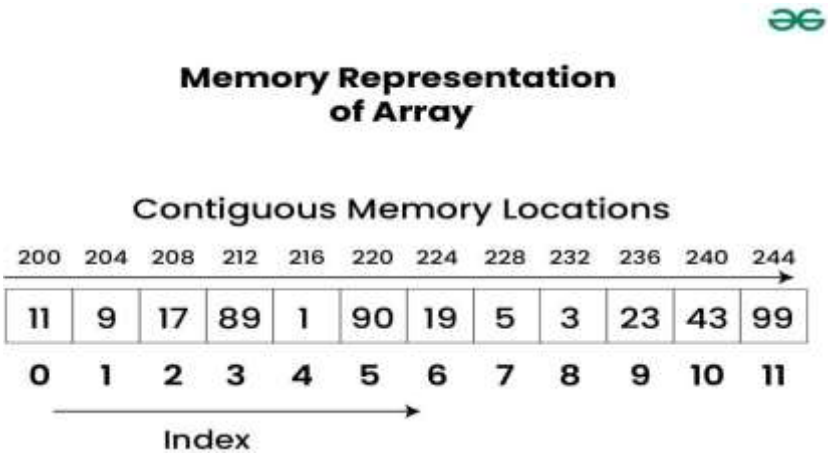
Array is a collection of items of the same variable type that are stored at contiguous memory locations. It is one of the most popular and simple data structures used in programming.

Basic terminologies of Array

- **Array Index:** In an array, elements are identified by their indexes. Array index starts from 0.
- **Array element:** Elements are items stored in an array and can be accessed by their index.
- **Array Length:** The length of an array is determined by the number of elements it can contain.

Memory representation of Array

In an array, all the elements are stored in contiguous memory locations. So, if we initialize an array, the elements will be allocated sequentially in memory. This allows for efficient access and manipulation of elements.



Declaration of Array

Arrays can be declared in various ways in different languages. For better illustration, below are some language-specific array declarations:

```
# In Python, all types of lists are created same way
arr = []
```

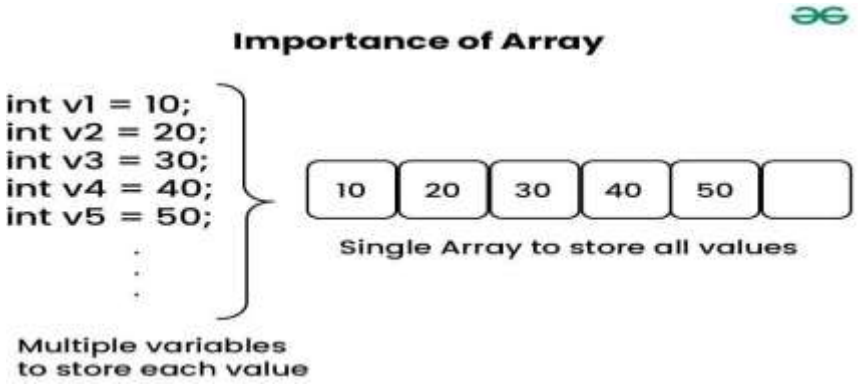
Initialization of Array

Arrays can be initialized in different ways in different languages.

```
# This list will store integer type elements
arr = [1, 2, 3, 4, 5]

# This list will store character type elements (strings in Python)
arr = ['a', 'b', 'c', 'd', 'e']

# This list will store float type elements
arr = [1.4, 2.0, 24.0, 5.0, 0.0]  # All float values
```

Importance of Array

Assume there is a class of five students and if we have to keep records of their marks in examination then, we can do this by declaring five variables individual and keeping track of records but what if the number of students becomes very large, it would be challenging to manipulate and maintain the data.
What it means is that, we can use normal variables (v1, v2, v3, ..) when we have a small number of objects. But if we want to store a large number of instances, it becomes difficult to manage them with normal variables. **The idea of an array is to represent many instances in one variable**.



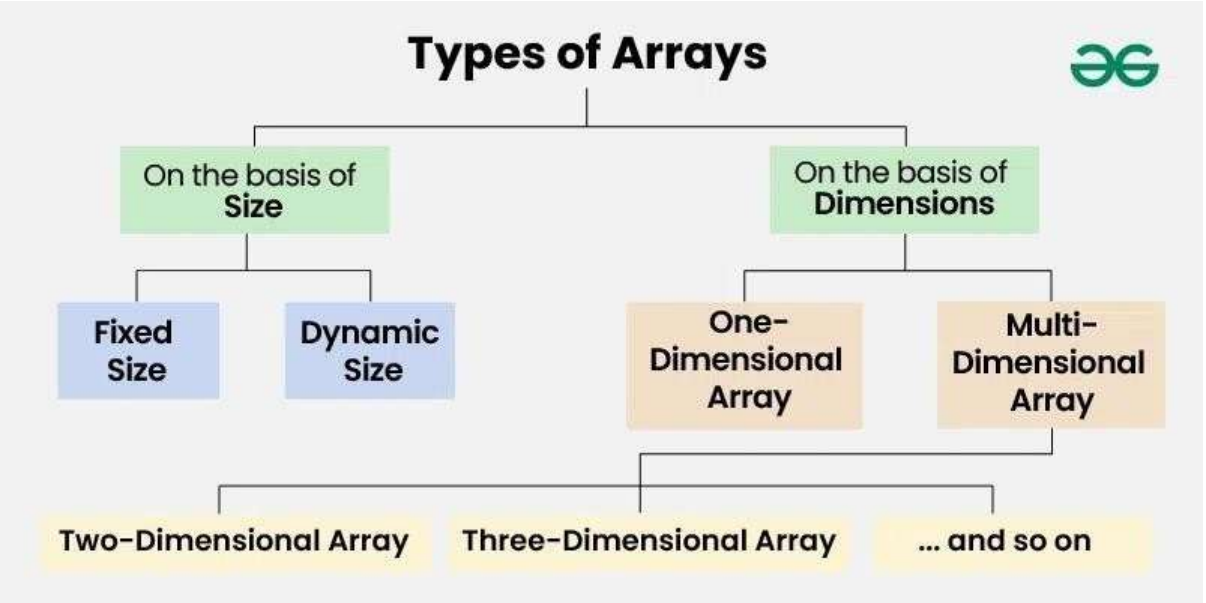
Need or Applications of Array Data Structures

- Array is a fundamental data structure and many other data structure are implemented using this. Implementing data structures such as stacks and queues
- Representing data in tables and matrices
- Creating dynamic data structures such as Hash Tables and Graph.
- When compared to other data structures, arrays have the advantages like random access (we can quickly access i-th item) and cache friendliness (all items are stored at contiguous location)

Types of Arrays

Arrays can be classified in two ways:

- On the basis of Size
- On the basis of Dimensions



Types of Arrays on the basis of Size:

1. Fixed Sized Arrays:

We cannot alter or update the size of this array. Here only a fixed size (i,e. the size that is mentioned in square brackets **[]**) of memory will be allocated for storage. In case, we don't know the size of the array then if we declare a larger size and store a lesser number of elements will result in a wastage of memory or we declare a lesser size than the number of elements then we won't get enough memory to store all the elements. In such cases, static memory allocation is not preferred.

```
# Create a fixed-size list of length 5,
# initialized with zeros
arr = [0] * 5

# Output the fixed-size list
print(arr)
```

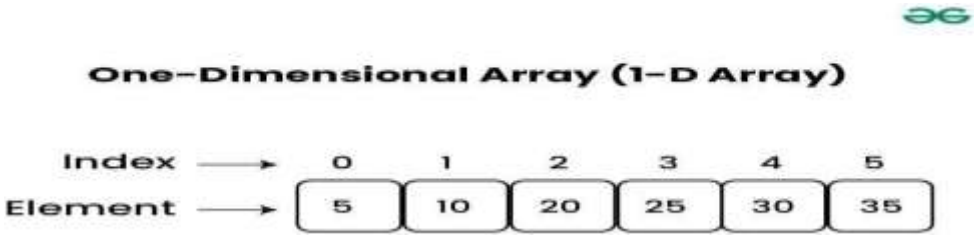
Dynamic Sized Arrays:

The size of the array changes as per user requirements during execution of code so the coders do not have to worry about sizes. They can add and removed the elements as per the need. The memory is mostly dynamically allocated and de-allocated in these arrays.

```
# Dynamic Array
arr = []
```

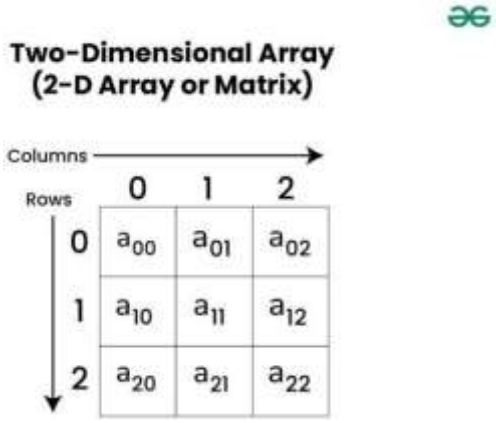
Types of Arrays on the basis of Dimensions:

1. **One-dimensional Array(1-D Array):** You can imagine a 1d array as a row, where elements are stored one after another.



