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 can be implemented in two ways as shown below:

- Iterative Method
- Recursive Method

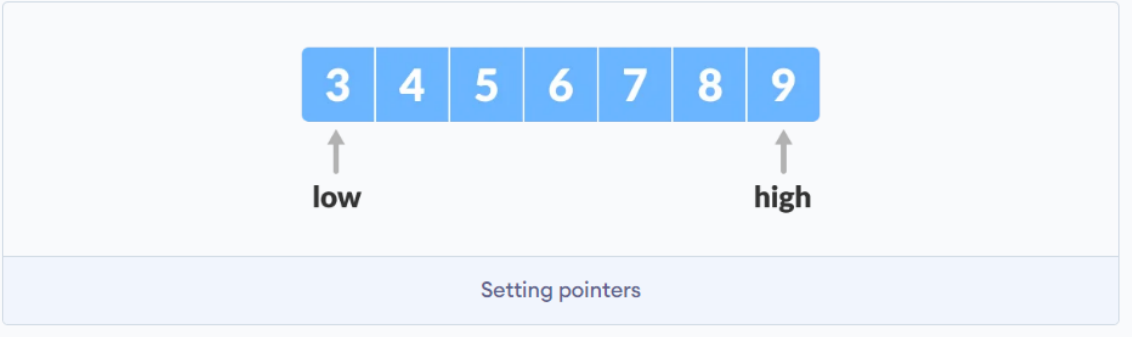
The recursive method follows the divide and conquer approach.

The general steps for both methods are discussed below.

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

Let x = 4 be the element to be searched.

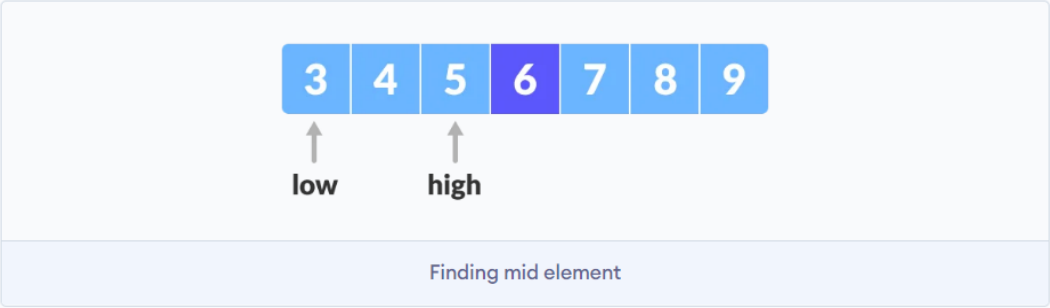
2. Set two pointers low and high at the lowest and the highest positions respectively.



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.

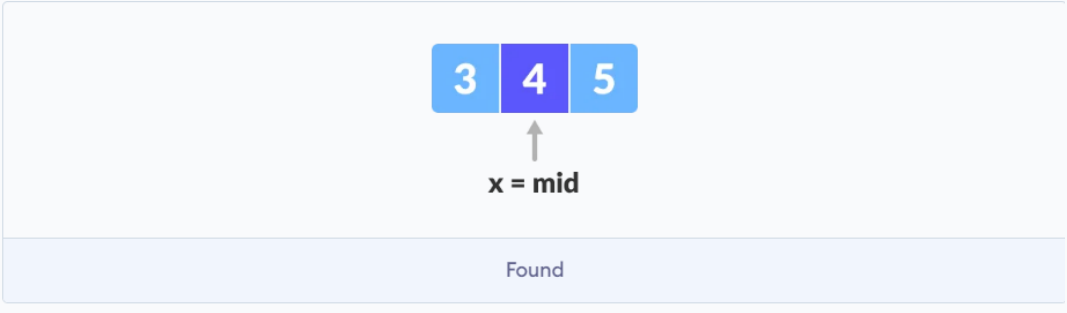


Finding mid element

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

Mid element

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)
```

Demonstration in c#

```
namespace Sorting_and_Searching
{
    0 references
    internal class Program
    {
        0 references
        static void Main(string[] args)
        {
            SearchandSort searchandSort = new SearchandSort();

            int[] array = { 1, 2, 3, 4, 5, 6, 6, 7, 8, 9};
            int x = 6;

            int result = searchandSort.binarySearch(array, x, 0, array.Length - 1);

