ASSIGNMENT

BINARY SEARCH

Binary Search is a searching algorithm for finding an element's position in a sorted array.

In this approach, the element is always searched in the middle of a portion of an array.

> Binary search can be implemented only on a sorted list of items. If the elements are not sorted already, we need to sort them first.

Binary Search Working

Binary Search Algorithm can be implemented in two ways which are discussed below.

1. Iterative Method
2. Recursive Method

The recursive method follows the divide and conquer approach.

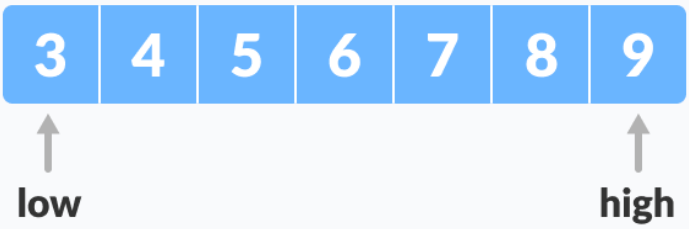
The general steps for both methods are discussed below.

1. The array in which searching is to be performed is:

Initial array

   Let `x = 4` be the element to be searched.

2. Set two pointers low and high at the lowest and the highest positions



   respectively.

Setting pointers

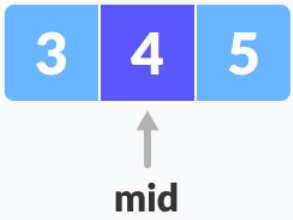
3. Find the middle element `mid` of the array ie. `arr[(low + high)/2] = 6`.

Mid element

4. If x == mid, then return mid.Else, compare the element to be searched with m.
5. If `x > mid`, compare `x` with the middle element of the elements on the right side of `mid`. This is done by setting `low` to `low = mid + 1`.

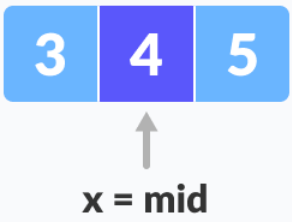
6. Else, compare `x` with the middle element of the elements on the left side of `mid`. This is done by setting `high` to `high = mid - 1`.

Finding mid element



Mid element

7. Repeat steps 3 to 6 until low meets high.



Found

8. `x = 4` is found.

Binary Search Algorithm

Iteration Method

```
do until the pointers low and high meet each other.
    mid = (low + high)/2
```

```
    if (x == arr[mid])
        return mid
    else if (x > arr[mid]) // x is on the right side
        low = mid + 1
    else                        // x is on the left side
        high = mid - 1
```

Recursive Method

```
binarySearch(arr, x, low, high)
    if low > high
        return False
    else
        mid = (low + high) / 2
        if x == arr[mid]
            return mid
        else if x > arr[mid]         // x is on the right side
            return binarySearch(arr, x, mid + 1, high)
        else                                 // x is on the right side
            return binarySearch(arr, x, low, mid - 1)
```

Python, Java, C/C++ Examples (Iterative Method)

Python
Java
C
C++

```
// Binary Search in Java

class BinarySearch {
  int binarySearch(int array[], int x, int low, int high) {

    // Repeat until the pointers low and high meet each other
    while (low <= high) {
      int mid = low + (high - low) / 2;

      if (array[mid] == x)
        return mid;

      if (array[mid] < x)
        low = mid + 1;

      else
```

```
        high = mid - 1;
    }

    return -1;
  }

  public static void main(String args[]) {
    BinarySearch ob = new BinarySearch();
    int array[] = { 3, 4, 5, 6, 7, 8, 9 };
    int n = array.length;
    int x = 4;
    int result = ob.binarySearch(array, x, 0, n - 1);
    if (result == -1)
      System.out.println("Not found");
    else
      System.out.println("Element found at index " + result);
  }
}
```

Python, Java, C/C++ Examples (Recursive Method)

Python
Java
C
C++

```
// Binary Search in Java

class BinarySearch {
  int binarySearch(int array[], int x, int low, int high) {

    if (high >= low) {
      int mid = low + (high - low) / 2;

      // If found at mid, then return it
      if (array[mid] == x)
        return mid;

      // Search the left half
      if (array[mid] > x)
        return binarySearch(array, x, low, mid - 1);

      // Search the right half
      return binarySearch(array, x, mid + 1, high);
```

```
    }

    return -1;
  }

  public static void main(String args[]) {
    BinarySearch ob = new BinarySearch();
    int array[] = { 3, 4, 5, 6, 7, 8, 9 };
    int n = array.length;
    int x = 4;
    int result = ob.binarySearch(array, x, 0, n - 1);
    if (result == -1)
      System.out.println("Not found");
    else
      System.out.println("Element found at index " + result);
  }
}
```

Binary Search Complexity

Time Complexities

- Best case complexity: `O(1)`
- Average case complexity: `O(log n)`
- Worst case complexity: `O(log n)`

Space Complexity

The space complexity of the binary search is `O(1)`.

Binary Search Applications

- In libraries of Java, .Net, C++ STL

- While debugging, the binary search is used to pinpoint the place where the error happens.

Linear Search

inear Search is defined as a sequential search algorithm that starts at one end and goes through each element of a list until the desired element is found, otherwise the search continues till the end of the data set. It is the easiest searching algorithm

Linear Search Algorithm

Given an array arr[] of N elements, the task is to write a function to search a given element x in arr[].

Examples:

Input: arr[] = {10, 20, 80, 30, 60, 50,110, 100, 130, 170}, x = 110;
Output: 6
Explanation: Element x is present at index 6

Input: arr[] = {10, 20, 80, 30, 60, 50,110, 100, 130, 170}, x = 175;
Output: -1
Explanation: Element x is not present in arr[].