Multi-dimensional Array: A multi-dimensional array is an array with more than one dimension. We can use multidimensional array to store complex data in the form of tables, etc. We can have 2-D arrays, 3-D arrays, 4-D arrays and so on.

- **Two-Dimensional Array(2-D Array or Matrix):** 2-D Multidimensional arrays can be considered as an array of arrays or as a matrix consisting of rows and columns.



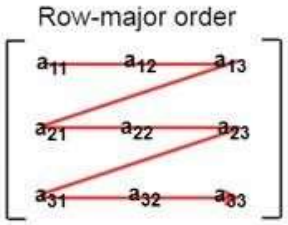
There are two main techniques of storing 2D array elements into memory

1. Row major ordering

In row major ordering, all the rows of the 2D array are stored into the memory contiguously. Considering the array shown in the above image, its memory allocation according to row major order is.

(0,0)	(0,1)	(0,2)	(1,0)	(1,1)	(1,2)	(2,0)	(2,1)	(2,2)

First, the 1st row of the array is stored into the memory completely, then the 2nd row of the array is stored into the memory completely and so on till the last row.

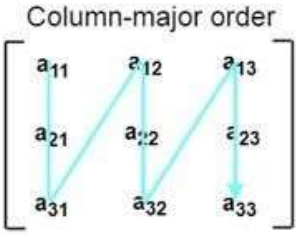


2. Column major ordering

According to the column major ordering, all the columns of the 2D array are stored into the memory contiguously. The memory allocation of the array is.

(0,0)	(1,0)	(2,0)	(0,1)	(1,1)	(2,1)	(0,2)	(1,2)	(2,2)

First, the 1st column of the array is stored into the memory completely, and then the 2nd row of the array is stored into the memory completely and so on till the last column of the array.



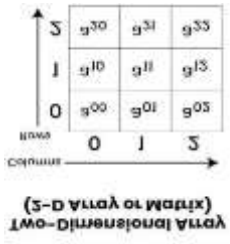
To find the address of the element using row-major order uses the following formula:

Address of A[I][J] = Base Address + (I * col dimn +(J)) * sizeof(datatype)

To find the address of the element using column-major order uses the following formula:

Address of A[I][J] = Base Address + (I + row dimn * (J)) * sizeof(datatype)

- **Three-Dimensional Array(3-D Array):** A 3-D Multidimensional array contains three dimensions, so it can be considered an array of two-dimensional arrays.



Operations on Array

1. Array Traversal:

Array traversal involves visiting all the elements of the array once. Below is the implementation of Array traversal in different Languages:

```
# This list will store integer type elements
arr = [1, 2, 3, 4, 5]

# Traversing over arr
for i in range(len(arr)):
    print(arr[i], end=" ")
```

2. Insertion in Array:

We can insert one or multiple elements at any position in the array. Below is the implementation of Insertion in Array in different languages:

```
# Example usage

arr = [1, 2, 3, 4, 5]

x = 10  # Element to be inserted

pos = 2  # Position to insert the element

arr.insert(pos, x)

# Print the updated list

print("Updated List:", arr)
```

3. Deletion in Array:

We can delete an element at any index in an array. Below is the implementation of Deletion of element in an array:

```
# Initialize a list

arr = [10, 20, 30, 40, 50]

# Value to delete

key = 40

# Remove the element with the specified value

# if present in the list

if key in arr:
```

```
    arr.remove(key)
else:
    print("Element Not Found")
# Output the modified list
print(arr)  # Output: [10, 20, 30, 50]
```

4. Searching in Array:
We can traverse over an array and search for an element. Below is the implementation of Searching of element in an array:

```
# Function to implement search operation
def find_element(arr, n, key):
    for i in range(n):
        if arr[i] == key:
            return i
    return -1
```

Complexity Analysis of Operations on Array

Operation	Time Complexity	Space Complexity
Traversal	O(n)	O(1)
Linear Search	O(n)	O(1)
Binary Search	O(log n)	O(1)
Insertion at End	O(1)	O(1)
Insertion at Beginning/Middle	O(n)	O(1)
Deletion at End	O(1)	O(1)
Deletion at Beginning/Middle	O(n)	O(1)

- **Traversal**: Traversing an array involves visiting every element once to perform an operation (e.g., printing all values or finding the maximum). The time taken increases linearly with the number of elements, so the time complexity is **O(n)**. The space complexity is **O(1)** because only a constant amount of extra space (e.g., for a loop counter) is needed, regardless of array size.
- **Insertion**:
  - **At the end**: If the array has available space, insertion at the end is an **O(1)** operation.

- **At the beginning or middle**: Inserting an element at the beginning or middle of an array requires shifting all subsequent elements to make room, which takes time proportional to the number of elements shifted. This results in an **O(n)** time complexity.
- **Space**: The space complexity is typically **O(1)** if the array has pre-allocated space. However, if the array needs to be resized (e.g., in a dynamic array that runs out of space), a new, larger array is created and elements are copied over, making the space complexity **O(n)** in that specific scenario.

- **Deletion**: Similar to insertion, the time complexity of deletion depends on the element's position:
  - **From the end**: Deleting the last element is an **O(1)** operation.
  - **From the beginning or middle**: Deleting an element from the beginning or middle requires shifting all subsequent elements to fill the resulting gap, leading to an **O(n)** time complexity.
  - **Space**: The space complexity for deletion is **O(1)**, as operations are typically done in-place.
- **Searching**:
  - **Unsorted array**: The standard method is a linear search, which in the worst case must check every element to find a target or confirm its absence. The time complexity is **O(n)**.
  - **Sorted array**: For a sorted array, the binary search algorithm can be used. It repeatedly divides the search interval in half, leading to a much faster time complexity of **O(log n)**.
  - **Space**: Both linear and binary search typically have a space complexity of **O(1)**, requiring constant extra space.

Searching: There are two types of :

- **Linear Search**
- **Binary Search**

Definition: Linear Search is a simple searching technique where each element of the array is checked sequentially until the target element is found or the end of the array is reached.

Algorithm Steps

Input: An array arr of n elements.

A target value x to search for.

Output:

The index of x in arr if found; otherwise, return -1.

Algorithm :

1. Start from the first element of the array (index = 0).
2. Compare each element of the array with the target value x:
3. If arr[index] == x, return index.
4. If the loop ends without finding x, return -1.

