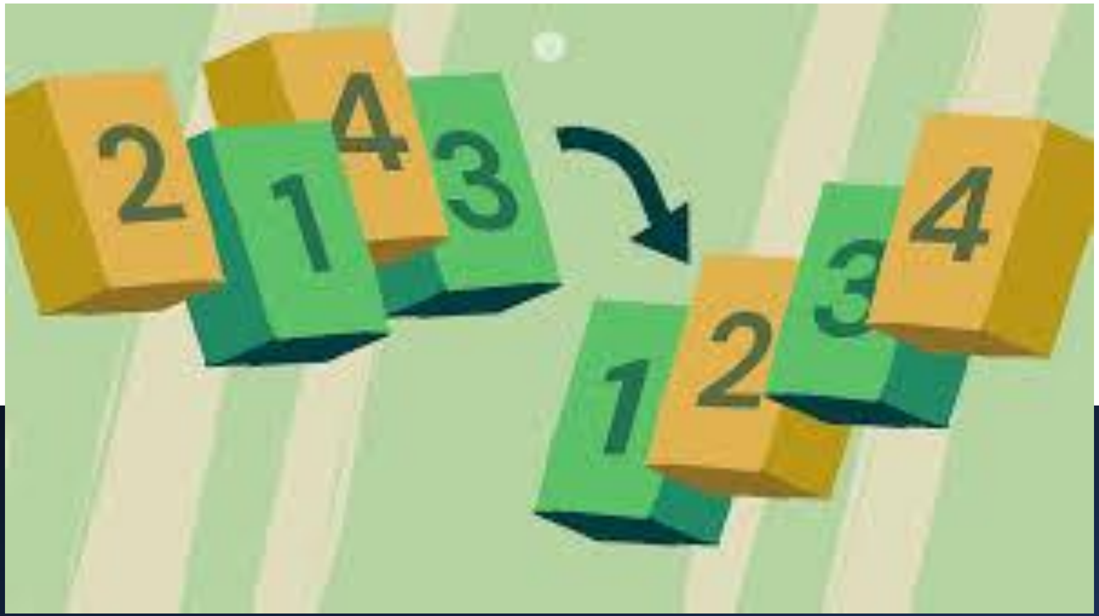
Algorithms

Lecture 7
Sorting Algorithms



A. S. M. Sanwar Hosen

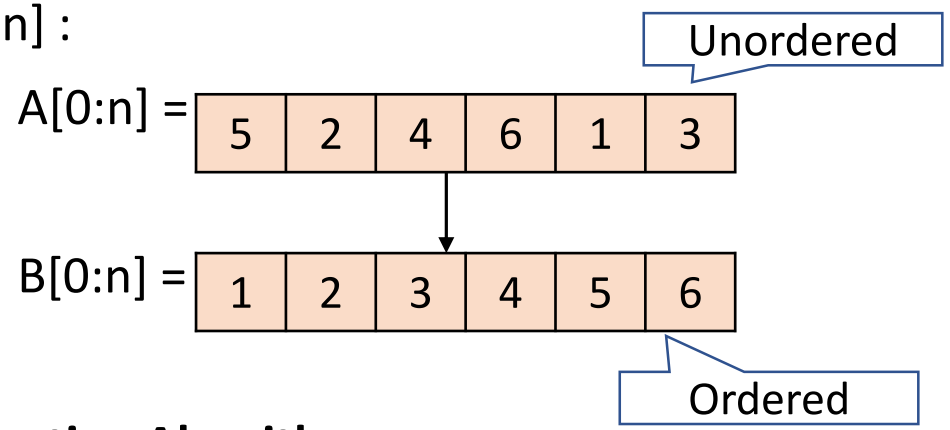
Email: sanwar@wsu.ac.kr

Date: 4 May, 2023

Sorting Algorithms

- **Sorting Algorithm:** It arranges elements or numbers in an array in increasing or decreasing order. For example,

A[0:n] → B[0:n] :



- **Types of Sorting Algorithms**

Different types of sorting algorithms are:

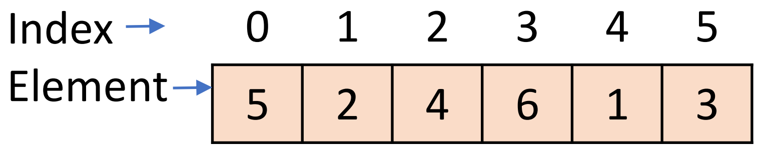
- ✓ Bubble Sort
- ✓ Insertion Sort
- ✓ Merge Sort
- ✓ Selection Sort
- ✓ Heap Sort

What is an array?: An array is a data structure that contains a set of elements.