            if(result == -1)
            {
                Console.WriteLine("Not Found");
            }
            else
            {
                Console.WriteLine("Item found at index {0}", result);
            }
        }
    }
}
```

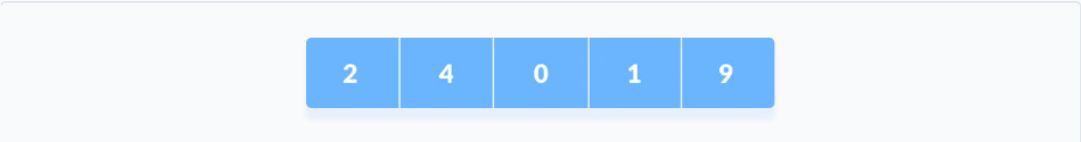
```
namespace Sorting_and_Searching
{
    2 references
    internal class SearchandSort
    {
        1 reference
        public int binarySearch(int[] arr, int x, int low, int high)
        {
            while(low <= high)
            {
                int mid = (low + high) / 2;

                if(arr[mid] == x)
                {
                    return mid;
                }
                else if (arr[mid] < x)
                {
                    low = mid + 1;
                }
                else
                {
                    high = mid - 1;
                }
            }
            return -1;
        }
    }
}
```

Linear Search

Linear search is a sequential searching algorithm where we start from one end and check every element of the list until the desired element is found. It is the simplest searching algorithm.

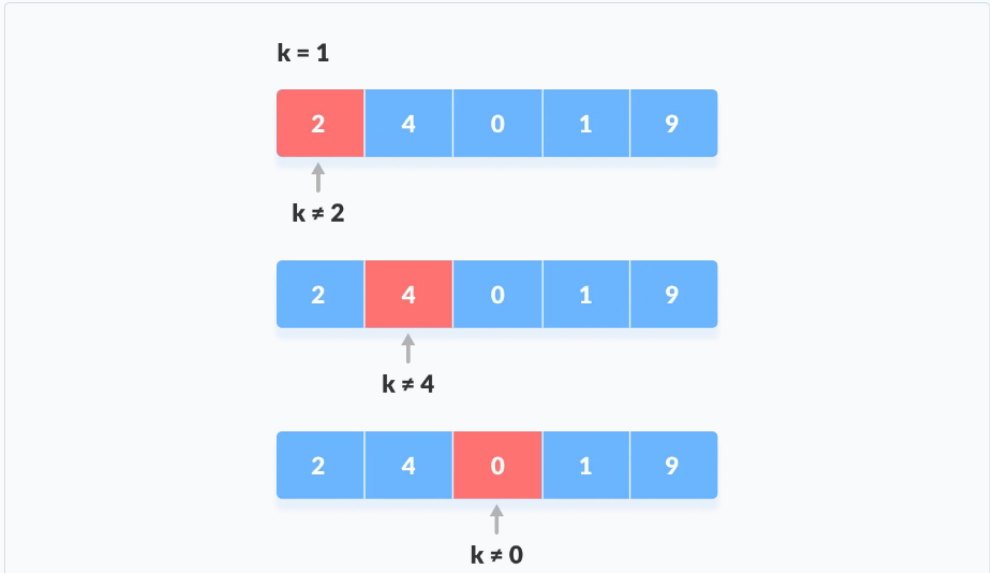
How Linear Search Works?

The following steps are followed to search for an element k = 1 in the list below.



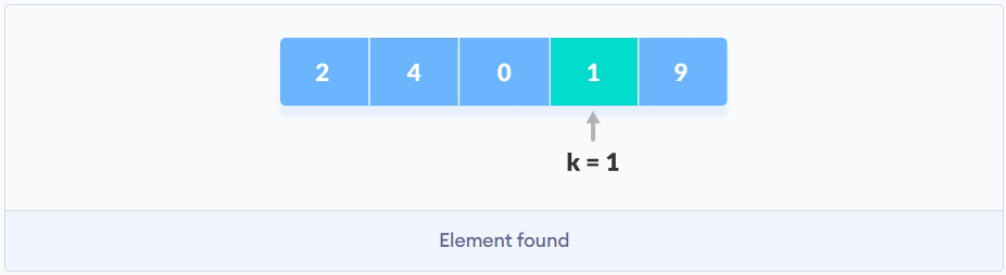
Array to be searched for

1. Start from the first element, compare k with each element x.



Compare with each element

2. If x == k, return th index.



3. Else return “Not Found”.

Linear Search Algorithm

```
LinearSearch(array, key)
  for each item in the array
    if item == value
      return its index
```

Demonstration in c#