Pseudocode

```
function linearSearch(arr, x):
    for index from 0 to length(arr) - 1:
        if arr[index] == x:
            return index # Element found
    return -1  # Element not found
```

1) Linear Search – Iterative Version

```
import array

def linear_search(arr, target):
    for index in range(len(arr)):
        if arr[index] == target:
            return index
    return -1

# Ask for user input to create the array
n = int(input("Enter the number of elements in the array: "))
arr = array.array('i', [])  # 'i' represents integer type

print("Enter the elements of the array:")
for _ in range(n):
    arr.append(int(input()))

# Ask for target element to search
target = int(input("Enter the target element to search for: "))
```

```
# Perform linear search
result = linear_search(arr, target)

if result != -1:
    print(f"Element {target} found at index {result}.")
else:
    print(f"Element {target} not found in the array.")
```

2) Linear search – Recursive Version :

```
import array

def recursive_linear_search(arr, target, index=0):
    # Base case: if index reaches the end of the array
    if index == len(arr):
        return -1

    # Check if the element at the current index matches the target
    if arr[index] == target:
        return index

    # Recursive call to check the next element
    return recursive_linear_search(arr, target, index + 1)

# Ask for user input to create the array
n = int(input("Enter the number of elements in the array: "))
arr = array.array('i', [])  # 'i' represents integer type

print("Enter the elements of the array:")
for _ in range(n):
    arr.append(int(input()))

# Ask for the target element to search
target = int(input("Enter the target element to search for: "))

# Perform recursive linear search
result = recursive_linear_search(arr, target)

if result != -1:
    print(f"Element {target} found at index {result}.")
else:
    print(f"Element {target} not found in the array.")
```

Time Complexity :

Best Case: (1) (if the target is the first element).

Worst Case: (n) (if the target is at the end or not in the array).

Average Case: (n)

Space Complexity :

(1) (no additional memory required).

Linear search is straightforward and works on unsorted arrays, unlike binary search, which requires a sorted array.

Binary Search Algorithm

Input:

- A sorted array `arr` of `n` elements.
- A target value `x` to search for.

Output:

- The index of `x` in `arr` if found; otherwise, return `-1`.

Step 1: Initialize two pointers:

- `low = 0` (starting index of the array).
- `high = n - 1` (last index of the array).

Step 2: Repeat until `low` is less than or equal to `high`:

1. Calculate the middle index:

   mid=low+(high−low)/2

   (Using this formula avoids overflow in large arrays.)

2. Check the value at `mid`:
   - If `arr[mid] == x`, return `mid`.
   - If `arr[mid] > x`, narrow the search to the left half: `high = mid - 1`.
   - If `arr[mid] < x`, narrow the search to the right half: `low = mid + 1`.

Step 3: If the loop ends without finding `x`, return `-1`.

Pseudo Code :

```
function binarySearch(arr, x):
    low = 0
    high = length(arr) - 1

    while low <= high:
        mid = low + (high - low) / 2
        if arr[mid] == x:
            return mid
        else if arr[mid] > x:
            high = mid - 1
        else:
            low = mid + 1

    return -1
```

Program in Python – Iterative version

1) Binary Search – Iterative

```
import array

def binary_search(arr, target):
    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = (low + high) // 2

        # Check if the target is at the mid
        if arr[mid] == target:
            return mid
        # If the target is smaller, ignore the right half
        elif arr[mid] > target:
            high = mid - 1
        # If the target is larger, ignore the left half
        else:
            low = mid + 1

    return -1  # Target not found

# Ask for user input to create the array
n = int(input("Enter the number of elements in the sorted array: "))
arr = array.array('i', [])  # 'i' represents integer type

print("Enter the elements of the sorted array (in increasing order):")
```

```
for _ in range(n):
    arr.append(int(input()))

# Ask for the target element to search
target = int(input("Enter the target element to search for: "))

# Perform binary search
result = binary_search(arr, target)

if result != -1:
    print(f"Element {target} found at index {result}.")
else:
    print(f"Element {target} not found in the array.")
```

2) Recursive binary search

```
import array

def recursive_binary_search(arr, target, low, high):
    # Base condition: if the element is not found
    if low > high:
        return -1

    mid = (low + high) // 2

    # Check if the target is at the mid
    if arr[mid] == target:
        return mid
    # If the target is smaller, search in the left half
    elif arr[mid] > target:
        return recursive_binary_search(arr, target, low, mid - 1)
    # If the target is larger, search in the right half
    else:
        return recursive_binary_search(arr, target, mid + 1, high)

# Ask for user input to create the array
n = int(input("Enter the number of elements in the sorted array: "))
arr = array.array('i', [])  # 'i' represents integer type

print("Enter the elements of the sorted array (in increasing order):")
for _ in range(n):
    arr.append(int(input()))

# Ask for the target element to search
target = int(input("Enter the target element to search for: "))

# Perform recursive binary search
result = recursive_binary_search(arr, target, 0, len(arr) - 1)

if result != -1:
    print(f"Element {target} found at index {result}.")
else:
    print(f"Element {target} not found in the array.")
```

Time Complexity

Best Case: **O(1) when the element is found at the middle index in the first comparison.**

Average Case and Worst Case : $O(\log_2 n)$

Space Complexity : **Binary search algorithm divides the input array in half at every step, reducing the search space by half, and hence has a time complexity of logarithmic order.**

Space Complexity :

O(1) for the iterative version (constant space). Binary search algorithm requires only constant space for storing the low, high, and mid indices, and does not require any additional data structures, so its auxiliary space complexity is O(1).

O(logn) for the recursive version (stack space for recursion).

Difference between Linear Search and Binary Search

Linear Search	Binary Search
In linear search input data need not to be in sorted.	In binary search input data need to be in sorted order.
It is also called sequential search.	It is also called half-interval search.
The time complexity of linear search **O(n)**.	The time complexity of binary search **$O(\log_2 n)$**.
Multidimensional array can be used.	Only single dimensional array is used.
Linear search performs equality comparisons	Binary search performs ordering comparisons
It is less complex.	It is more complex.
It is very slow process.	It is very fast process.

Sorting: Sorting

Sorting refers to the operation or technique of arranging and rearranging sets of data in some specific order. A collection of records called a list where every record has one or more fields. The fields which contain a unique value for each record is termed as the key field.
For example, a phone number directory can be thought of as a list where each record has three fields - 'name' of the person, 'address' of that person, and their 'phone numbers'. Being unique phone number can work as a key to locate any record in the list.
Sorting is the operation performed to arrange the records of a table or list in some order according to some specific ordering criterion. Sorting is performed according to some key value of each record.
The records are either sorted either numerically or alphanumerically. The records are then arranged in ascending or descending order depending on the numerical value of the key. Here is an example, where the sorting of a lists of marks obtained by a student in any particular subject of a class.

Categories of Sorting

The techniques of sorting can be divided into two categories.

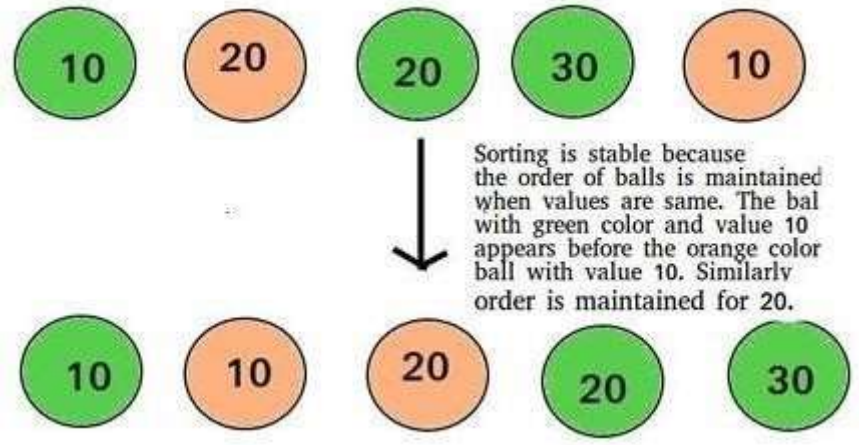
- Internal Sorting
- External Sorting

Internal Sorting: If all the data that is to be sorted can be adjusted at a time in the main memory, the internal sorting method is being performed.
External Sorting: When the data that is to be sorted cannot be accommodated in the memory at the same time and some has to be kept in auxiliary memory such as hard disk, floppy disk, magnetic tapes etc, then
external sorting methods are performed.

What is stable sorting?

Stability is mainly important when we have key value pairs with duplicate keys possible (like people names as keys and their details as values). And we wish to sort these objects by keys.

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted. Informally, stability means that equivalent elements retain their relative positions, after sorting.



Bubble Sort Algorithm

Definition:
Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the list is sorted.

Algorithm Steps

Input:

- An array `arr` of `n` elements.

Output:

- The sorted array `arr` in ascending order.