Recommended Problem
Search an Element in an array
Arrays
Searching
Solve Problem
Submission count: 1.1L
Follow the below idea to solve the problem:

Iterate from 0 to N-1 and compare the value of every index with x if they match return index

Follow the given steps to solve the problem:

Start from the leftmost element of arr[] and one by one compare x with each element of arr[]

If x matches with an element, return the index.
If x doesn’t match with any of the elements, return -1.
Below is the implementation of the above approach:

```
// C# code to linearly search x in arr[]. If x
// is present then return its location, otherwise
// return -1
using System;

class GFG {
    public static int search(int[] arr, int x)
    {
        int N = arr.Length;
        for (int i = 0; i < N; i++) {
            if (arr[i] == x)
                return i;
        }
        return -1;
    }

    // Driver's code
    public static void Main()
    {
        int[] arr = { 2, 3, 4, 10, 40 };
        int x = 10;

        // Function call
        int result = search(arr, x);
        if (result == -1)
            Console.WriteLine(
                "Element is not present in array");
        else
            Console.WriteLine("Element is present at index "
                              + result);
    }
}

// This code is contributed by DrRoot_
```

Output
Element is present at index 3
Time complexity: O(N)
Auxiliary Space: O(1)

Linear Search Recursive Approach:
Follow the given steps to solve the problem:

If the size of the array is zero then, return -1, representing that the element is not found. This can also be treated as the base condition of a recursion call.
Otherwise, check if the element at the current index in the array is equal to the key or not i.e, arr[size – 1] == key
If equal, then return the index of the found key.
Below is the implementation of the above approach:

```
// C# Recursive Code For Linear Search
using System;

static class Test {
    static int[] arr = { 5, 15, 6, 9, 4 };

    // Recursive Method to search key in the array
    static int linearsearch(int[] arr, int size, int key)
    {
        if (size == 0) {
            return -1;
        }
        else if (arr[size - 1] == key) {
            // Return the index of found key.
            return size - 1;
        }
        else {
            return linearsearch(arr, size - 1, key);
        }
    }

    // Driver method
    public static void Main(String[] args)
    {
        int key = 4;

        // Method call to find key
        int index = linearsearch(arr, arr.Length, key);

        if (index != -1)
            Console.Write("The element " + key
                          + " is found at " + index
                          + " index of the given array.");
```

```
        else
            Console.Write("The element " + key
                    + " is not found.");
    }
}

// This Code is submitted by Susobhan Akhuli
```

Output
The element 4 is found at 4 index of the given array.
Time Complexity: O(N)
Auxiliary Space: O(N), for using recursive stack space.

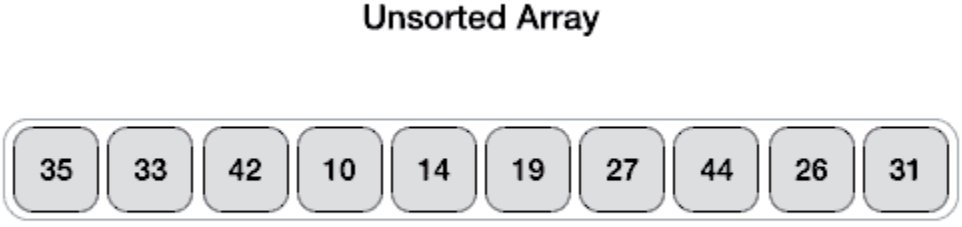
Quick Sort Algorithm

quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are O(n_2), respectively.

Partition in Quick Sort

The following animated representation explains how to find the pivot value in an array.



The pivot value divides the list into two parts. And recursively, we find the pivot for each sub-lists until all lists contains only one element.

Quick Sort Pivot Algorithm

Based on our understanding of partitioning in quick sort, we will now try to write an algorithm for it, which is as follows.

Step 1 − Choose the highest index value has pivot
Step 2 − Take two variables to point left and right of the list excluding pivot
Step 3 − left points to the low index
Step 4 − right points to the high
Step 5 − while value at left is less than pivot move right
Step 6 − while value at right is greater than pivot move left
Step 7 − if both step 5 and step 6 does not match swap left and right
Step 8 − if left ≥ right, the point where they met is new pivot

Quick Sort Pivot Pseudocode

The pseudocode for the above algorithm can be derived as –

```
function partitionFunc(left, right, pivot)
   leftPointer = left
   rightPointer = right - 1

   while True do
      while A[++leftPointer] < pivot do
         //do-nothing
      end while

      while rightPointer > 0 && A[--rightPointer] > pivot do
         //do-nothing
      end while

      if leftPointer >= rightPointer
         break
      else
         swap leftPointer,rightPointer
```

```
    end if

  end while

  swap leftPointer,right
  return leftPointer

end function
```

Quick Sort Algorithm

Using pivot algorithm recursively, we end up with smaller possible partitions. Each partition is then processed for quick sort. We define recursive algorithm for quicksort as follows –

```
Step 1 − Make the right-most index value pivot
Step 2 − partition the array using pivot value
Step 3 − quicksort left partition recursively
Step 4 − quicksort right partition recursively
```

Quick Sort Pseudocode

To get more into it, let see the pseudocode for quick sort algorithm –

```
procedure quickSort(left, right)

  if right-left <= 0
    return
  else
    pivot = A[right]
    partition = partitionFunc(left, right, pivot)
    quickSort(left,partition-1)
    quickSort(partition+1,right)
  end if

end procedure
```

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.

