Sorting Algorithm

Bubble sort

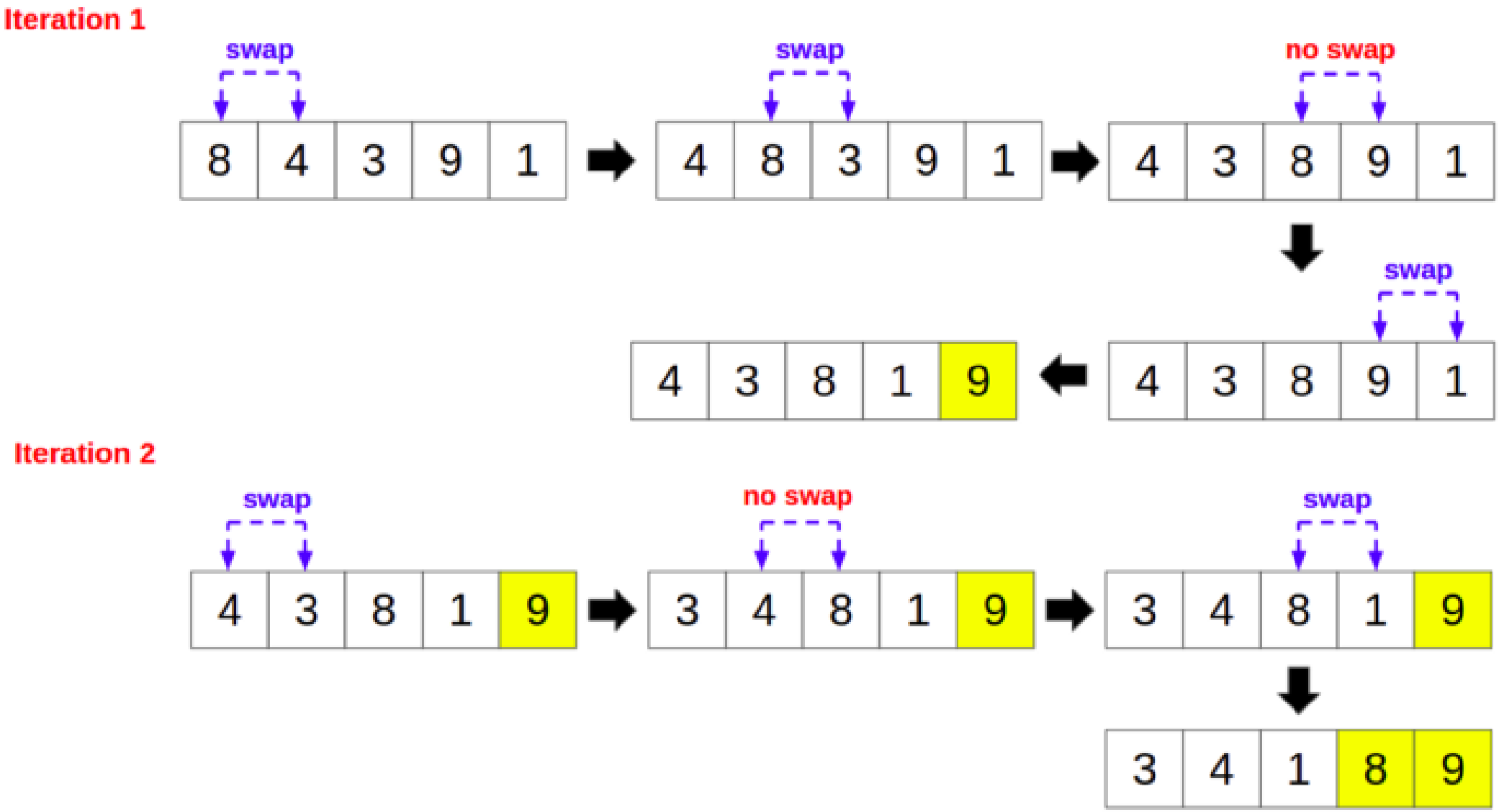
Insertion sort

Shoaib Rauf

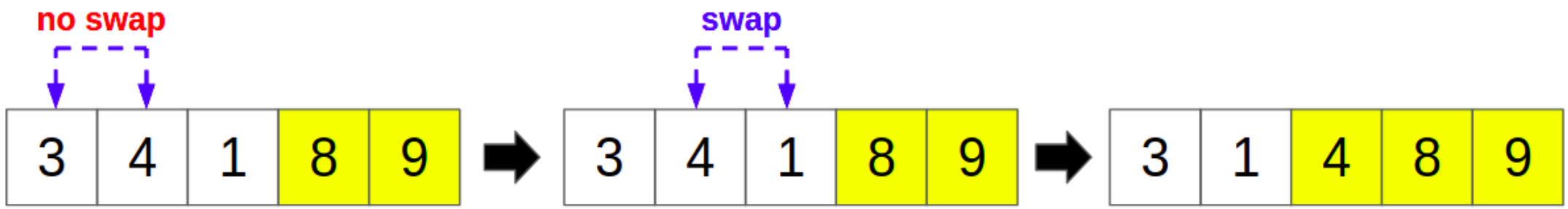
Bubble Sort

Algorithm 1: Bubble sort

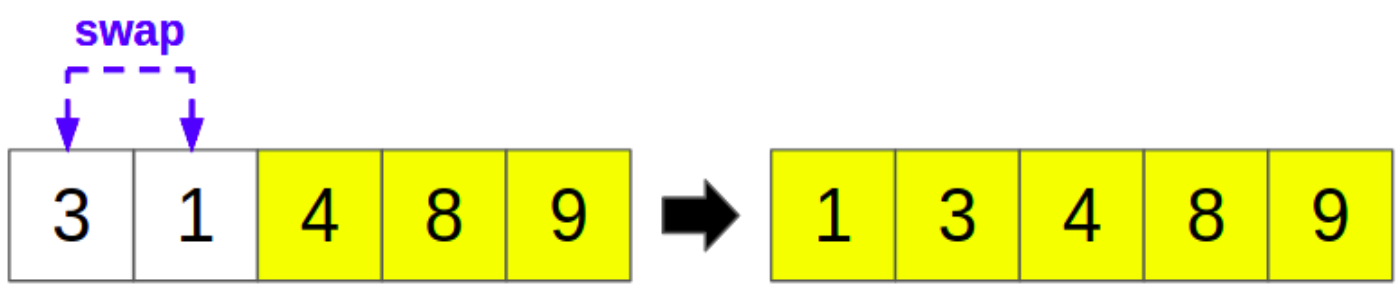
```
Data: Input array A[]
Result: Sorted A[]
int i, j, k;
N = length(A);
for j = 1 to N do
    for i = 0 to N-1 do
        if A[i] > A[i+1] then
            temp = A[i];
            A[i] = A[i+1];
            A[i+1] = temp;
        end
    end
end
```


Iteration 1
swap
8 4 3 9 1
swap
4 8 3 9 1
no swap
4 3 8 9 1
swap
4 3 8 9 1
4 3 8 1 9
Iteration 2
swap
4 3 8 1 9
no swap
3 4 8 1 9
swap
3 4 8 1 9
3 4 1 8 9

Iteration 3



Iteration 4



Time Complexity (Bubble Sort)

- In traditional bubble sort algorithm:
  - The Best case and Worst case scenario, time complexity is $O(n)^2$.
  - No flag variable is used to in the outer loop to determine for no of swaps