---

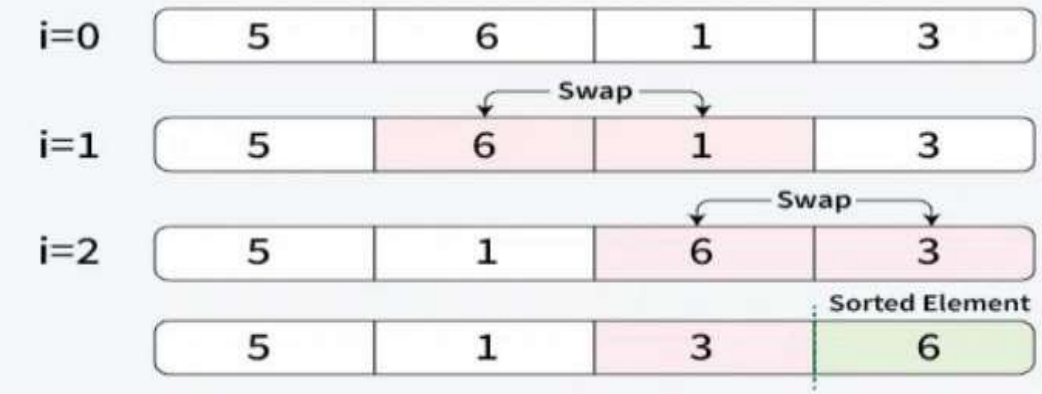
1. Start with the entire array.
2. Repeat the following steps for n−1n-1n−1 passes:
    - For each pass, compare each pair of adjacent elements:
        - If the first element is greater than the second, swap them.
    - After each pass, the largest unsorted element moves to its correct position at the end.
3. Repeat until no swaps are needed, indicating the array is sorted.

Pseudocode

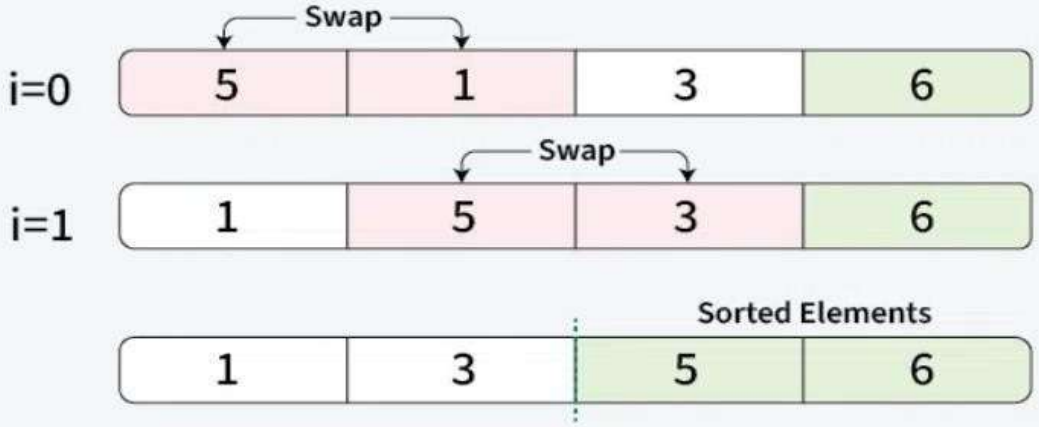
```
function bubbleSort(arr):
    n = length(arr)
    for i from 0 to n-1:
        for j from 0 to n-i-2:  # Compare adjacent elements
            if arr[j] > arr[j+1]:
                swap(arr[j], arr[j+1])  # Swap if in the wrong order
```

Python Implementation

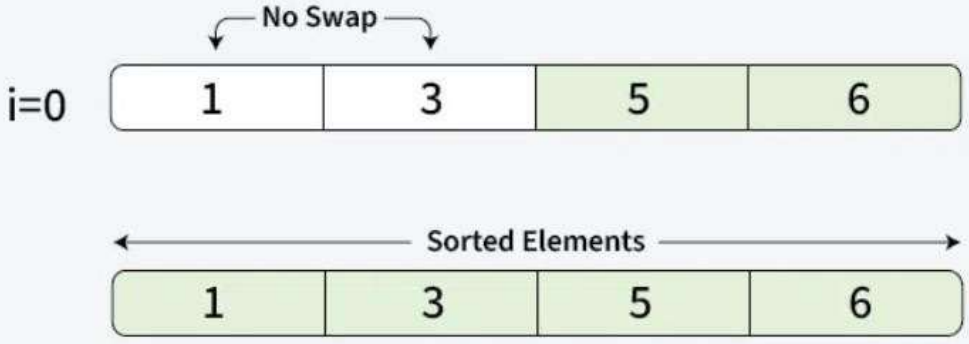
01 Step | Placing the 1st largest element at its correct position



02 Step | Placing 2nd largest element at its correct position



03 Step | Placing 3rd largest element at its correct position



Python Implementation

1) Bubble Sort – Ascending order

```
import array

def bubble_sort(arr):
    n = len(arr)
    # Traverse through all array elements
    for i in range(n):
        # Last i elements are already sorted, so no need to check them
        for j in range(0, n - i - 1):
            # Swap if the element found is greater than the next element
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

# Ask for user input to create the array
n = int(input("Enter the number of elements in the array: "))
arr = array.array('i', [])  # 'i' represents integer type

print("Enter the elements of the array:")
for _ in range(n):
    arr.append(int(input()))

# Perform Bubble Sort
bubble_sort(arr)

# Display the sorted array
print("Sorted array in ascending order:", arr)
```

2) Bubble Sort – in descending order

```
import array

def bubble_sort_descending(arr):
    n = len(arr)
    # Traverse through all array elements
    for i in range(n):
        # Last i elements are already sorted, no need to check them
        for j in range(0, n - i - 1):
            # Swap if the element found is smaller than the next element
            if arr[j] < arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

# Ask for user input to create the array
n = int(input("Enter the number of elements in the array: "))
arr = array.array('i', [])  # 'i' represents integer type

print("Enter the elements of the array:")
for _ in range(n):
    arr.append(int(input()))

# Perform Bubble Sort in descending order
bubble_sort_descending(arr)

# Display the sorted array
print("Sorted array in descending order:", arr)
```

Time Complexity

- **Best Case:** O(n) (when the array is already sorted, with optimization).
- **Worst Case:** $O(n^2)$ (when the array is in reverse order).
- **Average Case:** $O(n^2)$

Space Complexity

- O(1) (in-place sorting).

Bubble Sort is easy to understand but inefficient for large datasets, making it suitable for teaching or small-sized arrays. For larger datasets, more efficient algorithms like Quick Sort or Merge Sort are preferred.

Insertion Sort :

Definition:
Insertion Sort is a simple sorting algorithm that builds the final sorted array one element at a time. It works by taking one element from the unsorted portion and inserting it into the correct position in the sorted portion.

Algorithm Steps

Input:

- An array `arr` of `n` elements.

Output:

- The sorted array `arr` in ascending order.

