Sorting Algorithms

Sorting algorithms are methods used in data structures to arrange elements in a specific order, such as ascending or descending. Sorting is essential for improving data search efficiency and organizing information meaningfully.

OR

Sorting is the process of arranging data elements in a specific order, typically **ascending** or **descending**. Sorting is essential because:

- It helps in **efficient searching** (e.g., Binary Search needs sorted data).
- It makes data **easier to manage and analyze**.
- It is used in **database queries, data visualization, and optimization tasks**.

Insertion Sort Algorithm

Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.

Insertion sort works similarly as we sort cards in our hand in a card game.

We assume that the first card is already sorted then, we select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left. In the same way, other unsorted cards are taken and put in their right place.

A similar approach is used by insertion sort.

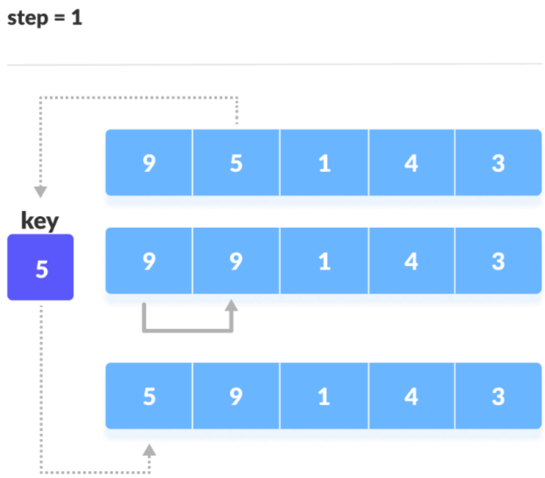
Working of Insertion Sort

Suppose we need to sort the following array.

Initial array

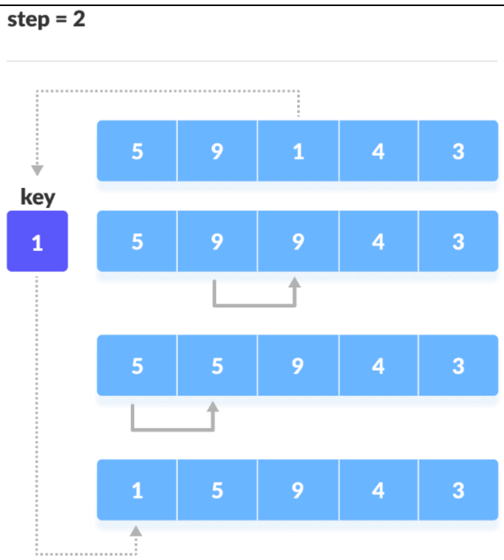
1. The first element in the array is assumed to be sorted. Take the second element and store it separately in key.

   Compare key with the first element. If the first element is greater than key, then key is placed in front of the first element.



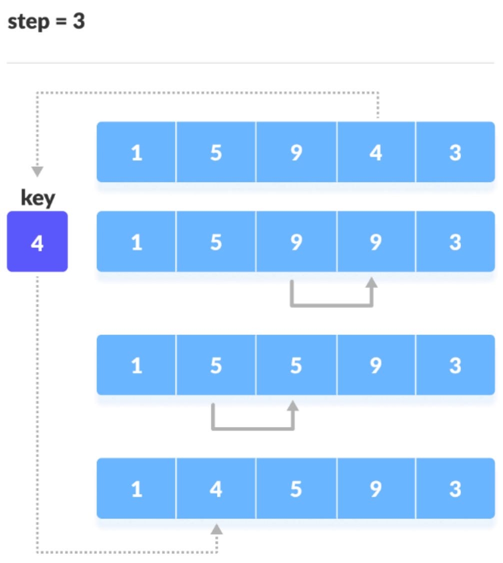
If the first element is greater than key, then key is placed in front of the first element.

2. Now, the first two elements are sorted.
   Take the third element and compare it with the elements on the left of it. Placed it just behind the element smaller than it. If there is no element smaller than it, then place it at the beginning of the array.