```
namespace Sorting_and_Searching
{
    0 references
    internal class Program
    {
        0 references
        static void Main(string[] args)
        {
            SearchandSort searchandSort = new SearchandSort();
            int[] array = { 1, 2, 3, 4, 5, 6, 6, 7, 8, 9};
            int x = 4;

            int linearResult = searchandSort.LinearSearch(array, x);
            if (linearResult == -1)
            {
                Console.WriteLine("Not Found");
            }
            else
            {
                Console.WriteLine("Result of linear search: Item found at index {0}\n", linearResult);
            }
        }
    }
}
```

```
1 reference
public int LinearSearch(int[] arr, int x)
{
    int n = arr.Length;

    for(int i = 0; i < n; i++)
    {
        if (arr[i] == x)
        {
            return i;
        }
    }
    return -1;
}
```

Quicksort

Quicksort is a sorting algorithm based on the divide and conquer approach where

1. An array is divided into subarrays by selecting a pivot element (element selected from the array).

   While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.
2. The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.
3. At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

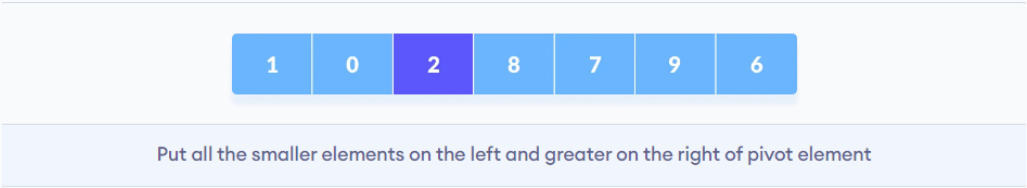
Working of Quicksort algorithm

1. **Select the pivot element.**

   There are different variations of quicksort where the pivot element is selected from different positions. Here, we will be selecting the rightmost element of the array as the pivot element.

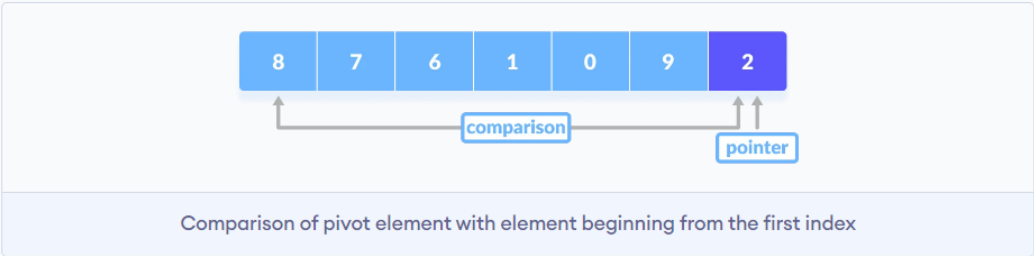
2. **Rearrange the array**

   Now the elements of the array are rearranged so that elements that are smaller than the pivot are put on the left and the elements greater than the pivot are put on the right.

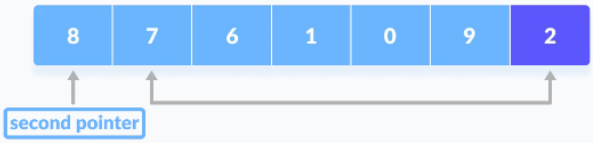


Here's how we rearrange the array:

a. A pointer is fixed at the pivot element. The pivot element is compared with the elements beginning from the first index.

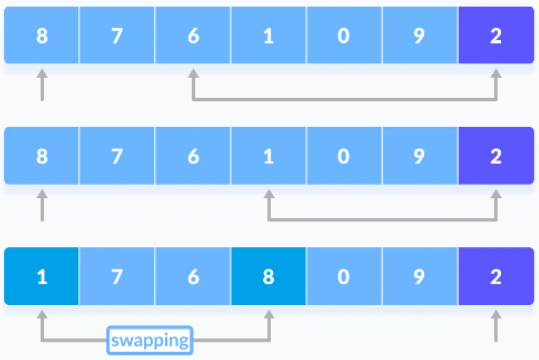


b. If the element is greater than the pivot element, a second pointer is set for that element.



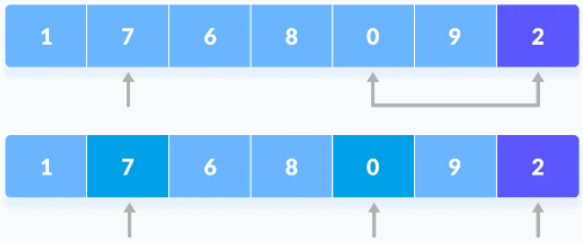
If the element is greater than the pivot element, a second pointer is set for that element.

c. Now, pivot is compared with other elements. If an element smaller than the pivot element is reached, the smaller element is swapped with the greater element found earlier.



Pivot is compared with other elements.

d. Again, the process is repeated to set the next greater element as the second pointer. And, swap it with another smaller element.

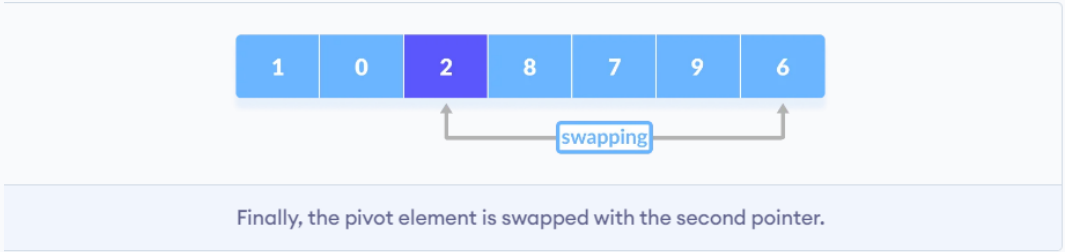


The process is repeated to set the next greater element as the second pointer.

e. The process goes on until the second last element is reached.

The process goes on until the second last element is reached.

f. Finally, the pivot element is swapped with the second pointer.



Finally, the pivot element is swapped with the second pointer.

3. Divide Subarrays

Pivot elements are again chosen for the left and the right sub-parts separately. And, step 2 is repeated.

Select pivot element of in each half and put at correct place using recursion

The subarrays are divided until each subarray is formed of a single element. At this point, the array is already sorted.

Quicksort Algorithm