Properties/components of an array:

1) Data structure, i.e., list or table, array
2) Index
3) Elements

Example:



Bubble Sort (1)

- ❑ **Bubble Sort:** It is a sorting algorithm that compares two adjacent elements and swaps them until they are in the order.
- ❑ **How It Works?**

Insertion sort works as follows:

- ✓ Starting from the first index, compare the first and the second elements either less than or greater than depending on the order (ascending or descending).
- ✓ If the first element is greater than the second element, they are swapped.
- ✓ Now, compare the second and the third elements. Swap them if they are not in order.
- ✓ The process is continued until the last element in the array.

- ❑ **Time Complexity:** The time complexity of bubble sort are:
- ✓ Best case: O(n)
- ✓ Worst case: O(n^2)

Bubble Sort (2)

❑ Bubble Sort

Let's look at an example, suppose the array to be sorted is A[0:n] → B[0:n], A[0:n] = {5, 2, 4, 6, 1, 3}

Step 0

Iteration 0:
Compare the adjacent elements in index **[0]** and **[1]** and swap

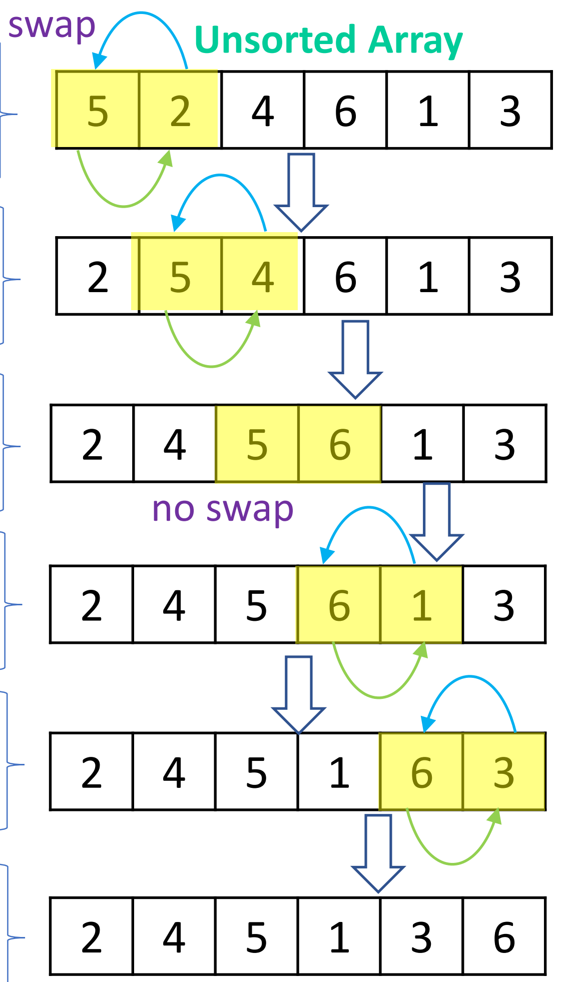
Iteration 1:
Compare the adjacent elements in index **[1]** and **[2]** and swap