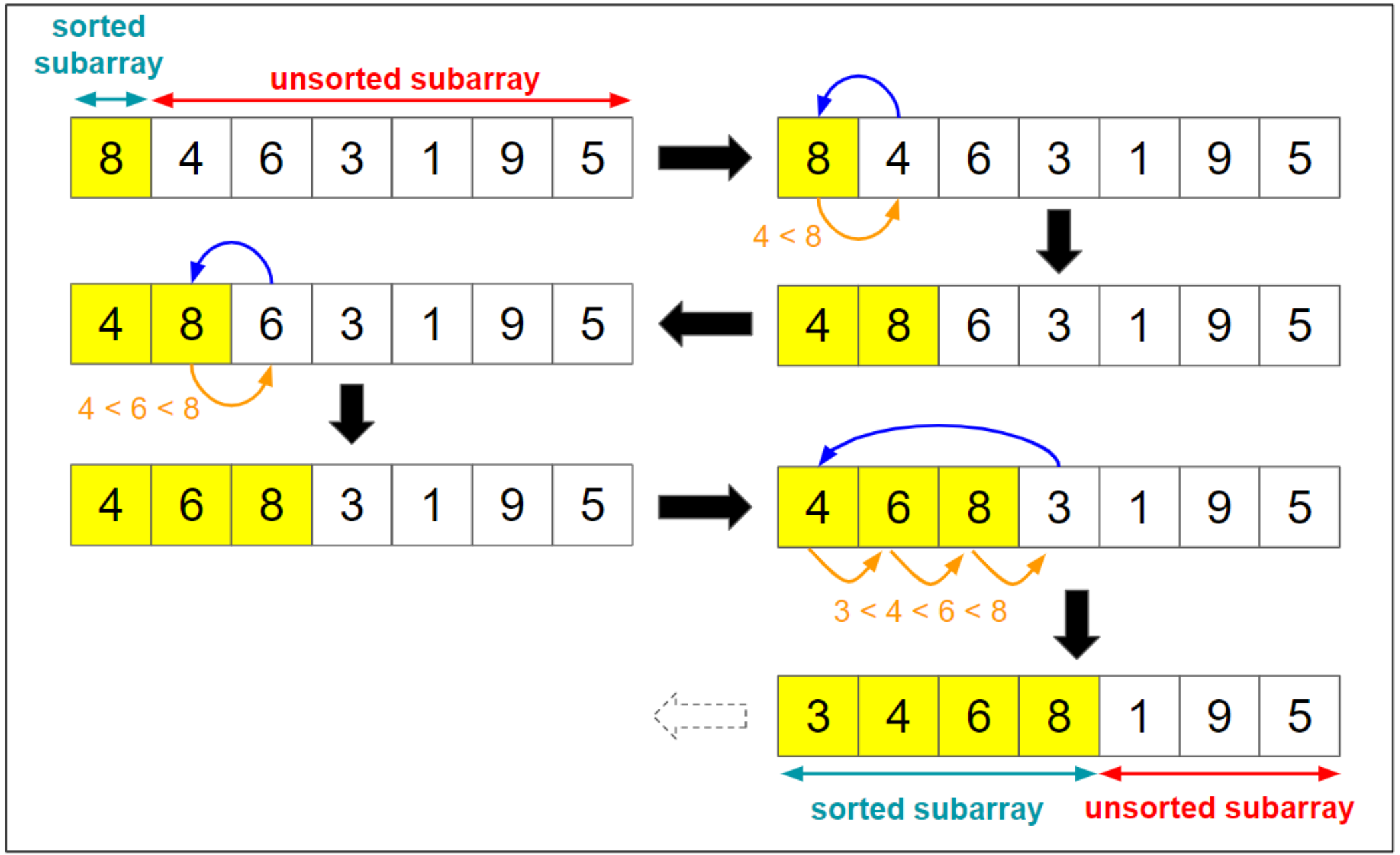
- In an optimized bubble sort algorithm:
  - If the numbers are already sorted in ascending order, the algorithm will determine in the first iteration that no number pairs need to be swapped (flag variable is used)and will then terminate immediately.