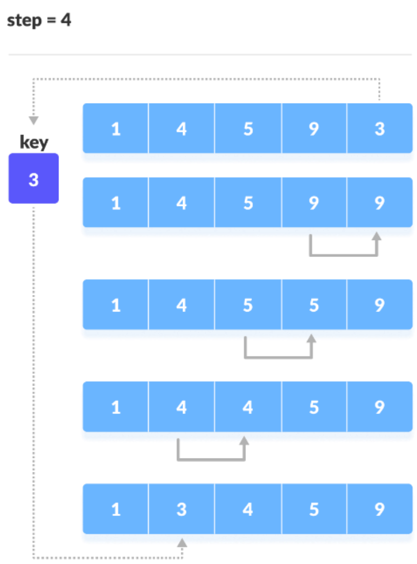


Place 1 at the beginning

3. Similarly, place every unsorted element at its correct position.



Place 4 behind 1



Place 3 behind 1 and the array is sorted

Insertion Sort Algorithm

```
insertionSort(array)
 mark first element as sorted
 for each unsorted element X
  'extract' the element X
  for j <- lastSortedIndex down to 0
   if current element j > X
    move sorted element to the right by 1
  break loop and insert X here
end insertionSort
```

```
// Insertion sort in C++

#include <iostream>

using namespace std;

// Function to print an array

void printArray(int array[], int size) {

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

    cout << array[i] << " ";

  }

  cout << endl;

}

void insertionSort(int array[], int size) {

  for (int step = 1; step < size; step++) {

    int key = array[step];

    int j = step - 1;

    // Compare key with each element on the left of it until an element smaller than
```

```
    // it is found.
    // For descending order, change key<array[j] to key>array[j].
    while (j >=0 && key < array[j]) {
      array[j + 1] = array[j];
      --j;
    }
    array[j + 1] = key;
  }
}
// Driver code
int main() {
  int data[] = {9, 5, 1, 4, 3};
  int size = sizeof(data) / sizeof(data[0]);
  insertionSort(data, size);
  cout << "Sorted array in ascending order:\n";
  printArray(data, size);
}
```

Insertion Sort Complexity

Time Complexity	
Best	O(n)
Worst	$O(n^2)$
Average	$O(n^2)$
Space Complexity	O(1)
Stability	Yes

Time Complexities

- **Worst Case Complexity:** $O(n^2)$

  Suppose, an array is in ascending order, and you want to sort it in descending order. In this case, worst case complexity occurs.

  Each element has to be compared with each of the other elements so, for every nth element, $(n-1)$ number of comparisons are made.

  Thus, the total number of comparisons = $n*(n-1) \sim n^2$

- **Best Case Complexity:** $O(n)$

  When the array is already sorted, the outer loop runs for n number of times whereas the inner loop does not run at all. So, there are only n number of comparisons. Thus, complexity is linear.

- **Average Case Complexity:** $O(n^2)$

  It occurs when the elements of an array are in jumbled order (neither ascending nor descending).

Space Complexity

Space complexity is $O(1)$ because an extra variable key is used.

Insertion Sort Applications

The insertion sort is used when:

- the array is has a small number of elements
- there are only a few elements left to be sorted

Selection Sort Algorithm

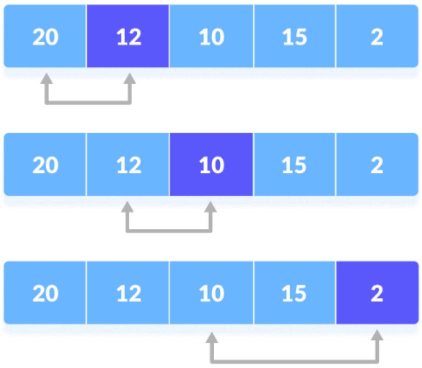
Selection sort is a sorting algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.

Working of Selection Sort

Select first element as minimum

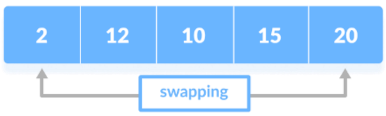
1. Set the first element as minimum.
2. Compare minimum with the second element. If the second element is smaller than minimum, assign the second element as minimum.

   Compare minimum with the third element. Again, if the third element is smaller, then assign minimum to the third element otherwise do nothing. The process goes on until the last element.