Iteration 2:
Compare the adjacent elements in index **[2]** and **[3]** and swap

Iteration 3:
Compare the adjacent elements in index **[3]** and **[4]** and swap

Iteration 4:
Compare the adjacent elements in index **[4]** and **[5]** and swap

Output: array of Step 0



Step 1

Iteration 0:
Compare the adjacent elements in index **[0]** and **[1]** and swap

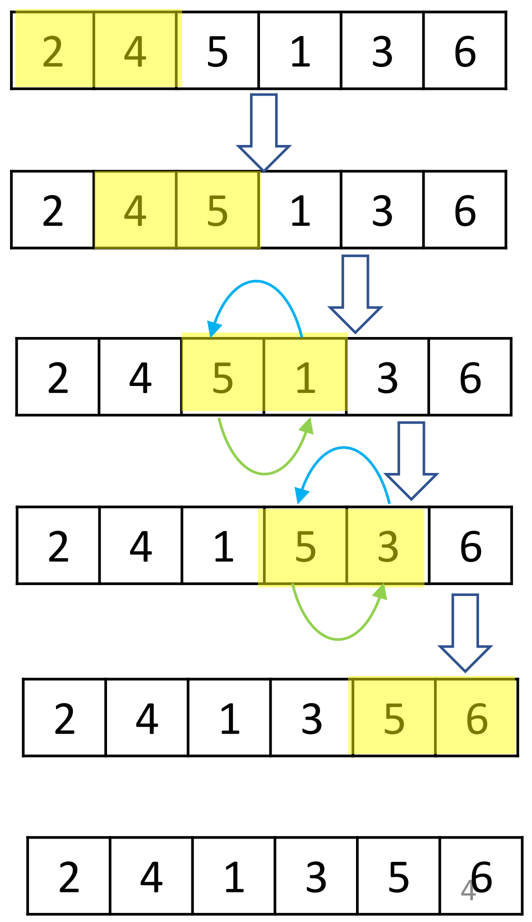
Iteration 1:
Compare the adjacent elements in index **[1]** and **[2]** and swap

Iteration 2:
Compare the adjacent elements in index **[2]** and **[3]** and swap

Iteration 3:
Compare the adjacent elements in index **[3]** and **[4]** and swap

Iteration 4:
Compare the adjacent elements in index **[4]** and **[5]** and swap

Output: array of Step 1



Bubble Sort (3)

❑ Bubble Sort

Let’s look at an example, suppose the array to be sorted is A[0:n] → B[0:n], A[0:n] = {5, 2, 4, 6, 1, 3}

Step 2

Iteration 0:
Compare the adjacent elements in index **[0]** and **[1]** and swap

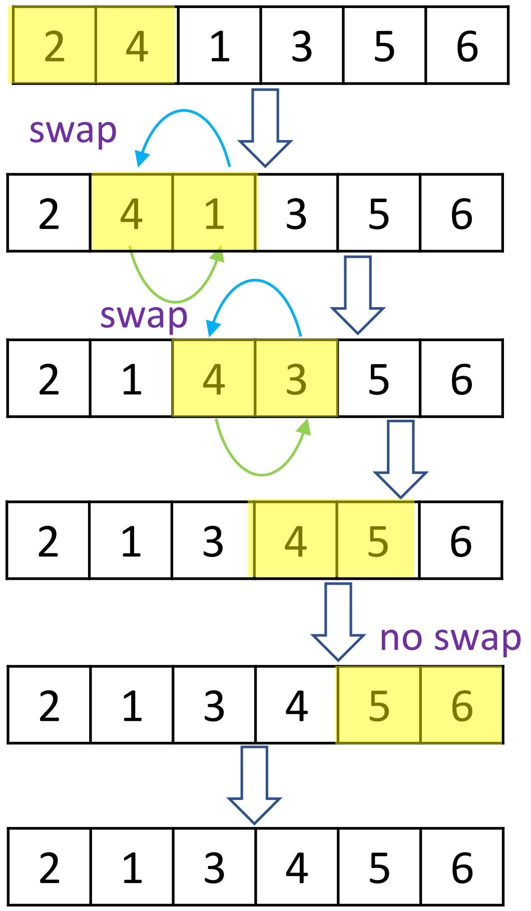
Iteration 1:
Compare the adjacent elements in index **[1]** and **[2]** and swap