---

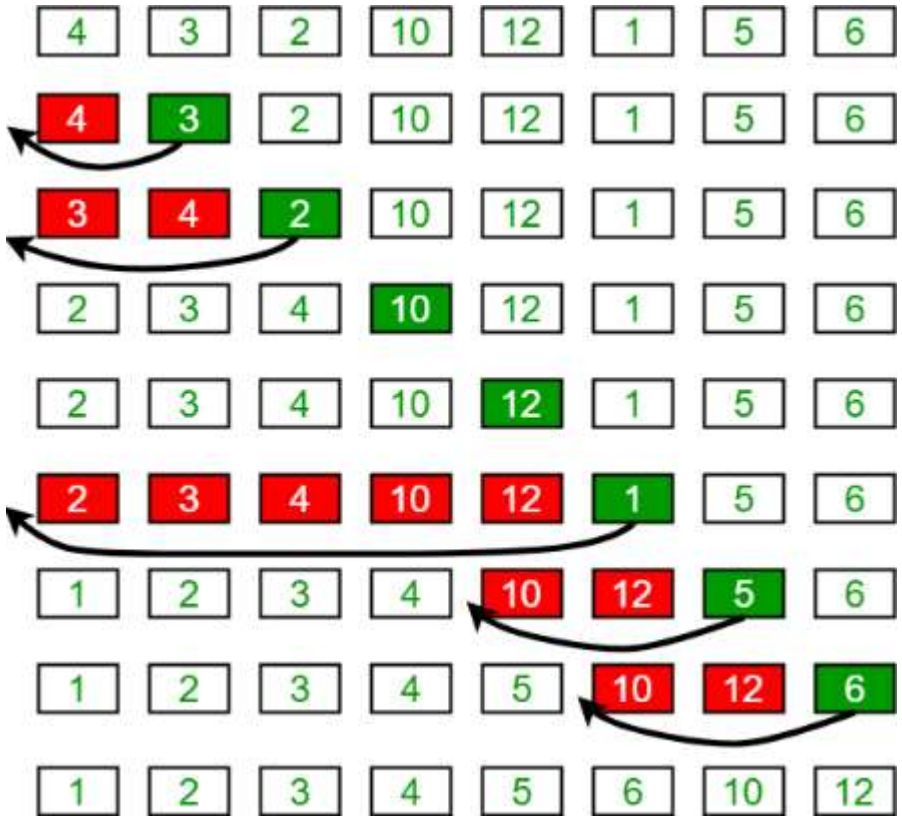
1. Start with the second element (`index = 1`) since the first element is considered sorted.
2. For each element in the unsorted portion of the array:
    - Take the current element (key).
    - Compare it with elements in the sorted portion of the array (from right to left).
    - Shift all larger elements in the sorted portion one position to the right.
    - Insert the key into its correct position in the sorted portion.
3. Repeat until the entire array is sorted.

Pseudocode

```
function insertionSort(arr):
```

```
for i from 1 to length(arr) - 1:
    key = arr[i]
    j = i - 1

    # Move elements of arr[0..i-1] that are greater than key
    # to one position ahead of their current position
    while j >= 0 and arr[j] > key:
        arr[j + 1] = arr[j]
        j = j - 1

    arr[j + 1] = key
```

Python Implementation
How It Works (Example)



1) Insertion Sort

```
import array

def insertion_sort_ascending(arr):
    # Traverse through 1 to len(arr)
    for i in range(1, len(arr)):
        key = arr[i]  # The current element to be inserted
        j = i - 1

        # Move elements of arr[0..i-1], that are greater than key, to one position ahead of their current position
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1

        # Place the key after the last element smaller than it
        arr[j + 1] = key

# Ask for user input to create the array
n = int(input("Enter the number of elements in the array: "))
arr = array.array('i', [])  # 'i' represents integer type (signed integer)

print("Enter the elements of the array:")
for _ in range(n):
    arr.append(int(input()))

# Perform Insertion Sort in ascending order
insertion_sort_ascending(arr)

# Display the sorted array
print("Sorted array in ascending order:", arr)
```

2) Insertion sort in descending order

```
import array

def insertion_sort_descending(arr):
    # Traverse through 1 to len(arr)
    for i in range(1, len(arr)):
        key = arr[i]  # The current element to be inserted
        j = i - 1

        # Move elements of arr[0..i-1], that are smaller than key, to one position ahead of their current
        position
        while j >= 0 and arr[j] < key:
            arr[j + 1] = arr[j]
            j -= 1

        # Place the key after the last element smaller than it
        arr[j + 1] = key

# Ask for user input to create the array
n = int(input("Enter the number of elements in the array: "))
arr = array.array('i', [])  # 'i' represents integer type (signed integer)

print("Enter the elements of the array:")
for _ in range(n):
    arr.append(int(input()))

# Perform Insertion Sort in descending order
insertion_sort_descending(arr)

# Display the sorted array
print("Sorted array in descending order:", arr)
```

Time Complexity

- **Best Case:** O(n) (when the array is already sorted).
- **Worst Case:** O(n^2) (when the array is in reverse order).
- **Average Case:** O(n^2)

Space Complexity

- O(1) (in-place sorting).

Insertion Sort is efficient for small or nearly sorted datasets and is often used in hybrid algorithms like Timsort for small subarrays.

Divide and Conquer strategy

Divide and Conquer Algorithm involves breaking a larger problem into smaller subproblems, solving them independently, and then combining their solutions to solve the original problem. The basic idea is to recursively divide the problem into smaller subproblems until they become simple enough to be solved directly. Once the solutions to the subproblems are obtained, they are then combined to produce the

overall solution**.**

Quick Sort

QuickSort is a sorting algorithm based on the Divide and Conquer that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.
Working of quick Sort Algorithm

QuickSort works on the principle of divide and conquer, breaking down the problem into smaller sub-problems.

There are mainly three steps in the algorithm:

- Choose a Pivot: Select an element from the array as the pivot. The choice of pivot can vary (e.g., first element, last element, random element, or median).
- Partition the Array: Rearrange the array around the pivot. After partitioning, all elements smaller than the pivot will be on its left, and all elements greater than the pivot will be on its right. The pivot is then in its correct position, and we obtain the index of the pivot.
- Recursively Call: Recursively apply the same process to the two partitioned sub-arrays (left and right of the pivot).
- Base Case: The recursion stops when there is only one element left in the sub-array, as a single element is already sorted

```
initial call to sort entire array Quicksort(A, 1, length[A])

Quicksort (A, p, r)
if p < r
   then q = Partition(A, p, r)
      Quicksort(A, p, q)
      Quicksort(A, q+1, r)

Partition (A, p, r)
   x = A[p]
   i = p-1
   j = r+1
   while TRUE
   do repeat j = j-1
         until A[j] <= x
      repeat i = i+1
         until A[i] >= x
      if i < j
         then exchange A[i], A[j]
         else return j
```