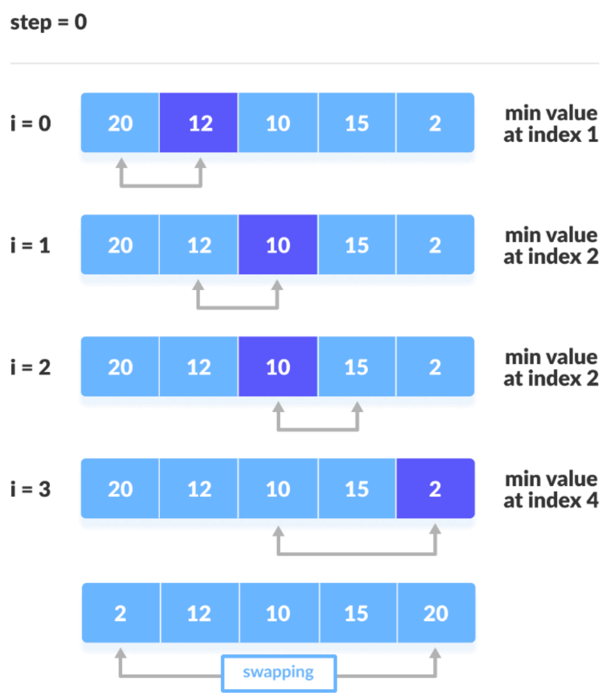
Compare minimum with the remaining elements

3. After each iteration, minimum is placed in the front of the unsorted list.

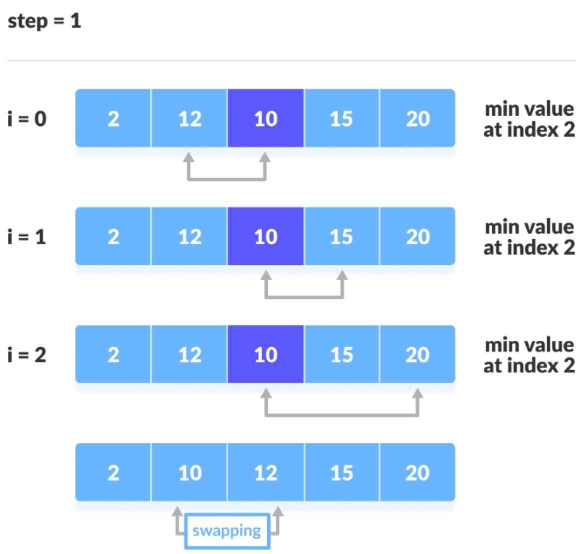


Swap the first with minimum

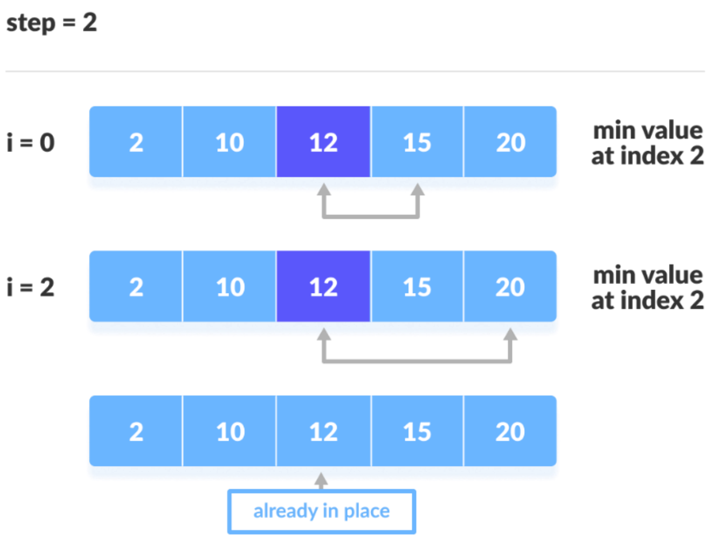
4. For each iteration, indexing starts from the first unsorted element. Step 1 to 3 are repeated until all the elements are placed at their correct positions.