Iteration 2:
Compare the adjacent elements in index **[2]** and **[3]** and swap

Iteration 3:
Compare the adjacent elements in index **[3]** and **[4]** and swap

Iteration 4:
Compare the adjacent elements in index **[4]** and **[5]** and swap

Output: array of Step 2



Step 3

Iteration 0:
Compare the adjacent elements in index **[0]** and **[1]** and swap

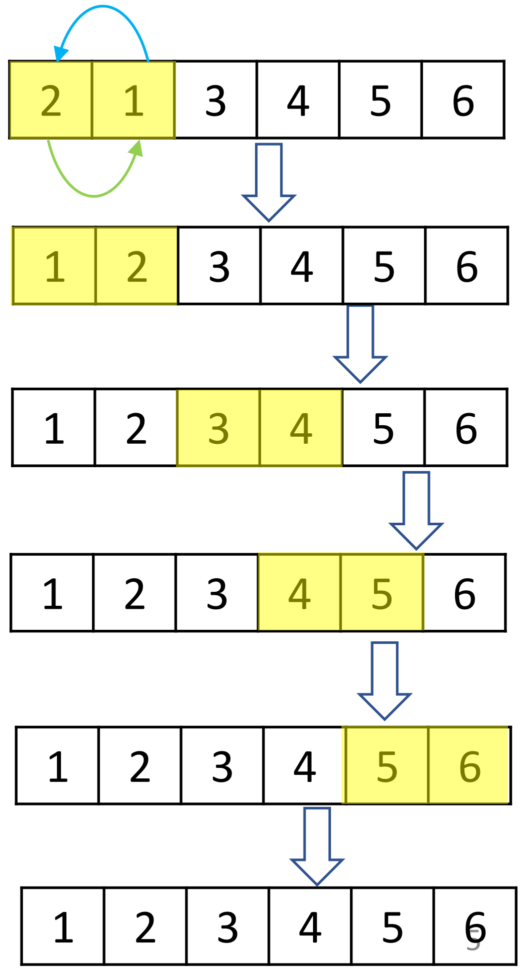
Iteration 1:
Compare the adjacent elements in index **[1]** and **[2]** and swap

Iteration 2:
Compare the adjacent elements in index **[2]** and **[3]** and swap

Iteration 3:
Compare the adjacent elements in index **[3]** and **[4]** and swap

Iteration 4:
Compare the adjacent elements in index **[4]** and **[5]** and swap

Output: sorted array



Insertion Sort (1)

- ❑ **Insertion Sort:** It places an unsorted element at its right position (in sorted array) by comparing with the elements in already sorted array in each iteration.
- ❑ **How It Works?**

Insertion sort works as follows:

- ✓ Divide the array into two subarrays: sorted and unsorted subarrays.
- ✓ Select the second element as a key and compare with the element of sorted subarray. If it is less than the element, swaps the elements.
- ✓ Move the key to the next element (towards right side) of the unsorted array and compare with the element in the sorted array, swaps the elements until the first element of the sorted array.
- ✓ Iterate or repeat the process until the elements are sorted in the array.