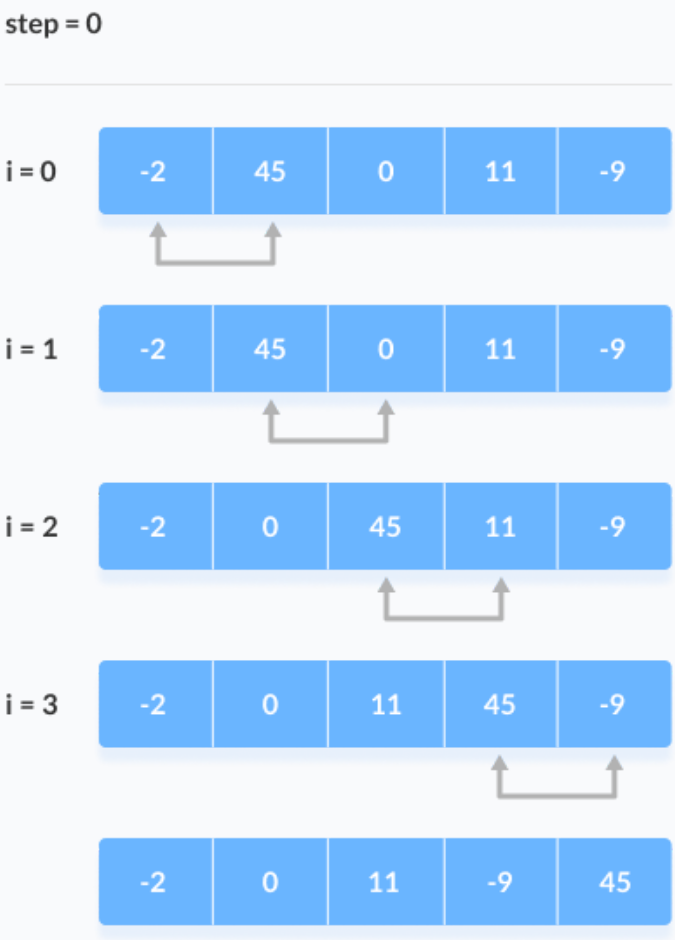
Working of Bubble Sort

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.

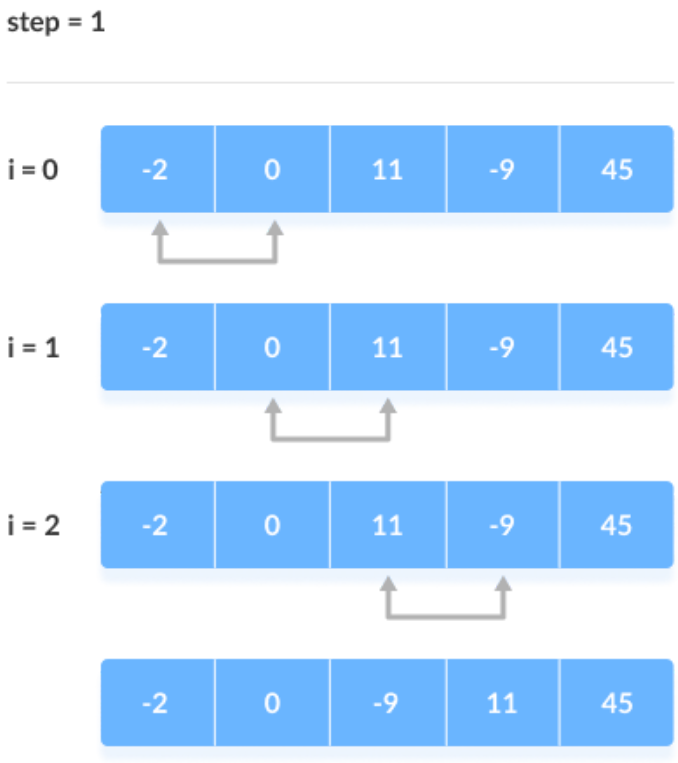


Compare the Adjacent Elements

2. Remaining Iteration

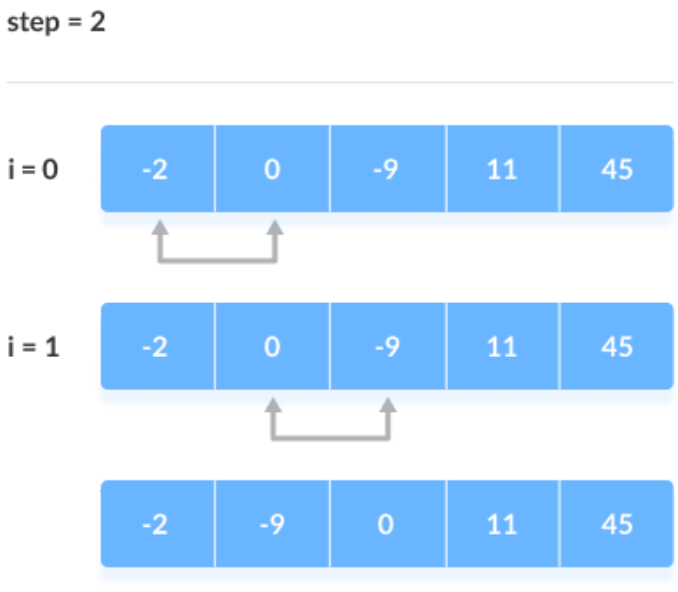
The same process goes on for the remaining iterations.

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



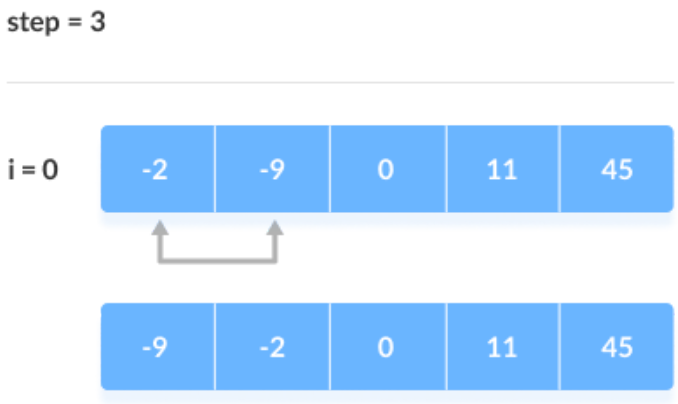
Put the largest element at the end

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 indexOfLastUnsortedElement-1
    if leftElement > rightElement
      swap leftElement and rightElement
end bubbleSort
```

Bubble Sort Code in Python, Java and C/C++

Python
Java
C
C++

```
// Bubble sort in Java

import java.util.Arrays;

class Main {

  // perform the bubble sort
  static void bubbleSort(int array[]) {
    int size = array.length;

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

```
      // loop to compare array elements
      for (int j = 0; j < size - i - 1; j++)

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

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

  public static void main(String args[]) {

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

    // call method using class name
    Main.bubbleSort(data);

    System.out.println("Sorted Array in Ascending Order:");
    System.out.println(Arrays.toString(data));
  }
}
```