Python Implementation of Quick Sort

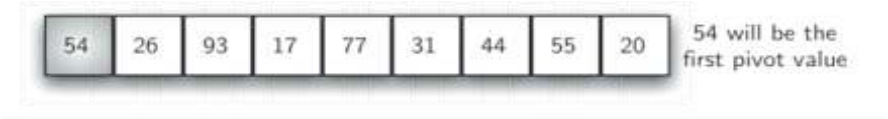
1) Quick sort – ascending order

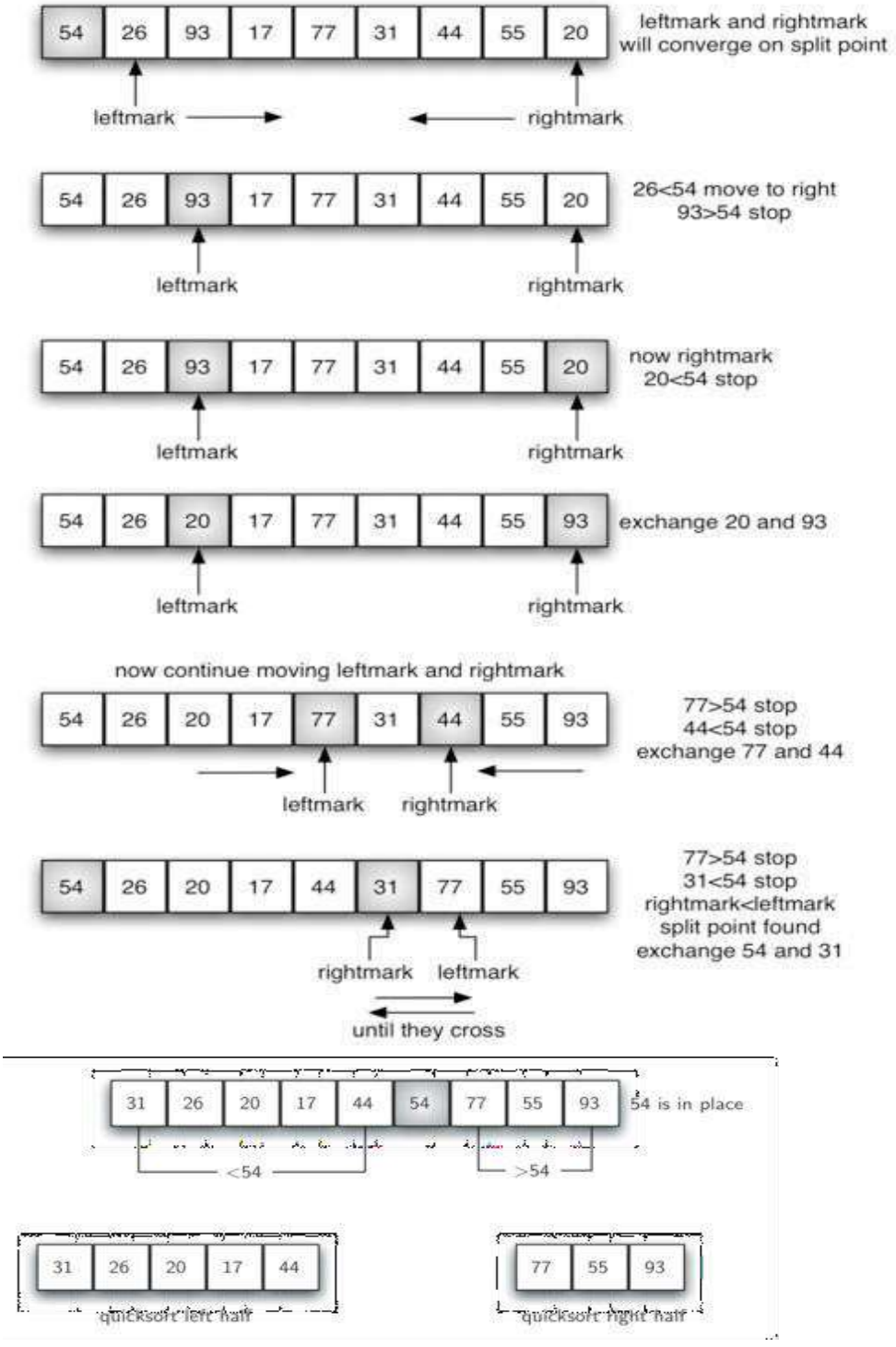
```
import array

def quick_sort(arr, start, end):
    if start < end:
        pivot_index = partition(arr, start, end)
        quick_sort(arr, start, pivot_index - 1)
        quick_sort(arr, pivot_index + 1, end)

def partition(arr, start, end):
    pivot = arr[start]  # First element as pivot
    low = start + 1
    high = end

    while True:
        while low <= high and arr[high] >= pivot:
            high -= 1
        while low <= high and arr[low] <= pivot:
            low += 1
        if low <= high:
            arr[low], arr[high] = arr[high], arr[low]
        else:
            break

    arr[start], arr[high] = arr[high], arr[start]
    return high

# Ask for user input to create the array
n = int(input("Enter the number of elements in the array: "))
arr = array.array('i', [])  # 'i' represents integer type (signed integer)

print("Enter the elements of the array:")
for _ in range(n):
    arr.append(int(input()))

# Perform Quick Sort in ascending order
quick_sort(arr, 0, len(arr) - 1)

# Display the sorted array
print("Sorted array in ascending order:", arr)
```

2) Quick sort – descending order

```
import array

def quick_sort_desc(arr, start, end):
    if start < end:
        pivot_index = partition_desc(arr, start, end)
        quick_sort_desc(arr, start, pivot_index - 1)
        quick_sort_desc(arr, pivot_index + 1, end)

def partition_desc(arr, start, end):
    pivot = arr[start]  # First element as pivot
    low = start + 1
    high = end

    while True:
        # Find the first element smaller than or equal to the pivot from the left
        while low <= high and arr[high] <= pivot:
```

```
            high -= 1
        # Find the first element larger than or equal to the pivot from the right
        while low <= high and arr[low] >= pivot:
            low += 1
        if low <= high:
            arr[low], arr[high] = arr[high], arr[low]  # Swap values
        else:
            break

    arr[start], arr[high] = arr[high], arr[start]  # Swap pivot with high
    return high

# User input
user_input = input("Enter integers separated by spaces: ")
numbers = list(map(int, user_input.split()))

# Convert to array
arr = array.array('i', numbers)

print(f"Original array: {list(arr)}")
quick_sort_desc(arr, 0, len(arr) - 1)
print(f"Sorted array (descending order): {list(arr)}")
```





Best Case Complexity - In Quicksort, the best-case occurs when the pivot element is the middle element or near to the middle element. The best-case time complexity of quicksort is $O(n*\log_2 n)$.

Average Case Complexity - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of quicksort is $O(n*\log_2 n)$.

Worst Case Complexity - In quick sort, worst case occurs when the pivot element is either greatest or smallest element. Suppose, if the pivot element is always the last element of the array, the worst case would occur when the given array is sorted already in ascending or descending order. The worst-case time complexity of quicksort is $O(n^2)$.

Space Complexity of Quick Sort

Space Complexity	Explanation
O(log n)	Space complexity for the recursive call stack in the best and average cases, where the array is evenly partitioned.
O(n)	Space complexity in the worst case, where the array is highly unbalanced, leading to deep recursion.

It is not stable sort.

Merge Sort :
Definition:
Merge Sort is a divide-and-conquer sorting algorithm that recursively divides the array into smaller subarrays, sorts those subarrays, and then merges them back together to produce the sorted array.

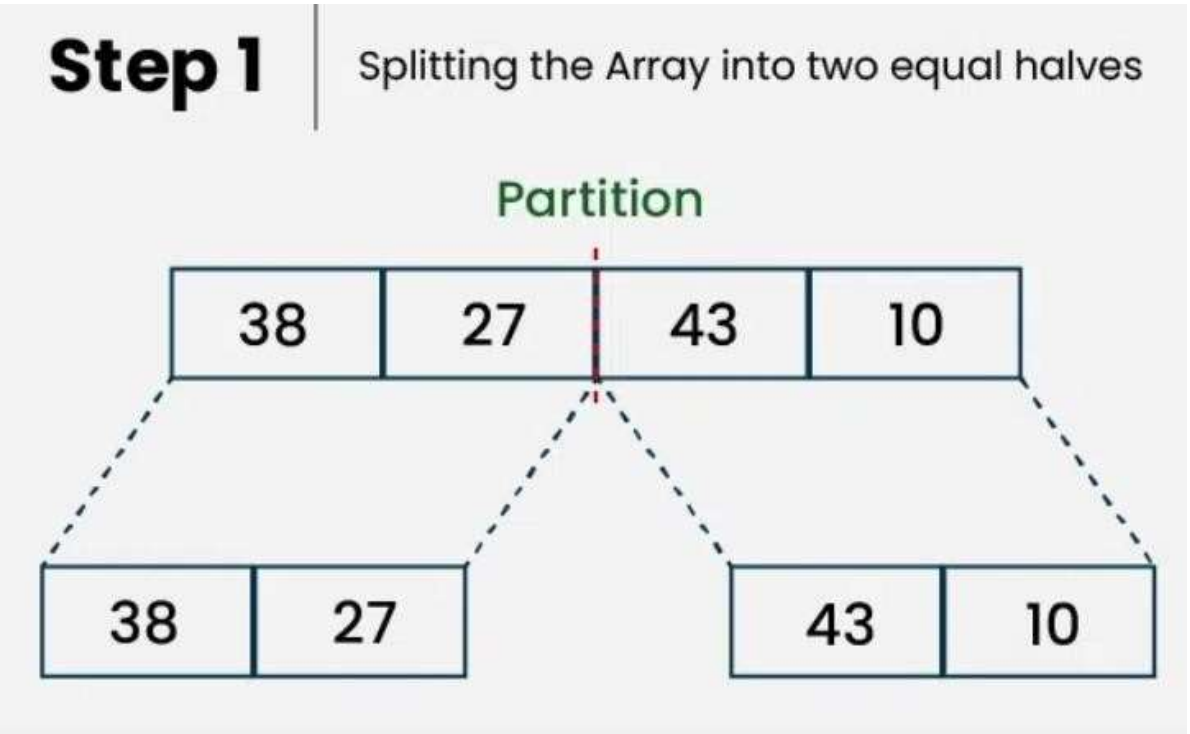
Algorithm Steps
Input: An array arr of n elements.
Output: The sorted array arr in ascending order.