```
quickSort(array, leftmostIndex, rightmostIndex)
  if (leftmostIndex < rightmostIndex)
    pivotIndex <- partition(array,leftmostIndex, rightmostIndex)
    quickSort(array, leftmostIndex, pivotIndex - 1)
    quickSort(array, pivotIndex, rightmostIndex)

partition(array, leftmostIndex, rightmostIndex)
  set rightmostIndex as pivotIndex
  storeIndex <- leftmostIndex - 1
  for i <- leftmostIndex + 1 to rightmostIndex
  if element[i] < pivotElement
    swap element[i] and element[storeIndex]
    storeIndex++
  swap pivotElement and element[storeIndex+1]
return storeIndex + 1
```

Bubble Sort

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

Working of Bubble Sort

1. **First Iteration (Compare and Swap)**
    a. Starting from the first index, compare the first and the second elements.
    b. If the first element is greater than the second element, they are swapped.
    c. Now, compare the second and the third elements. Swap them if they are not in order.
    d. The above process goes on until the last element.

Compare the Adjacent Elements

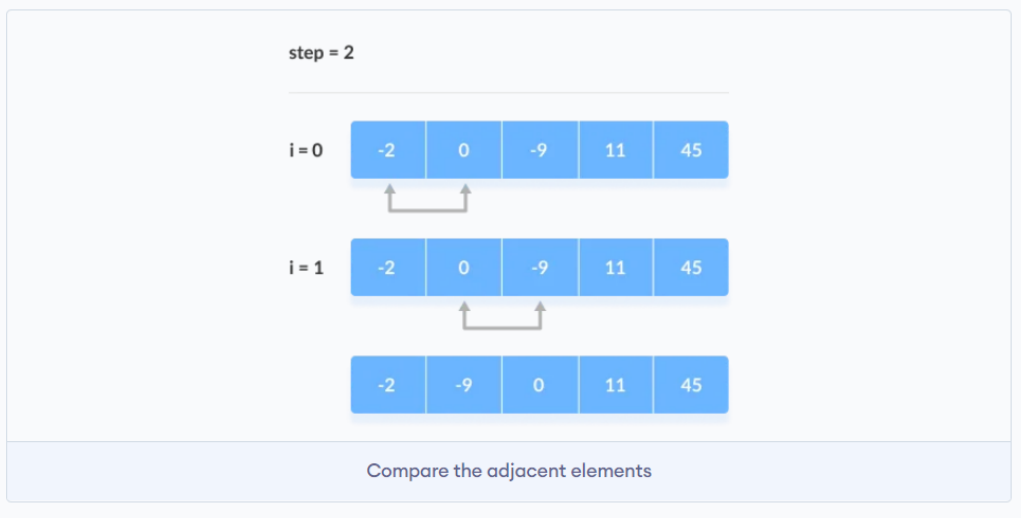
2. Remaining Iteration

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.



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

Bubble Sort Algorithm

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

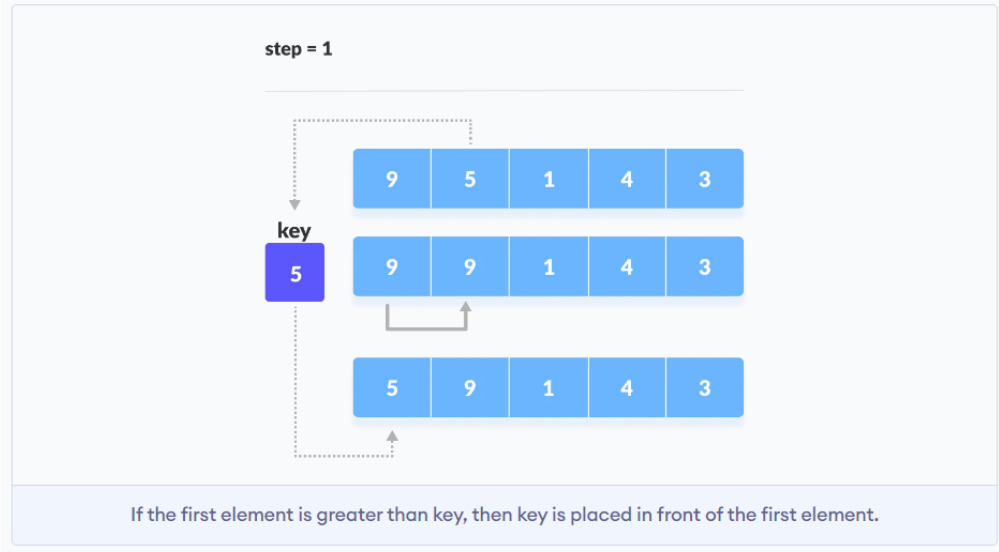