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)
  swapped <- false
  for i <- 1 to indexOfLastUnsortedElement-1
    if leftElement > rightElement
      swap leftElement and rightElement
      swapped <- true
end bubbleSort
```

Optimized Bubble Sort in Python, Java, and C/C++

Python
Java
C
C++

```
// Optimized Bubble sort in Java

import java.util.Arrays;

class Main {

  // perform the bubble sort
  static void bubbleSort(int array[]) {
    int size = array.length;

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

      // check if swapping occurs
      boolean swapped = false;

      // loop to compare adjacent elements
```

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

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

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

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

    }
  }

  public static void main(String args[]) {

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

    // call method using the class name
    Main.bubbleSort(data);

    System.out.println("Sorted Array in Ascending Order:");
    System.out.println(Arrays.toString(data));
  }
}
```

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

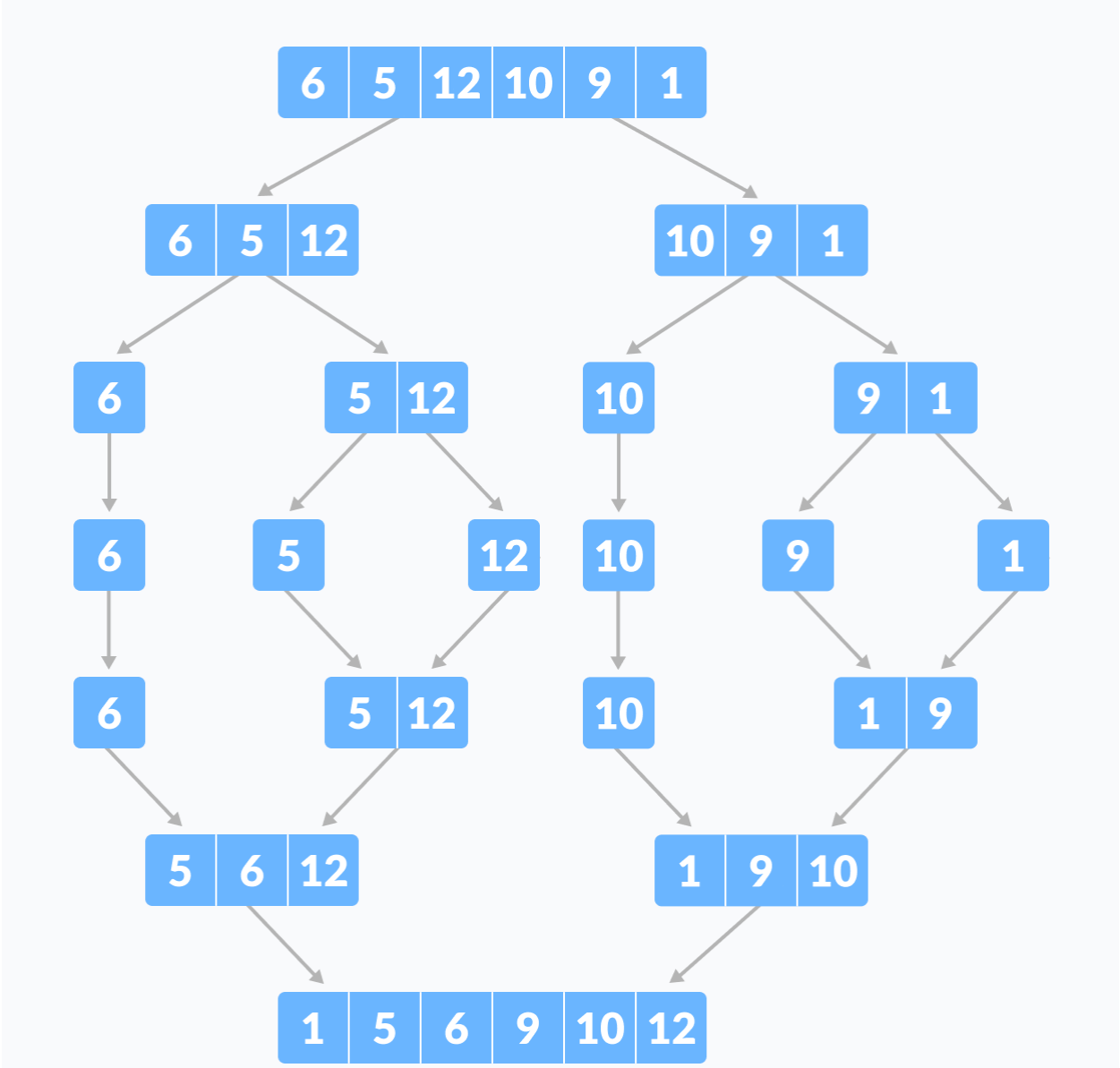
Bubble sort is used if

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

Merge Sort Algorithm

Merge Sort is one of the most popular sorting algorithms that is based on the principle of Divide and Conquer Algorithm.

Here, a problem is divided into multiple sub-problems. Each sub-problem is solved individually. Finally, sub-problems are combined to form the final solution.



Merge Sort example

Divide and Conquer Strategy

Using the Divide and Conquer technique, we divide a problem into subproblems. When the solution to each subproblem is ready, we 'combine' the results from the subproblems to solve the main problem.

Suppose we had to sort an array `A`. A subproblem would be to sort a sub-section of this array starting at index `p` and ending at index `r`, denoted as `A[p..r]`.

Divide

If q is the half-way point between p and r, then we can split the subarray `A[p..r]` into two arrays `A[p..q]` and `A[q+1, r]`.

Conquer

In the conquer step, we try to sort both the subarrays `A[p..q]` and `A[q+1, r]`. If we haven't yet reached the base case, we again divide both these subarrays and try to sort them.

Combine

When the conquer step reaches the base step and we get two sorted subarrays `A[p..q]` and `A[q+1, r]` for array `A[p..r]`, we combine the results by creating a sorted array `A[p..r]` from two sorted subarrays `A[p..q]` and `A[q+1, r]`.