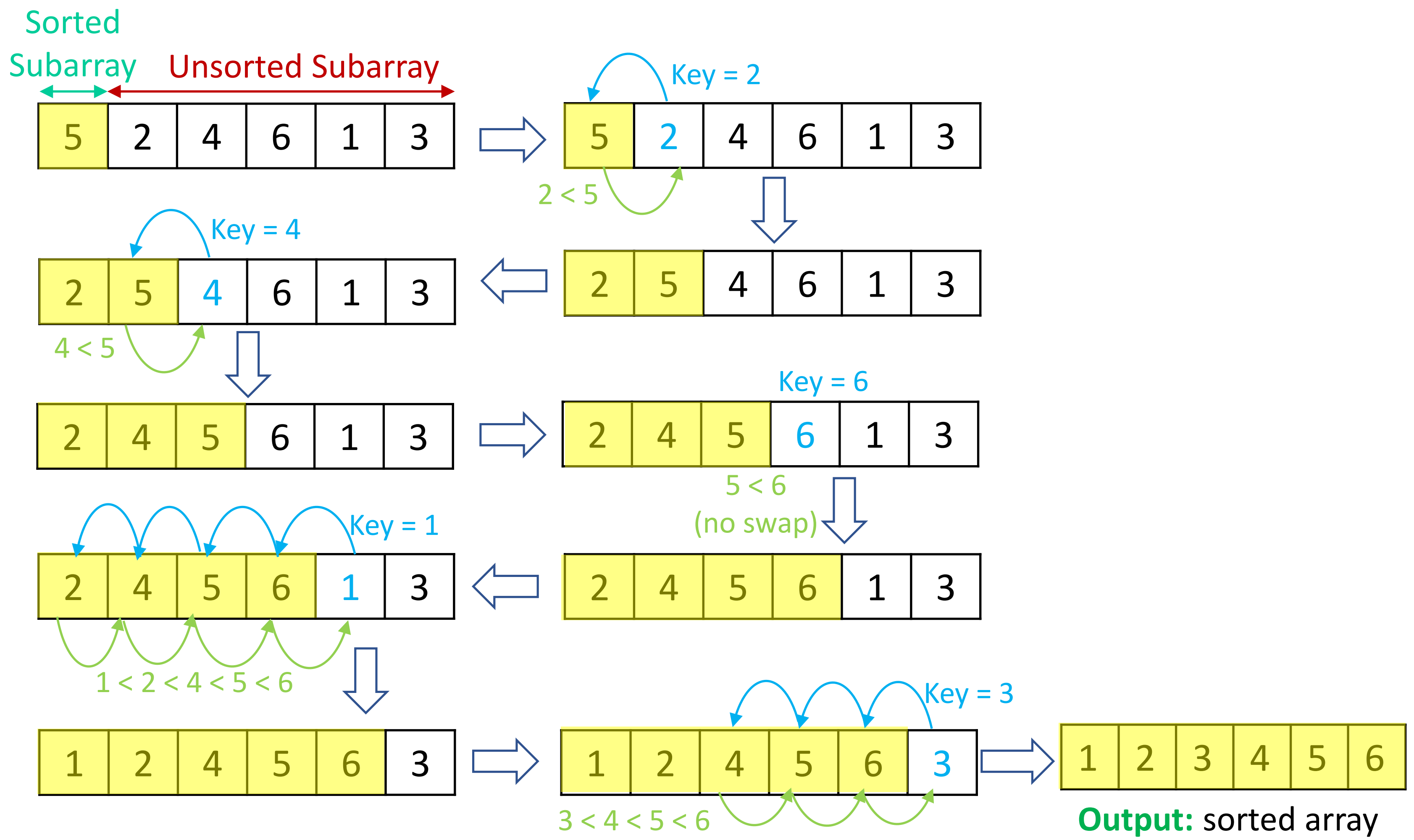
- ❑ **Time Complexity:** The time complexity of insertion sort are:

- ✓ Best case: O(n)
- ✓ Worst case: O(n^2)
- ✓ Average case: O(n^2)

Insertion Sort (2)

❑ Insertion Sort

Let’s look at an example, suppose the array to be sorted is A[0:n] → B[0:n], A[0:n] = {5, 2, 4, 6, 1, 3}



Merge Sort (1)

- **Merge Sort:** It divides a problem into multiple subproblems, then each subproblem is solved individually, and finally the solutions are combined to from the final solution. It follows the divide and conquer strategy in solving the problem.
- **How It Works?**

Merge sort works as follows:

- ✓ **Divide:** It splits the array into two subarrays at the half way point. For example, if m is the half-way point between l and r, it splits the array into $A[l, \dots, m]$ and $A[m+1, \dots, r]$.
- ✓ **Conquer:** In conquer, it sorts the both subarrays. If the conquer step does not reach in the base case, it again divides the both subarrays and sort them.
- ✓ **Combine:** Once the conquer step reaches the base case and produces two sorted subarrays, it combines the results by creating a sorted array $A[l, \dots, r]$.