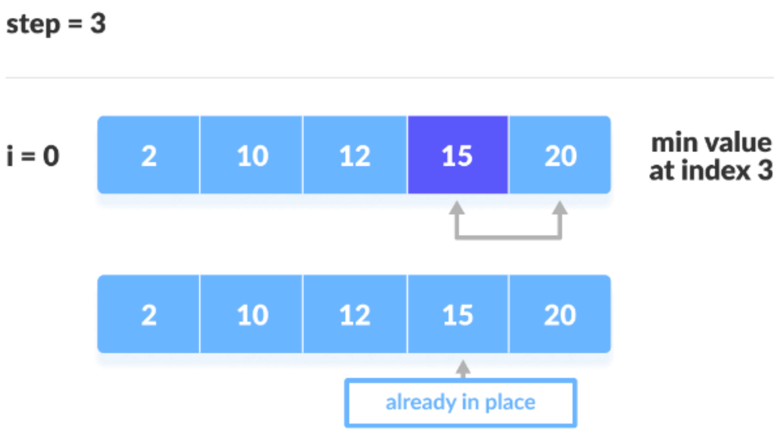
The first iteration



The second iteration



The third iteration



The fourth iteration

Selection Sort Algorithm

```
selectionSort(array, size)
  for i from 0 to size - 1 do
    set i as the index of the current minimum
    for j from i + 1 to size - 1 do
      if array[j] < array[current minimum]
```

```
        set j as the new current minimum index
    if current minimum is not i
        swap array[i] with array[current minimum]
end selectionSort
```

```
// Selection sort in C++
#include <iostream>
using namespace std;
// function to swap the the position of two elements
void swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}
// function to print an array
void printArray(int array[], int size) {
  for (int i = 0; i < size; i++) {
    cout << array[i] << " ";
  }
  cout << endl;
}
void selectionSort(int array[], int size) {
  for (int step = 0; step < size - 1; step++) {
    int min_idx = step;
    for (int i = step + 1; i < size; i++) {
```

```
      // To sort in descending order, change > to < in this line.
      // Select the minimum element in each loop.
      if (array[i] < array[min_idx])
        min_idx = i;
    }
    // put min at the correct position
    swap(&array[min_idx], &array[step]);
  }
}
// driver code
int main() {
  int data[] = {20, 12, 10, 15, 2};
  int size = sizeof(data) / sizeof(data[0]);
  selectionSort(data, size);
  cout << "Sorted array in Acsending Order:\n";
  printArray(data, size);
}
```

Selection Sort Complexity

Time Complexity	
Best	$O(n^2)$
Worst	$O(n^2)$
Average	$O(n^2)$
Space Complexity	$O(1)$
Stability	No

Cycle	Number of Comparison
1st	(n-1)
2nd	(n-2)
3rd	(n-3)
...	...
last	1

Number of comparisons: $(n - 1) + (n - 2) + (n - 3) + ... + 1 = n(n - 1) / 2$ nearly equals to n^2.

Complexity = $O(n^2)$

Also, we can analyze the complexity by simply observing the number of loops. There are 2 loops so the complexity is $n*n = n^2$.

Time Complexities:

- **Worst Case Complexity:** $O(n^2)$

  If we want to sort in ascending order and the array is in descending order then, the worst case occurs.

- **Best Case Complexity:** $O(n^2)$

  It occurs when the array is already sorted

- **Average Case Complexity:** $O(n^2)$

  It occurs when the elements of the array are in jumbled order (neither ascending nor descending).

The time complexity of the selection sort is the same in all cases. At every step, you have to find the minimum element and put it in the right place. The minimum element is not known until the end of the array is not reached.

Space Complexity:

Space complexity is $O(1)$ because an extra variable min_idx is used.

Selection Sort Applications

The selection sort is used when

- a small list is to be sorted
- cost of swapping does not matter

- checking of all the elements is compulsory
- cost of writing to a memory matters like in flash memory (number of writes/swaps is $O(n)$ as compared to $O(n^2)$ of bubble sort)

Bubble Sort

Bubble sort is a sorting algorithm that compares two adjacent elements and swaps them until they are in the intended order.

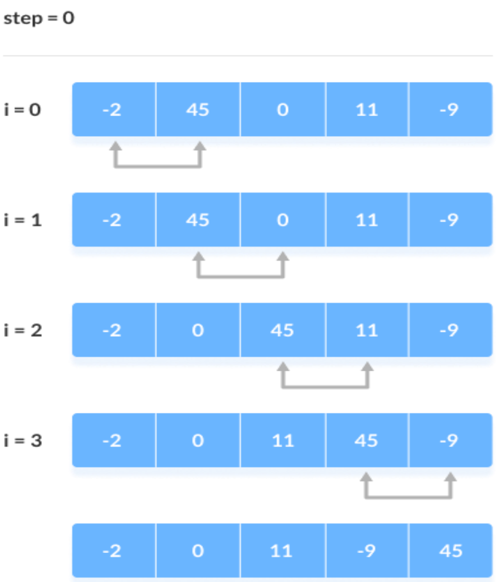
Just like the movement of air bubbles in the water that rise up to the surface, each element of the array move to the end in each iteration. Therefore, it is called a bubble sort.