MergeSort Algorithm

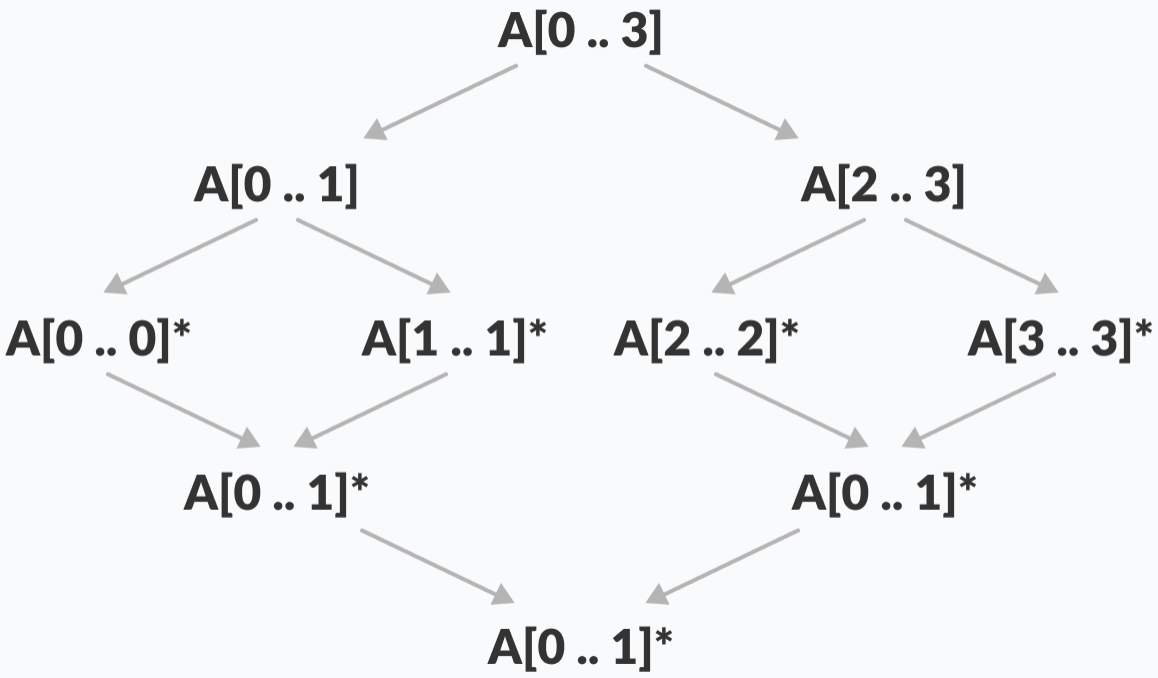
The MergeSort function repeatedly divides the array into two halves until we reach a stage where we try to perform MergeSort on a subarray of size 1 i.e. `p == r`.

After that, the merge function comes into play and combines the sorted arrays into larger arrays until the whole array is merged.

```
MergeSort(A, p, r):
    if p > r
        return
    q = (p+r)/2
    mergeSort(A, p, q)
    mergeSort(A, q+1, r)
    merge(A, p, q, r)
```

To sort an entire array, we need to call `MergeSort(A, 0, length(A)-1)`.

As shown in the image below, the merge sort algorithm recursively divides the array into halves until we reach the base case of array with 1 element. After that, the merge function picks up the sorted sub-arrays and merges them to gradually sort the entire array.