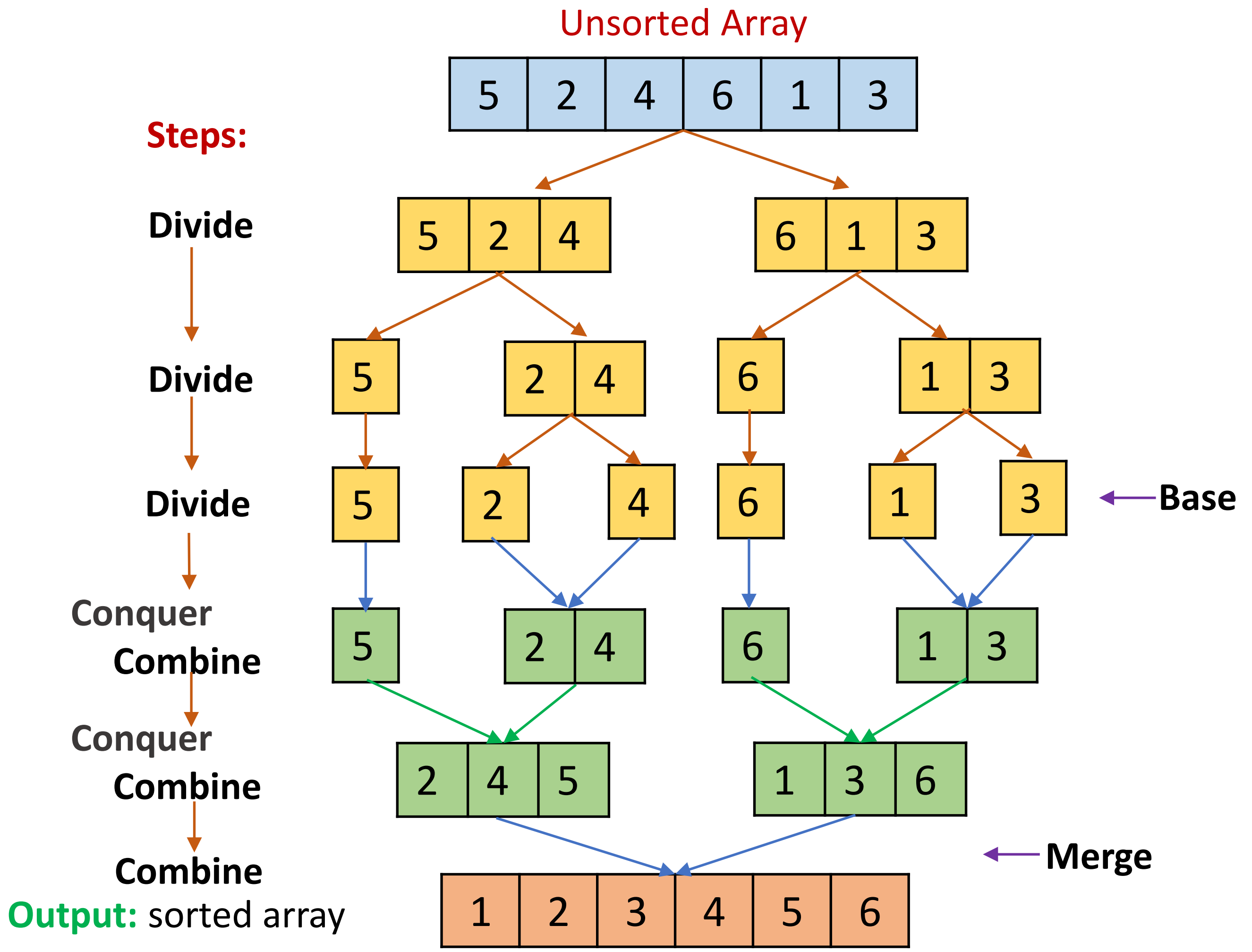
- **Time complexity:** The time complexity of merge sort are:

- ✓ Best case: O(n log n)
- ✓ Worst case: O(n log n)
- ✓ Average case: O(n log n)

Merge Sort (2)

❑ Merge Sort

Let's look at an example, suppose the array to be sorted is A[0:n] → B[0:n], A[0:n] = {5, 2, 4, 6, 1, 3}