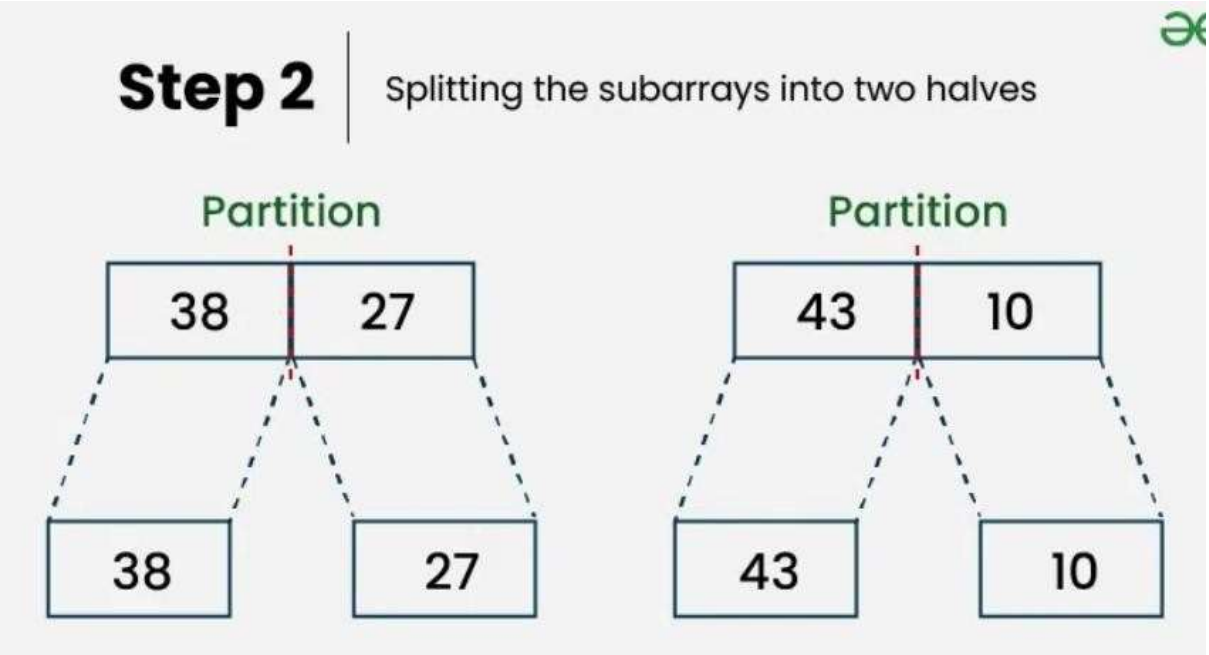
- Divide:
  If the array has more than one element, divide it into two halves.
- Recursively apply merge sort to each half.
- Merge:
  Merge the two sorted halves into a single sorted array by comparing elements from both halves.
- Base Case:
  If the array has one or zero elements, it is already sorted.

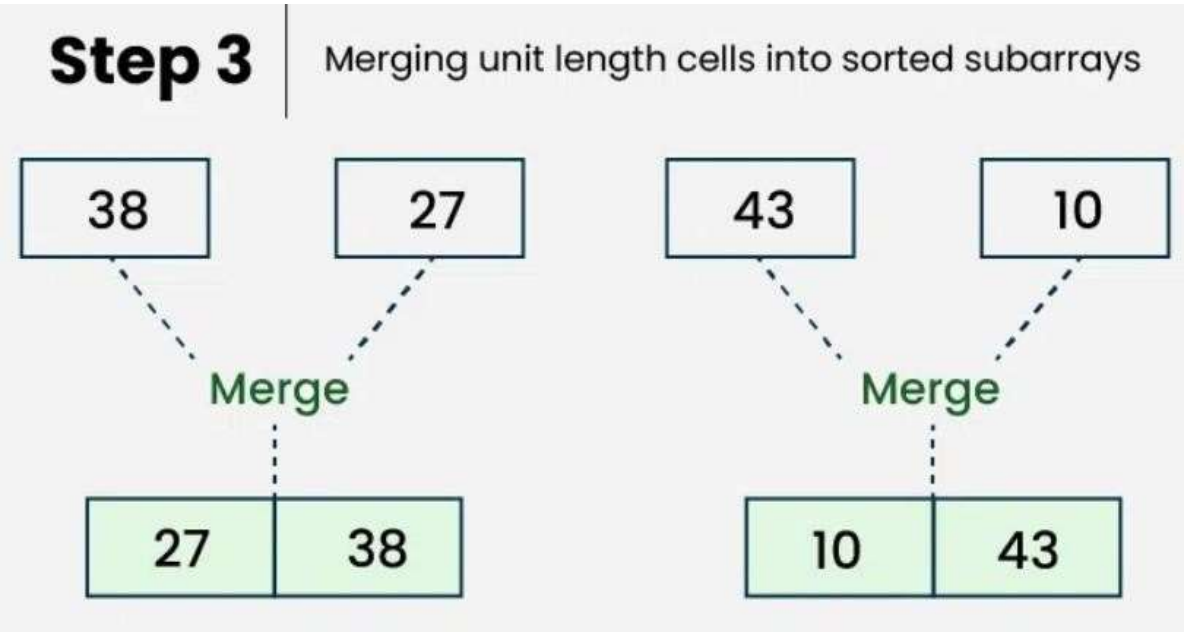
Pseudo Code

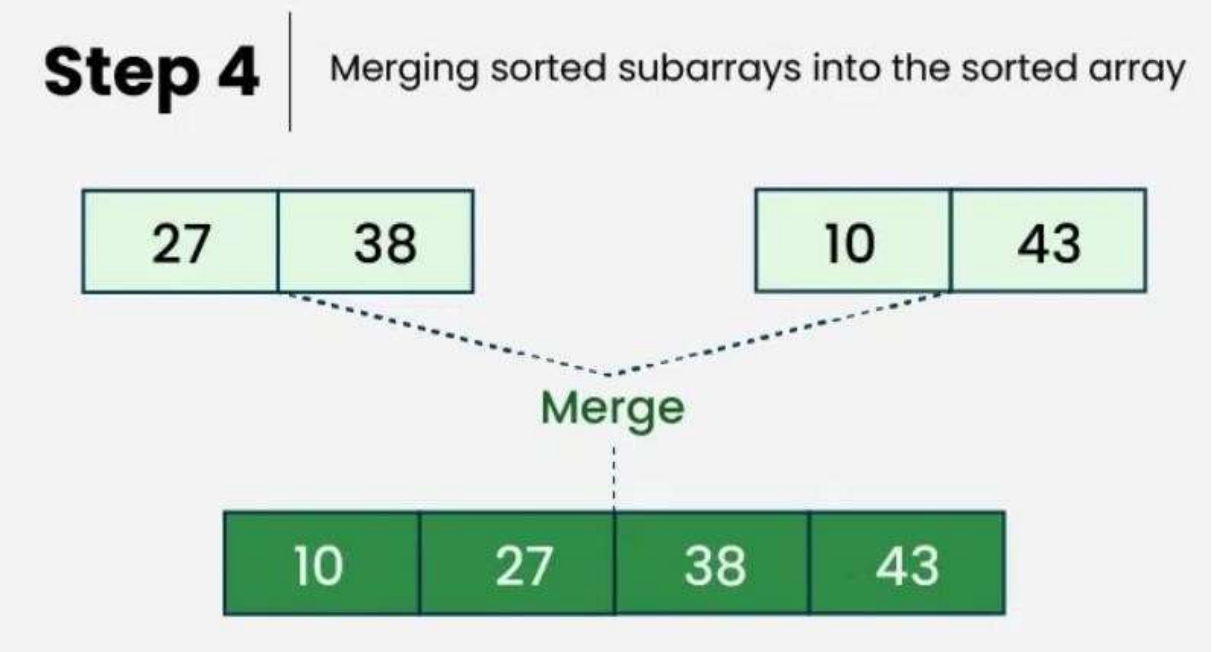
```
If len(array) > 1 Then
  # This is the point where the array is divided into two subarrays
  halfArray = len(array) / 2

  FirstHalf = array[:halfArray]
  # The first half of the data set
```