Working of Bubble Sort

Suppose we are trying to sort the elements in **ascending order**.

1. First Iteration (Compare and Swap)

1. Starting from the first index, compare the first and the second elements.
2. If the first element is greater than the second element, they are swapped.
3. Now, compare the second and the third elements. Swap them if they are not in order.
4. The above process goes on until the last element.

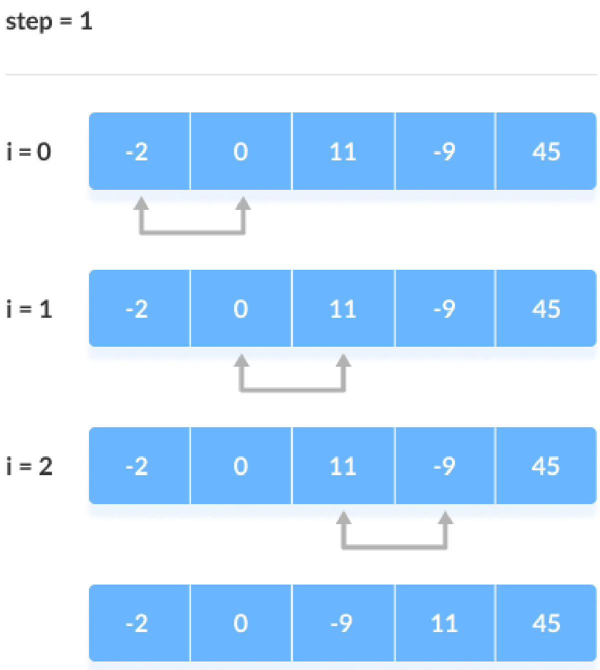


Compare the Adjacent Elements

2. Remaining Iteration

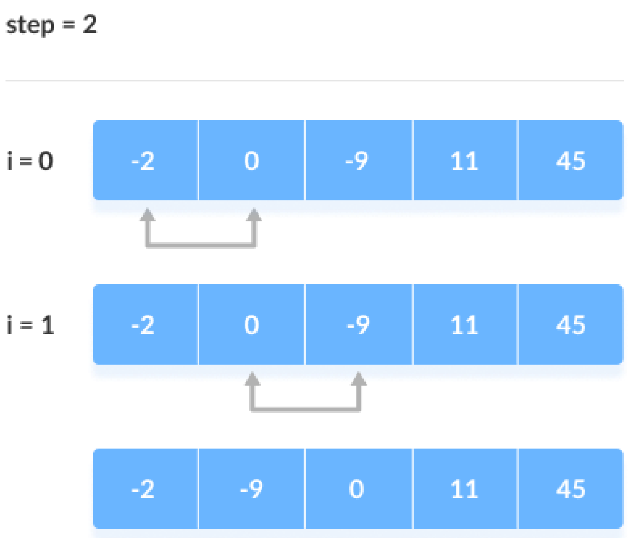
The same process goes on for the remaining iterations.

After each iteration, the largest element among the unsorted elements is placed at the end.



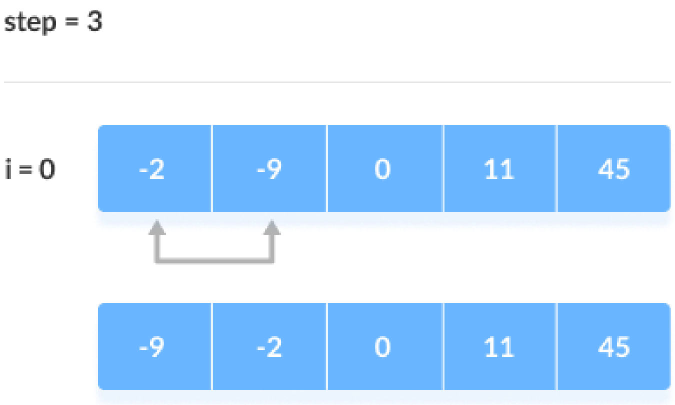
Put the largest element at the end

In each iteration, the comparison takes place up to the last unsorted element.



Compare the adjacent elements

The array is sorted when all the unsorted elements are placed at their correct positions.