Applications of Sorting Algorithms

- **Applications:** Sorting algorithms are essential in a broad variety of applications:
- ✓ Sort a list of names or numbers
- ✓ Data compression
- ✓ Computer graphics
- ✓ Find the median
- ✓ Supply chain management
- ✓ Computational biology
- ✓ Find duplicates in a mailing list
- ✓ Binary search in a database
- ✓ Find the closest pair
- ✓ Display google PageRank list

Sorting Algorithms Implementation in Python (1)

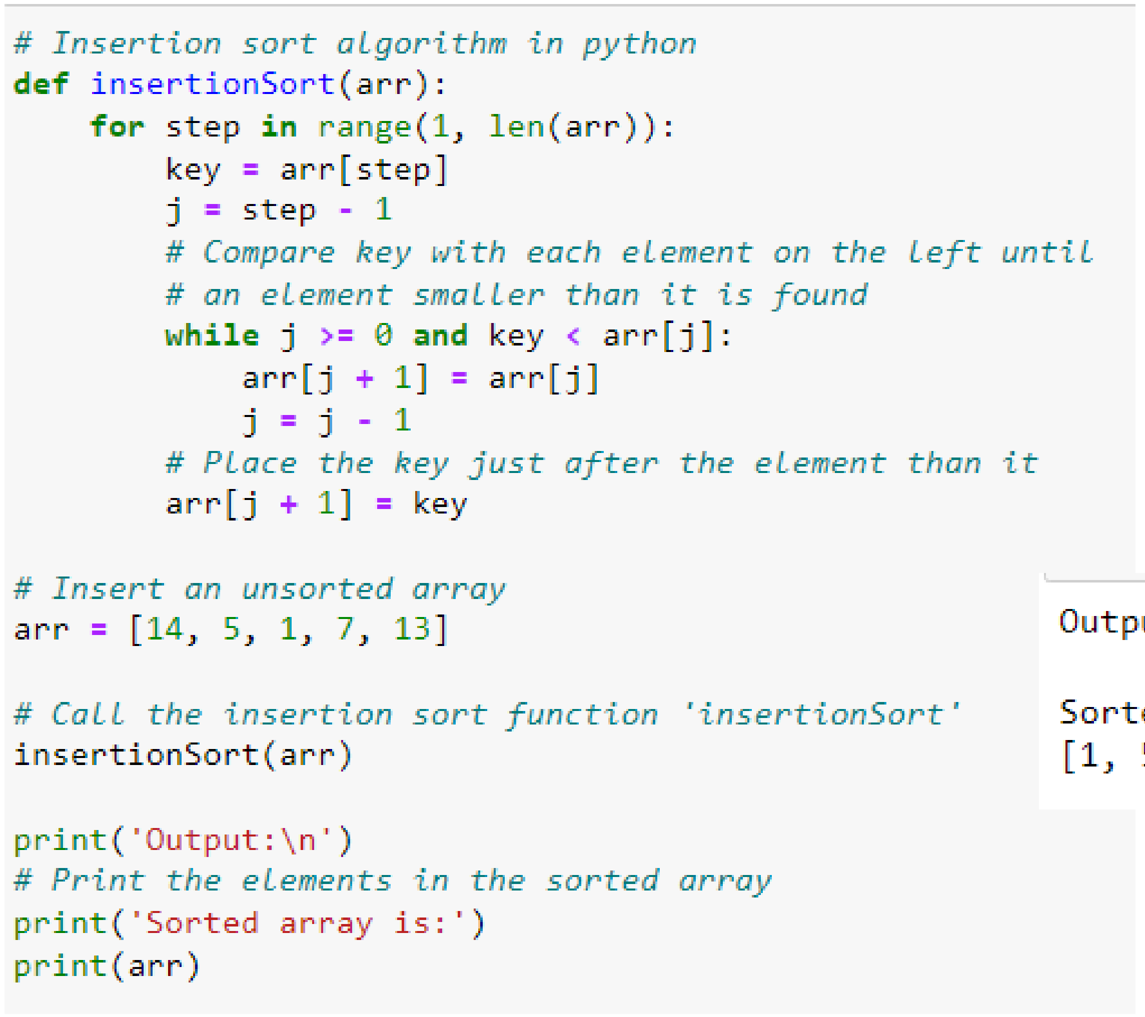
❑ Bubble Sort Algorithm in Python:

```python
# An example of a bubble sort algorithm in python:
def bubbleSort(arr):
    n = len(arr) # define the length of the array
    # If the array is already sorted,no need to go to the entire process
    swapped = False
    # Search all the elements in the array
    for i in range(n-1):
        for j in range(0, n-i-1):
            # If the elements fond is greater, then swap
            if arr[j] > arr[j + 1]:
                swapped = True
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
        if not swapped:
            # if we dont need to make a single swap, exit the Loop
            return

# Insert an unsorted array
arr = [14, 13, 5, 7, 11, 12, 30]

# Call the bubble sort function 'bubbleSort'
bubbleSort(arr)

# Print the elements of the sorted array
print("Output:\n")
print("Sorted array is:")
for i in range(len(arr)):
    print("% d" % arr[i], end=" ")
```