Merge sort in action

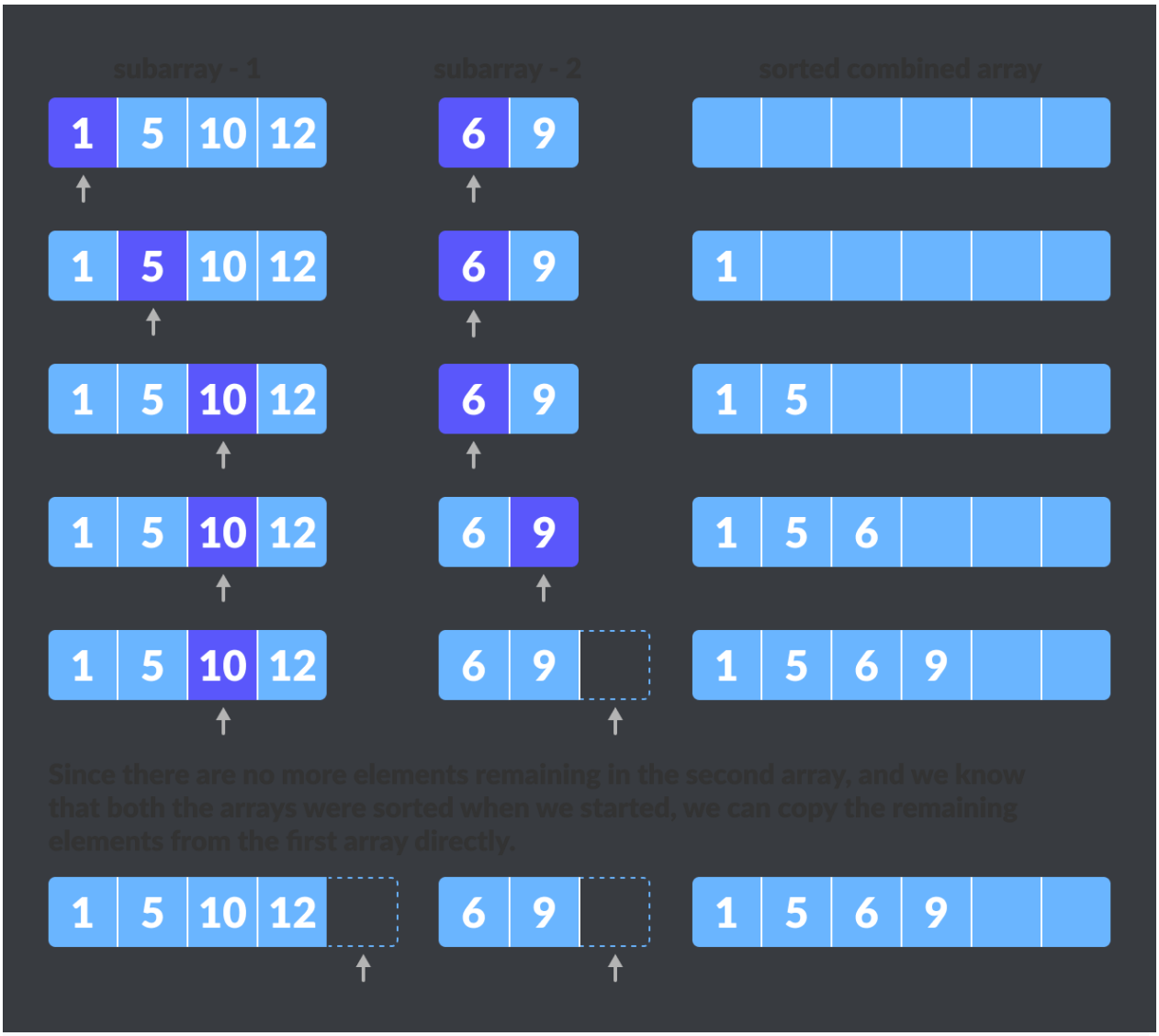
The merge Step of Merge Sort

Every recursive algorithm is dependent on a base case and the ability to combine the results from base cases. Merge sort is no different. The most important part of the merge sort algorithm is, you guessed it, `merge` step.

The merge step is the solution to the simple problem of merging two sorted lists(arrays) to build one large sorted list(array).

The algorithm maintains three pointers, one for each of the two arrays and one for maintaining the current index of the final sorted array.

```
Have we reached the end of any of the arrays?
    No:
        Compare current elements of both arrays
        Copy smaller element into sorted array
        Move pointer of element containing smaller element
    Yes:
        Copy all remaining elements of non-empty array
```



Merge step

Writing the Code for Merge Algorithm

A noticeable difference between the merging step we described above and the one we use for merge sort is that we only perform the merge function on consecutive sub-arrays.

This is why we only need the array, the first position, the last index of the first subarray(we can calculate the first index of the second subarray) and the last index of the second subarray.

Our task is to merge two subarrays `A[p..q]` and `A[q+1..r]` to create a sorted array `A[p..r]`. So the inputs to the function are A, p, q and r

The merge function works as follows:

1. Create copies of the subarrays `L ← A[p..q]` and `M ← A[q+1..r]`.
2. Create three pointers `i`, `j` and `k`
    1. `i` maintains current index of `L`, starting at 1
    2. `j` maintains current index of `M`, starting at 1
    3. `k` maintains the current index of `A[p..q]`, starting at `p`.
3. Until we reach the end of either `L` or `M`, pick the larger among the elements from `L` and `M` and place them in the correct position at `A[p..q]`
4. When we run out of elements in either `L` or `M`, pick up the remaining elements and put in `A[p..q]`

In code, this would look like:

```
// Merge two subarrays L and M into arr
void merge(int arr[], int p, int q, int r) {

    // Create L ← A[p..q] and M ← A[q+1..r]
    int n1 = q - p + 1;
    int n2 = r - q;

    int L[n1], M[n2];

    for (int i = 0; i < n1; i++)
        L[i] = arr[p + i];
    for (int j = 0; j < n2; j++)
        M[j] = arr[q + 1 + j];

    // Maintain current index of sub-arrays and main array
    int i, j, k;
```

```
    i = 0;
    j = 0;
    k = p;

    // Until we reach either end of either L or M, pick larger among
    // elements L and M and place them in the correct position at A[p..r]
    while (i < n1 && j < n2) {
        if (L[i] <= M[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = M[j];
            j++;
        }
        k++;
    }

    // When we run out of elements in either L or M,
    // pick up the remaining elements and put in A[p..r]
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = M[j];
        j++;
        k++;
    }
}
```

Merge( ) Function Explained Step-By-Step

A lot is happening in this function, so let's take an example to see how this would work.

As usual, a picture speaks a thousand words.

Merging two consecutive subarrays of array

The array `A[0..5]` contains two sorted subarrays `A[0..3]` and `A[4..5]`. Let us see how the merge function will merge the two arrays.

```
void merge(int arr[], int p, int q, int r) {
// Here, p = 0, q = 4, r = 6 (size of array)
```

Step 1: Create duplicate copies of sub-arrays to be sorted

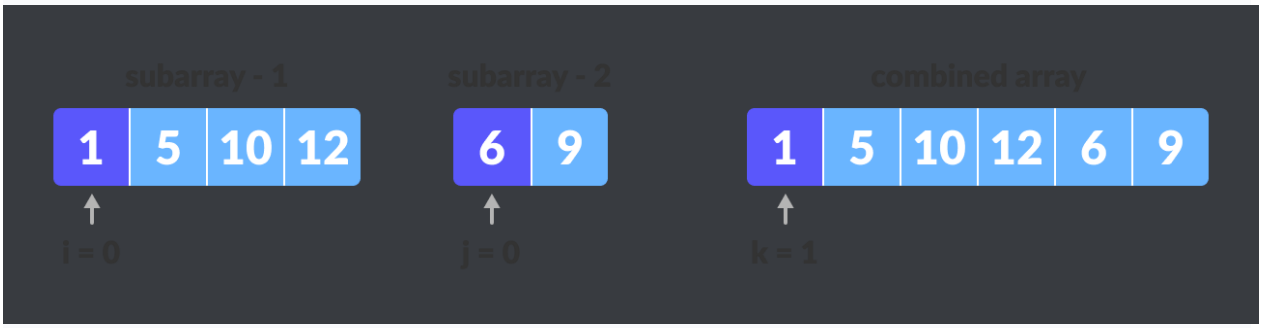
```
   // Create L ← A[p..q] and M ← A[q+1..r]
    int n1 = q - p + 1 = 3 - 0 + 1 = 4;
    int n2 = r - q = 5 - 3 = 2;

    int L[4], M[2];

    for (int i = 0; i < 4; i++)
        L[i] = arr[p + i];
        // L[0,1,2,3] = A[0,1,2,3] = [1,5,10,12]

    for (int j = 0; j < 2; j++)
        M[j] = arr[q + 1 + j];
        // M[0,1] = A[4,5] = [6,9]
```