The array is sorted if all elements are kept in the right order

Bubble Sort Algorithm

```
bubbleSort(array)
  for i <- 1 to sizeOfArray - 1
    for j <- 1 to sizeOfArray - 1 - i
      if leftElement > rightElement
        swap leftElement and rightElement
end bubbleSort
```

```
// Bubble sort in C++
#include <iostream>
using namespace std;

// perform bubble sort
void bubbleSort(int array[], int size) {

  // loop to access each array element
  for (int step = 0; step < size -1; ++step) {

    // loop to compare array elements
```

```
    for (int i = 0; i < size - step - 1; ++i) {

      // compare two adjacent elements
      // change > to < to sort in descending order
      if (array[i] > array[i + 1]) {

        // swapping elements if elements
        // are not in the intended order
        int temp = array[i];
        array[i] = array[i + 1];
        array[i + 1] = temp;
      }
    }
  }
}

// print array
void printArray(int array[], int size) {
  for (int i = 0; i < size; ++i) {
    cout << "  " << array[i];
  }
  cout << "\n";
}

int main() {
  int data[] = {-2, 45, 0, 11, -9};

  // find array's length
  int size = sizeof(data) / sizeof(data[0]);

  bubbleSort(data, size);

  cout << "Sorted Array in Ascending Order:\n";
  printArray(data, size);
```

}

Optimized Bubble Sort Algorithm

In the above algorithm, all the comparisons are made even if the array is already sorted.

This increases the execution time.

To solve this, we can introduce an extra variable `swapped`. The value of `swapped` is set true if there occurs swapping of elements. Otherwise, it is set **false**.

After an iteration, if there is no swapping, the value of `swapped` will be **false**. This means elements are already sorted and there is no need to perform further iterations.

This will reduce the execution time and helps to optimize the bubble sort.

Algorithm for optimized bubble sort is

```
bubbleSort(array)
  for i <- 1 to sizeOfArray - 1
    swapped <- false
    for j <- 1 to sizeOfArray - 1 - i
      if leftElement > rightElement
        swap leftElement and rightElement
        swapped <- true
    if swapped == false
      break
end bubbleSort
```

```
// Optimized bubble sort in C++
#include <iostream>
using namespace std;

// perform bubble sort
void bubbleSort(int array[], int size) {

  // loop to access each array element
  for (int step = 0; step < (size-1); ++step) {
```

```
    // check if swapping occurs
    int swapped = 0;

    // loop to compare two elements
    for (int i = 0; i < (size-step-1); ++i) {

      // compare two array elements
      // change > to < to sort in descending order
      if (array[i] > array[i + 1]) {

        // swapping occurs if elements
        // are not in intended order
        int temp = array[i];
        array[i] = array[i + 1];
        array[i + 1] = temp;

        swapped = 1;
      }
    }

    // no swapping means the array is already sorted
    // so no need of further comparison
    if (swapped == 0)
      break;
  }
}

// print an array
void printArray(int array[], int size) {
  for (int i = 0; i < size; ++i) {
    cout << "  " << array[i];
  }
  cout << "\n";
```

```
}

int main() {
  int data[] = {-2, 45, 0, 11, -9};

  // find the array's length
  int size = sizeof(data) / sizeof(data[0]);

  bubbleSort(data, size);

  cout << "Sorted Array in Ascending Order:\n";
  printArray(data, size);
}
```

Bubble Sort Complexity

Time Complexity	
Best	O(n)
Worst	$O(n^2)$
Average	$O(n^2)$
Space Complexity	O(1)
Stability	Yes

Complexity in Detail

Bubble Sort compares the adjacent elements.

Cycle	Number of Comparisons
1st	(n-1)
2nd	(n-2)
3rd	(n-3)

....	......
last	1

Hence, the number of comparisons is

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

nearly equals to n^2

Hence, **Complexity:** $O(n^2)$

Also, if we observe the code, bubble sort requires two loops. Hence, the complexity is $n*n = n^2$

1. Time Complexities

- **Worst Case Complexity:** $O(n^2)$

  If we want to sort in ascending order and the array is in descending order then the worst case occurs.

- **Best Case Complexity:** $O(n)$

  If the array is already sorted, then there is no need for sorting.

- **Average Case Complexity:** $O(n^2)$

  It occurs when the elements of the array are in jumbled order (neither ascending nor descending).

2. Space Complexity

- Space complexity is $O(1)$ because an extra variable is used for swapping.
- In the **optimized bubble sort algorithm**, two extra variables are used. Hence, the space complexity will be $O(2)$.

Bubble Sort Applications

Bubble sort is used if

- complexity does not matter
- short and simple code is preferred