  - Best case Scenario: O(n)
  - Worst case Scenario: $O(n)^2$

Insertion Sort

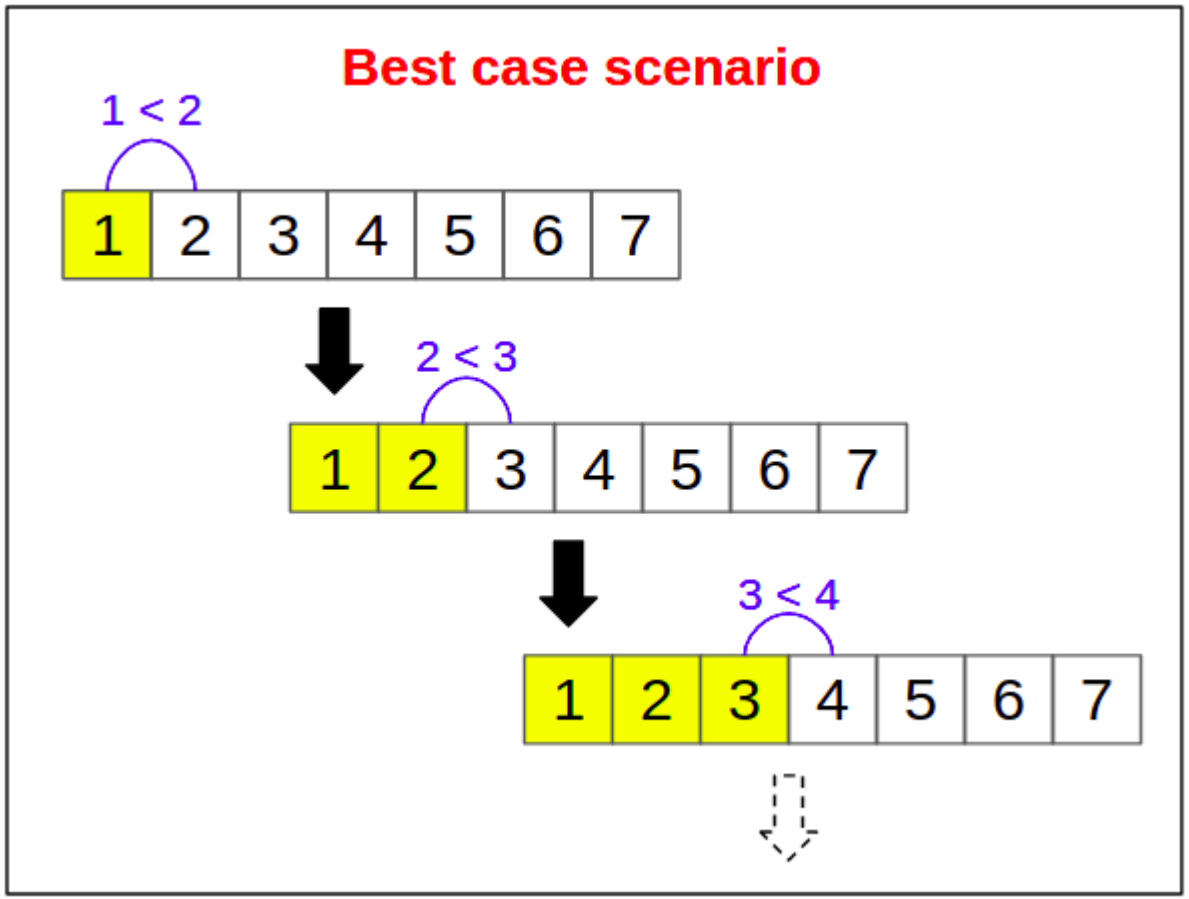
ALGORITHM *InsertionSort*$(A[0..n-1])$

```
//Sorts a given array by insertion sort
//Input: An array A[0..n − 1] of n orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
for i ← 1 to n − 1 do
    v ← A[i]
    j ← i − 1
    while j ≥ 0 and A[j] > v do
        A[j + 1] ← A[j]
        j ← j − 1
    A[j + 1] ← v
```


sorted subarray
unsorted subarray
8 4 6 3 1 9 5
4 < 8
4 8 6 3 1 9 5
4 < 6 < 8
4 6 8 3 1 9 5
3 < 4 < 6 < 8
3 4 6 8 1 9 5
sorted subarray
unsorted subarray

Time Complexity (Insertion Sort)

- The **best-case** time complexity of insertion sort is **O(n)**. When the array is already sorted (which is the best case), insertion sort has to perform only one comparison in each iteration



Time Complexity (Insertion Sort)

- The **worst-case complexity is $O(n^2)$**. When the array is sorted in reverse order (which is the worst case), we have to perform *i* number of comparisons in the i^{th} iteration