Create copies of subarrays for merging

Step 2: Maintain current index of sub-arrays and main array

```
    int i, j, k;
```

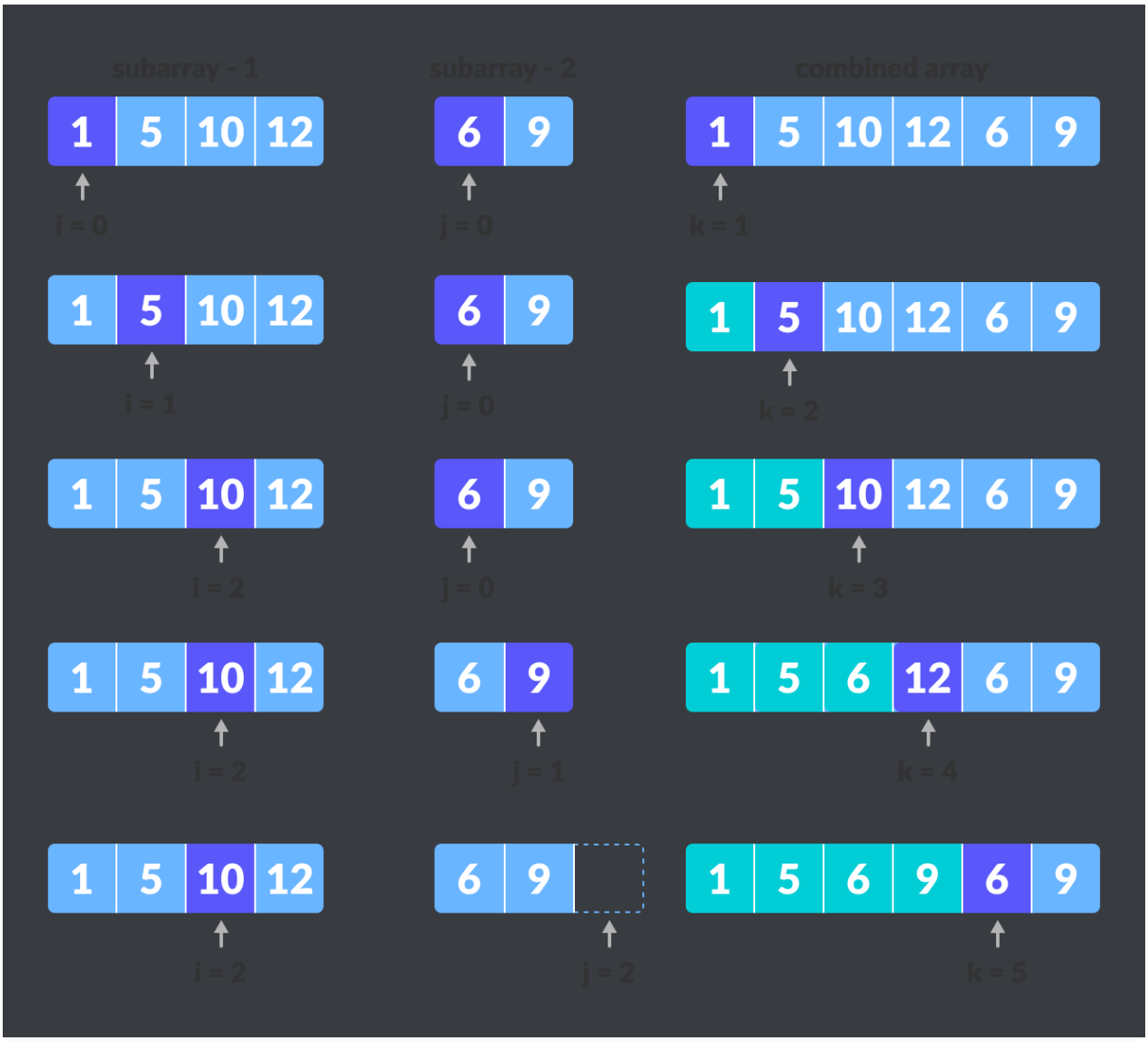
```
    i = 0;
    j = 0;
    k = p;
```



Maintain indices of copies of sub array and main array

Step 3: Until we reach the end of either L or M, pick larger among elements L and M and place them in the correct position at A[p..r]

