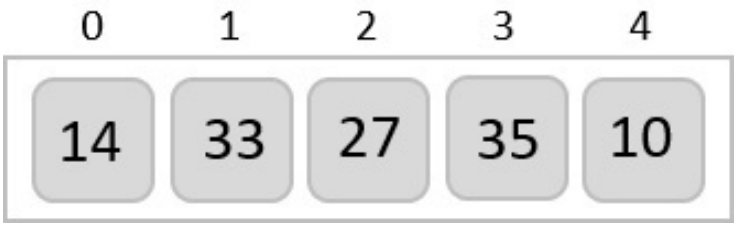
In this algorithm, the number of comparison is irrespective of the data set, i.e. whether the provided input elements are in sorted order or in reverse order or at random.

Memory Requirement

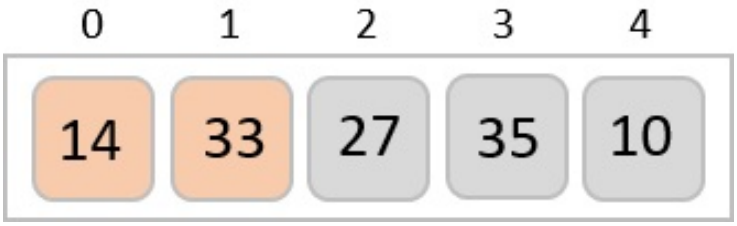
From the algorithm stated above, it is clear that bubble sort does not require extra memory.

Example

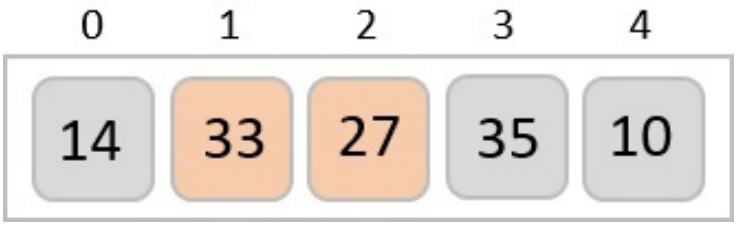
We take an unsorted array for our example. Bubble sort takes O(n2) time so we're keeping it short and precise.



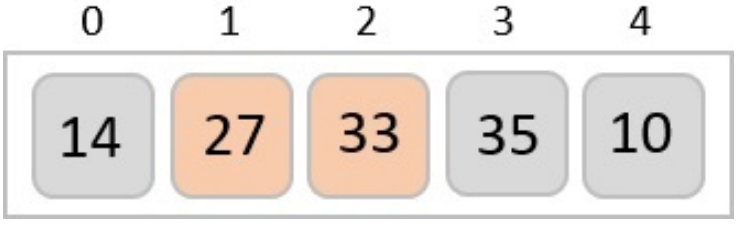
Bubble sort starts with very first two elements, comparing them to check which one is greater.



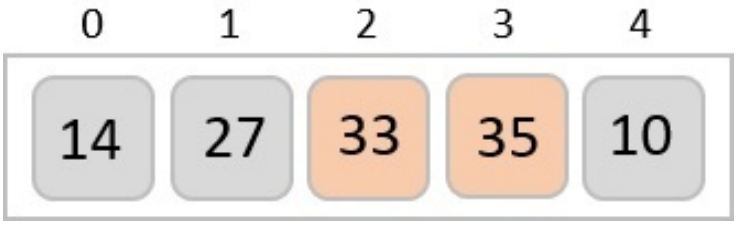
In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.



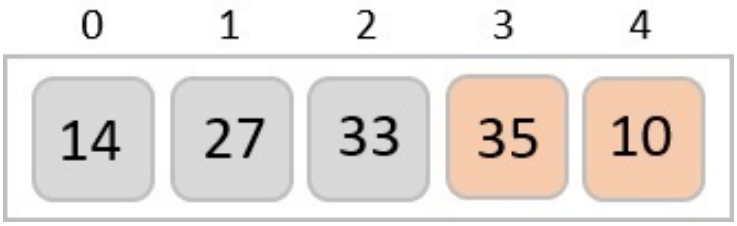
We find that 27 is smaller than 33 and these two values must be swapped.



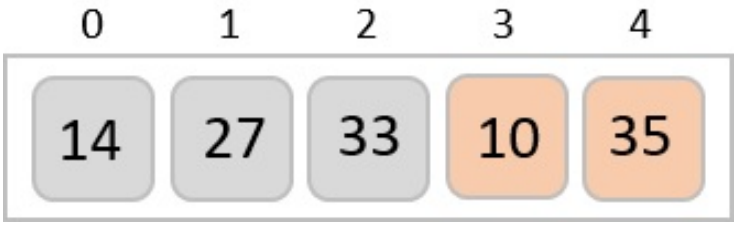
Next we compare 33 and 35. We find that both are in already sorted positions.



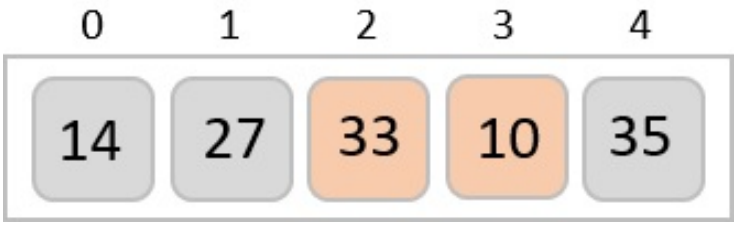
Then we move to the next two values, 35 and 10.