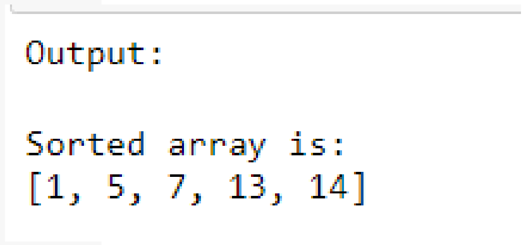
```
Output:

Sorted array is:
 5  7  11  12  13  14  30
```

Sorting Algorithms Implementation in Python (2)

- **Insertion Sort Algorithm in Python:**



```python
# Insertion sort algorithm in python
def insertionSort(arr):
    for step in range(1, len(arr)):
        key = arr[step]
        j = step - 1
        # Compare key with each element on the left until
        # an element smaller than it is found
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j = j - 1
        # Place the key just after the element than it
        arr[j + 1] = key

# Insert an unsorted array
arr = [14, 5, 1, 7, 13]

# Call the insertion sort function 'insertionSort'
insertionSort(arr)

print('Output:\n')
# Print the elements in the sorted array
print('Sorted array is:')
print(arr)
```



```
Output:

Sorted array is:
[1, 5, 7, 13, 14]
```

Sorting Algorithms Implementation in Python (3)

Merge Sort Algorithm in Python:

```
# Merge sort algorithm in python
# Merge two subarrays of arr[l,...,r].
# First subarray: [l..m]
# Second subarray: [m+1..r]
def merge(arr, l, m, r):
    n1 = m - l + 1
    n2 = r - m
    # Define temporary arrays
    L = [0] * (n1)
    R = [0] * (n2)
    # Copy data to temporary arrays L[] and R[]
    for i in range(0, n1):
        L[i] = arr[l + i]
    for j in range(0, n2):
        R[j] = arr[m + 1 + j]
    # Merge the temporary arrays back into arr[l..r]
    i = 0  # Initial index of first subarray
    j = 0  # Initial index of second subarray
    k = l  # Initial index of merged subarray
    while i < n1 and j < n2:
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1
    # Copy the remaining elements of L[], if there are any
    while i < n1:
        arr[k] = L[i]
        i += 1
        k += 1
    # Copy the remaining elements of R[], if there are any
    while j < n2:
        arr[k] = R[j]
        j += 1
        k += 1
# l is for left index and r is right index of the subarrays
```

```
# Define the function of merge sort
def mergeSort(arr, l, r):
    if l < r:
        # Same as (l+r)//2, but avoids overflow for
        # large l and h
        m = l+(r-l)//2
        # Sort first and second halves
        mergeSort(arr, l, m)
        mergeSort(arr, m+1, r)
        merge(arr, l, m, r)

# Insert an unsorted array
arr = [12, 11, 13, 5, 6, 7]
n = len(arr)

# Call the merge sort function 'mergeSort'
mergeSort(arr, 0, n-1)

# Print the element in the sorted array
print("Output:\n")
print("Sorted array is:\n")
for i in range(n):
    print("%d" % arr[i],end=" ")
```

```
Output:

Sorted array is:

5 6 7 11 12 13
```