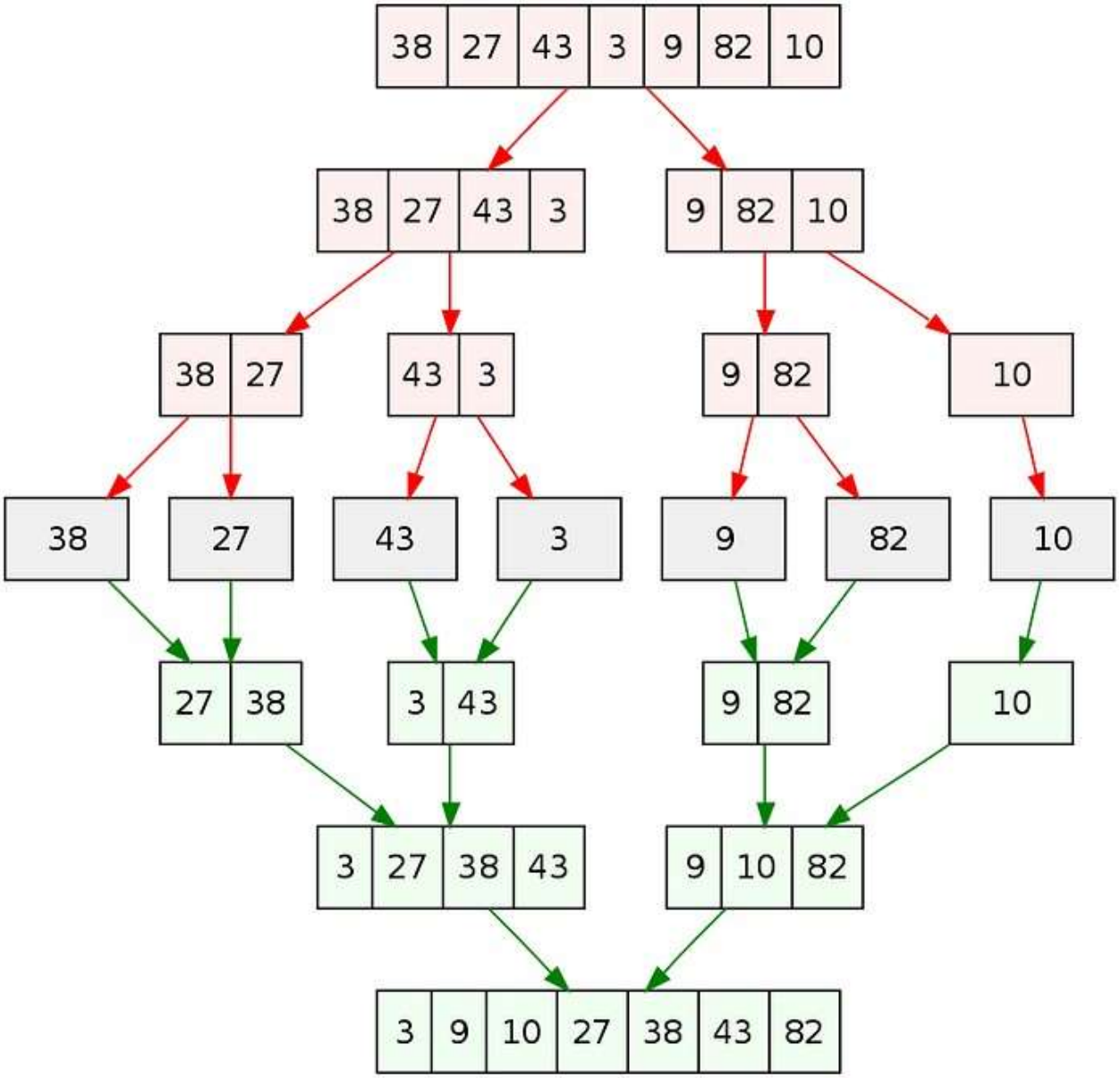
```
  SecondHalf = array[halfArray:]
  # The second half of the data set

  # Sort the two halves
  mergeSort(FirstHalf)
  mergeSort(SecondHalf)

  k = 0

  # Begin swapping values
  While i < len(FirstHalf) and j < len(SecondHalf)
    If FirstHalf[i] < SecondHalf[j] Then
      array[k] = FirstHalf[i]
      i += 1
    Else
      array[k] = SecondHalf[j]
      j += 1
      k += 1
    EndIf
  EndWhile
EndIf
```




Step 2
Splitting the subarrays into two halves
Partition
38
27
38
27
Partition
43
10
43
10


Step 3
Merging unit length cells into sorted subarrays
38
27
Merge
27
38
43
10
Merge
10
43


Step 4
Merging sorted subarrays into the sorted array
27
38
10
43
Merge
10
27
38
43


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

1) **Python Implementation** Merge Sort

```
import array

def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]

        # Recursively sort both halves
        merge_sort(left_half)
        merge_sort(right_half)

        # Merge the sorted halves
        i = j = k = 0
        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1

        # Copy remaining elements of left_half (if any)
        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

        # Copy remaining elements of right_half (if any)
        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1

# User input
user_input = input("Enter integers separated by spaces: ")
numbers = list(map(int, user_input.split()))

# Convert to array
arr = array.array('i', numbers)

print(f"Original array: {list(arr)}")
merge_sort(arr)
print(f"Sorted array: {list(arr)}")
```

    - **Best Case:** O(n log n), When the array is already sorted or nearly sorted.
    - **Average Case:** O(n log n), When the array is randomly ordered.
    - **Worst Case:** O(n log n), When the array is sorted in reverse order.
- **Auxiliary Space:** O(n), Additional space is required for the temporary array used during merging.

Advantages of Merge Sort:

- **Stability:** Merge sort is a stable sorting algorithm, which means it maintains the relative order of equal elements in the input array.
- **Guaranteed worst-case performance:** Merge sort has a worst-case time complexity of **O(N logN)** , which means it performs well even on large datasets.
- **Simple to implement:** The divide-and-conquer approach is straightforward.
- **Naturally Parallel**: We independently merge subarrays that makes it suitable for parallel processing.
-

Disadvantages of Merge Sort:

- **Space complexity:** Merge sort requires additional memory to store the merged sub-arrays during the sorting process.
- **Not in-place:** Merge sort is not an in-place sorting algorithm, which means it requires additional memory to store the sorted data. This can be a disadvantage in applications where memory usage is a concern.
- S**lower than QuickSort in general**. QuickSort is more cache friendly because it works in-place.

Time Complexity :

Algorithm	Best Case	Worst Case	Average Case	Stable	In-Place
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes	Yes
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes	Yes
Quick Sort	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	No	Yes
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Yes	No

Space Complexity :

Algorithm	Best Case Space Complexity	Worst Case Space Complexity	Auxiliary Space	In-Place
Bubble Sort	$O(1)$	$O(1)$	$O(1)$	Yes
Insertion Sort	$O(1)$	$O(1)$	$O(1)$	Yes
Quick Sort	$O(\log n)$	$O(n)$ (due to recursion)	$O(\log n)$	Yes
Merge Sort	$O(n)$	$O(n)$	$O(n)$	No

Advantages of Array

- Arrays allow **random access** to elements. This makes accessing elements by position faster.
- Arrays have **better cache locality** which makes a pretty big difference in performance.
- Arrays **represent multiple data items of the same type** using a single name.
- Arrays are used to implement the other data structures like linked lists, stacks, queues, trees, graphs, etc.

Disadvantages of Array

- As arrays have a fixed size, once the memory is allocated to them, it cannot be

increased or decreased, making it impossible to store extra data if required. An array of fixed size is referred to as a static array.

- Allocating less memory than required to an array leads to loss of data.
- An array is homogeneous in nature so, a single array cannot store values of different data types.
- Arrays store data in contiguous memory locations, which makes deletion and insertion very difficult to implement. This problem is overcome by implementing linked lists, which allow elements to be accessed sequentially.

Applications of Array

- They are used in the implementation of other data structures such as array lists, heaps, hash tables, vectors, and matrices.
- Database records are usually implemented as arrays.
- It is used in lookup tables by computer.