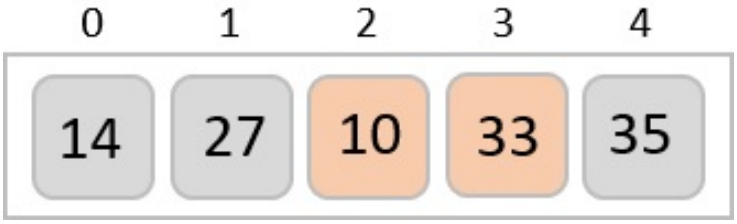


We know then that 10 is smaller 35. Hence they are not sorted. We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this –

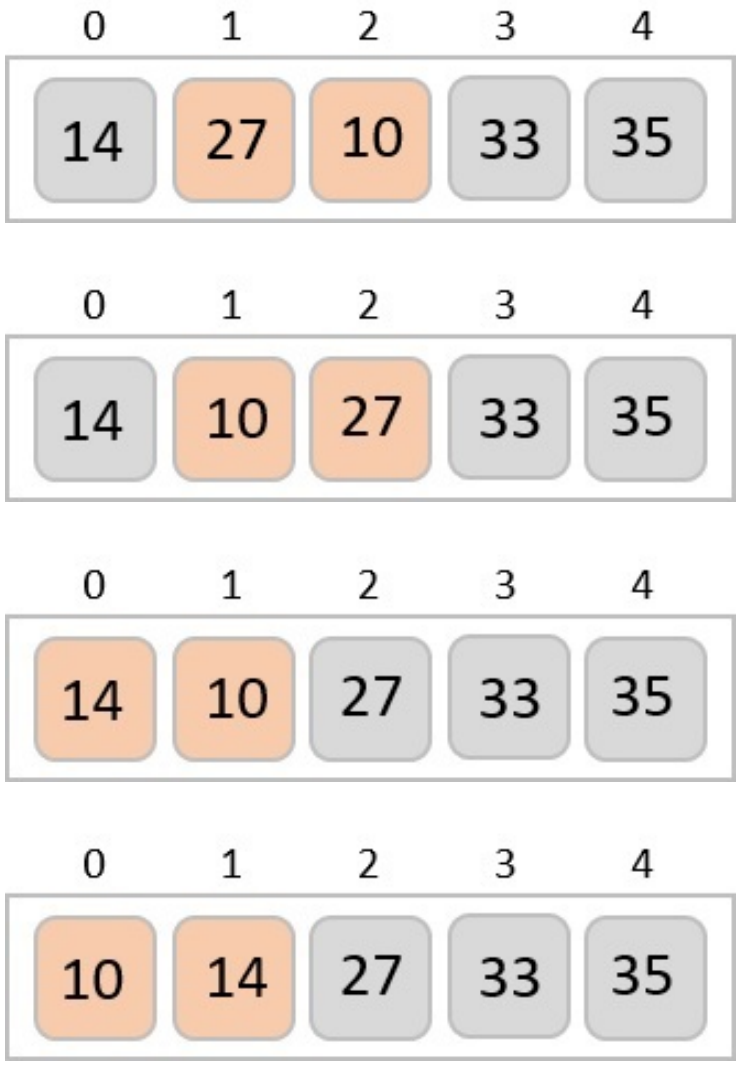


To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this –

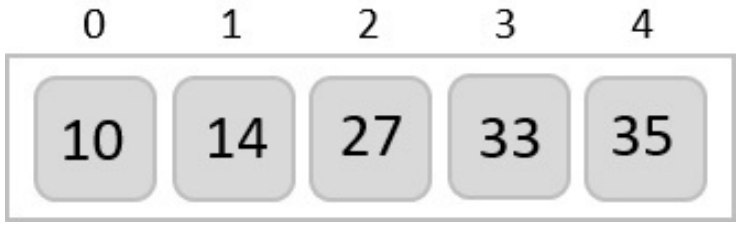




Notice that after each iteration, at least one value moves at the end.



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.

Implementation

One more issue we did not address in our original algorithm and its improvised pseudocode, is that, after every iteration the highest values settles down at the end of the array. Hence, the next iteration need not include already sorted elements. For this purpose, in our implementation, we restrict the inner loop to avoid already sorted values.

C | C++ | Java | Python

```c
#include <stdio.h>
void bubbleSort(int array[], int size){
   for(int i = 0; i<size; i++) {
      int swaps = 0; //flag to detect any swap is there or not
      for(int j = 0; j<size-i-1; j++) {
         if(array[j] > array[j+1]) { //when the current item is
            int temp;
            temp = array[j];
            array[j] = array[j+1];
            array[j+1] = temp;
            swaps = 1; //set swap flag
         }
      }
      if(!swaps)
         break; // No swap in this pass, so array is sorted
   }
}
int main(){
   int n;
   n = 5;
   int arr[5] = {67, 44, 82, 17, 20}; //initialize an array
   printf("Array before Sorting: ");
   for(int i = 0; i<n; i++)
      printf("%d ",arr[i]);
   printf("\n");
   bubbleSort(arr, n);
   printf("Array after Sorting: ");
   for(int i = 0; i<n; i++)
      printf("%d ", arr[i]);
   printf("\n");
}
```

Output

```
Array before Sorting: 67 44 82 17 20
Array after Sorting: 17 20 44 67 82
```