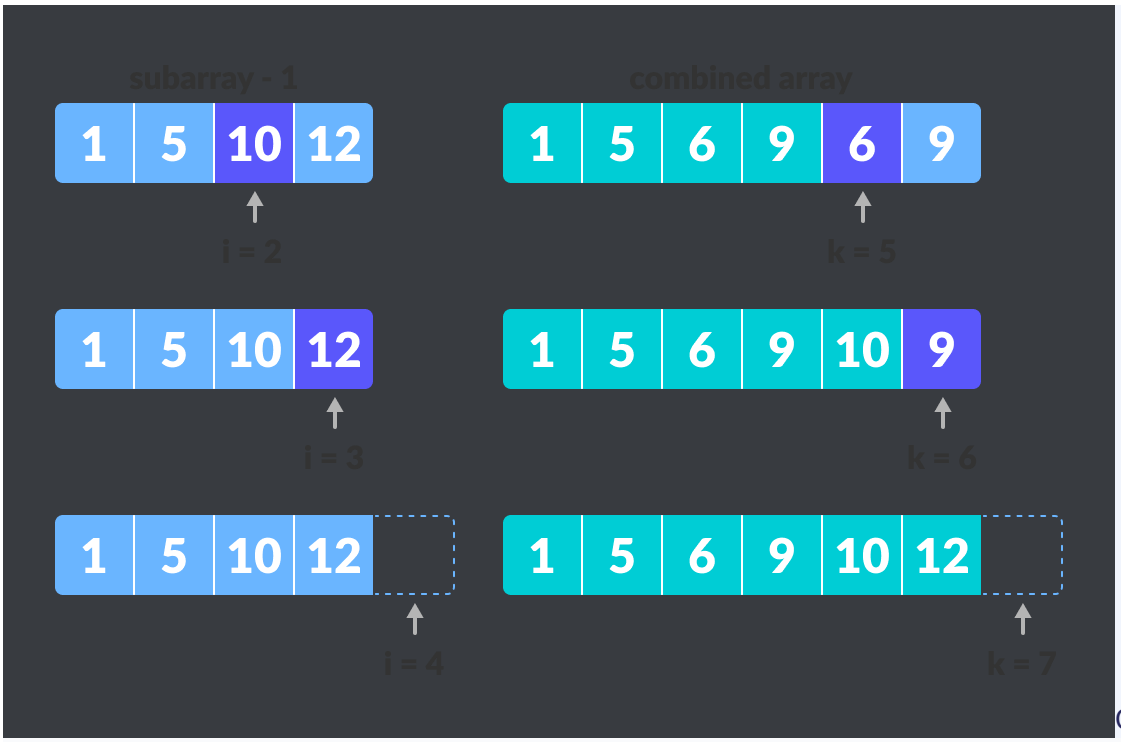
```
  while (i < n1 && j < n2) {
        if (L[i] <= M[j]) {
            arr[k] = L[i]; i++;
        }
        else {
            arr[k] = M[j];
            j++;
        }
        k++;
    }
```



Comparing individual elements of sorted subarrays until we reach end of one

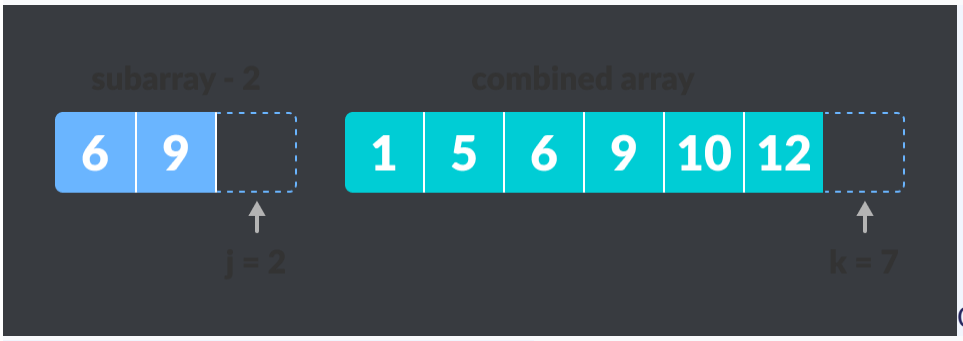
Step 4: When we run out of elements in either L or M, pick up the remaining elements and put in A[p..r]

```
    // We exited the earlier loop because j < n2 doesn't hold
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }
```



Copy the remaining elements from the first array to main subarray

```
    // We exited the earlier loop because i < n1 doesn't hold
    while (j < n2)
    {
        arr[k] = M[j];
        j++;
        k++;
    }
}
```



Copy remaining elements of second array to main subarray

This step would have been needed if the size of M was greater than L.

At the end of the merge function, the subarray `A[p..r]` is sorted.

Merge Sort Code in Python, Java, and C/C++

Python

Java

C

C++

```
// Merge sort in Java

class MergeSort {

  // Merge two subarrays L and M into arr
  void merge(int arr[], int p, int q, int r) {

    // Create L ← A[p..q] and M ← A[q+1..r]
    int n1 = q - p + 1;
    int n2 = r - q;

    int L[] = new int[n1];
    int M[] = new int[n2];

    for (int i = 0; i < n1; i++)
      L[i] = arr[p + i];
    for (int j = 0; j < n2; j++)
      M[j] = arr[q + 1 + j];

    // Maintain current index of sub-arrays and main array
    int i, j, k;
    i = 0;
    j = 0;
    k = p;

    // Until we reach either end of either L or M, pick larger among
    // elements L and M and place them in the correct position at A[p..r]
    while (i < n1 && j < n2) {
      if (L[i] <= M[j]) {
        arr[k] = L[i];
        i++;
      } else {
        arr[k] = M[j];
        j++;
      }
      k++;
    }

    // When we run out of elements in either L or M,
```

```
    // pick up the remaining elements and put in A[p..r]
    while (i < n1) {
      arr[k] = L[i];
      i++;
      k++;
    }

    while (j < n2) {
      arr[k] = M[j];
      j++;
      k++;
    }
  }

  // Divide the array into two subarrays, sort them and merge them
  void mergeSort(int arr[], int l, int r) {
    if (l < r) {

      // m is the point where the array is divided into two subarrays
      int m = (l + r) / 2;

      mergeSort(arr, l, m);
      mergeSort(arr, m + 1, r);

      // Merge the sorted subarrays
      merge(arr, l, m, r);
    }
  }

  // Print the array
  static void printArray(int arr[]) {
    int n = arr.length;
    for (int i = 0; i < n; ++i)
      System.out.print(arr[i] + " ");
    System.out.println();
  }

  // Driver program
  public static void main(String args[]) {
    int arr[] = { 6, 5, 12, 10, 9, 1 };

    MergeSort ob = new MergeSort();
    ob.mergeSort(arr, 0, arr.length - 1);

    System.out.println("Sorted array:");
    printArray(arr);
  }
```

```
}
```

Merge Sort Complexity

Time Complexity	
Best	O(n*log n)
Worst	O(n*log n)
Average	O(n*log n)
Space Complexity	O(n)
Stability	Yes

Time Complexity

Best Case Complexity: `O(n*log n)`

Worst Case Complexity: `O(n*log n)`

Average Case Complexity: `O(n*log n)`

Space Complexity

The space complexity of merge sort is `O(n)`.

Merge Sort Applications

- Inversion count problem
- External sorting
- E-commerce applications