Insertion Sort

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.

Working of Insertion Sort
Suppose we need to sort the following array:

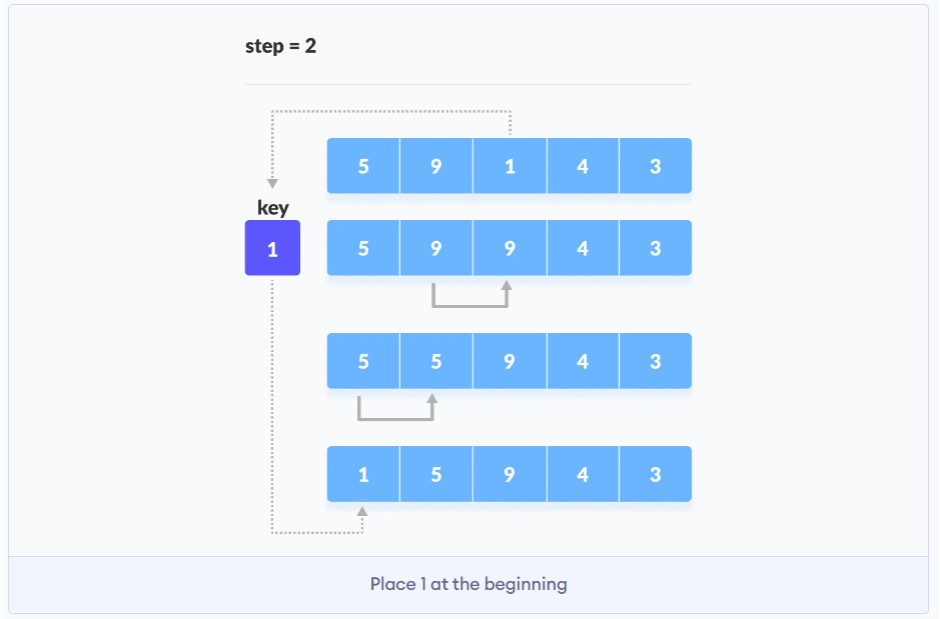
1. The first element in the array is assumed to be sorted. Take the second element and store it separately in a key.

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

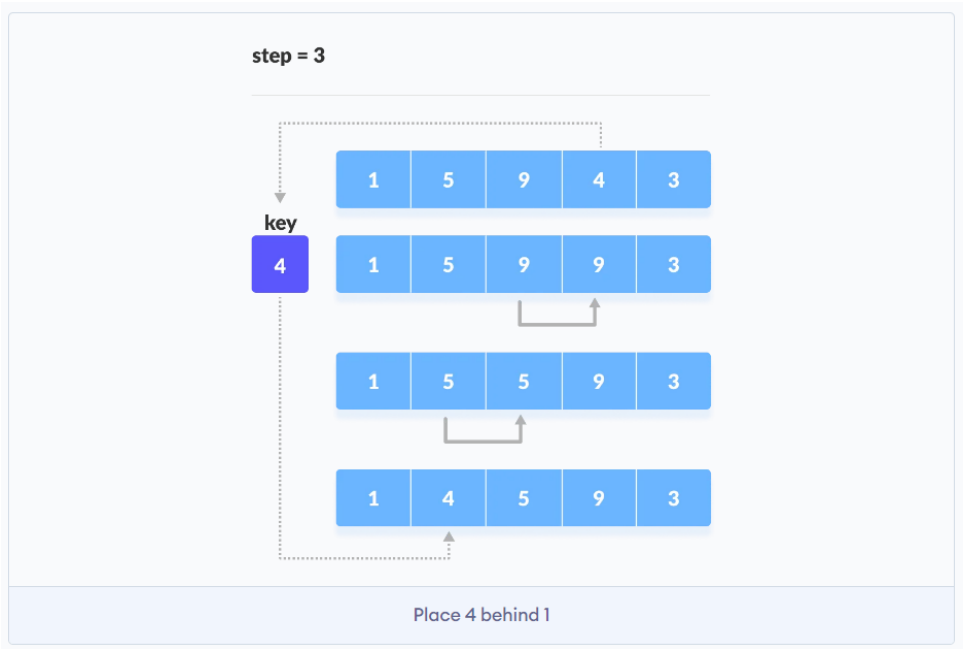


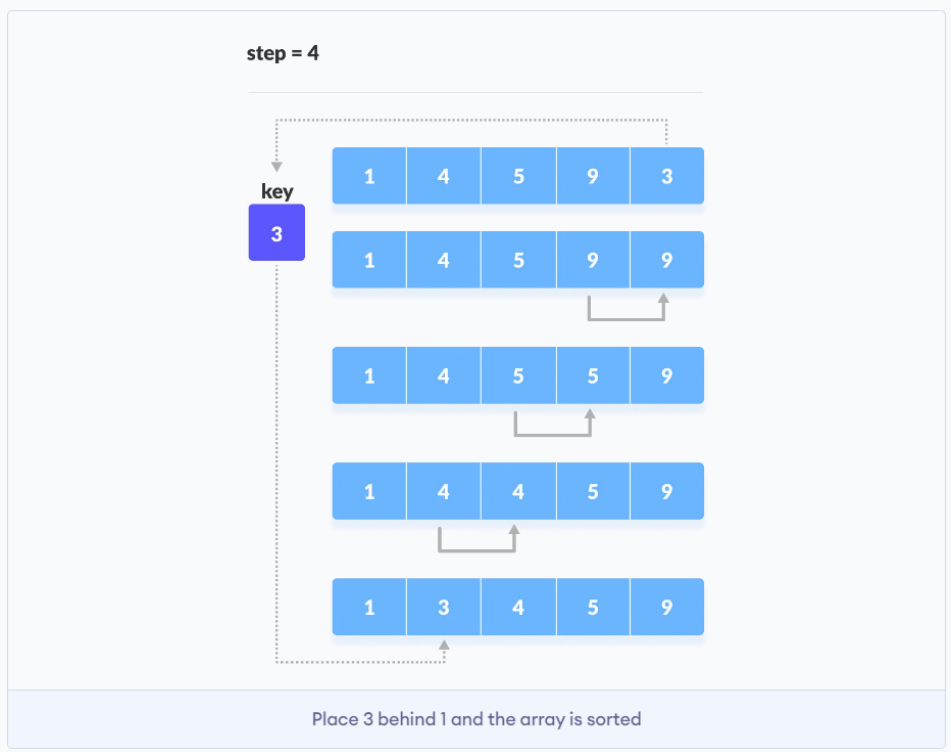
2. Now the first two elements are sorted.

   Take the third element and compare it with the elements on the left of it. Place 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.



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





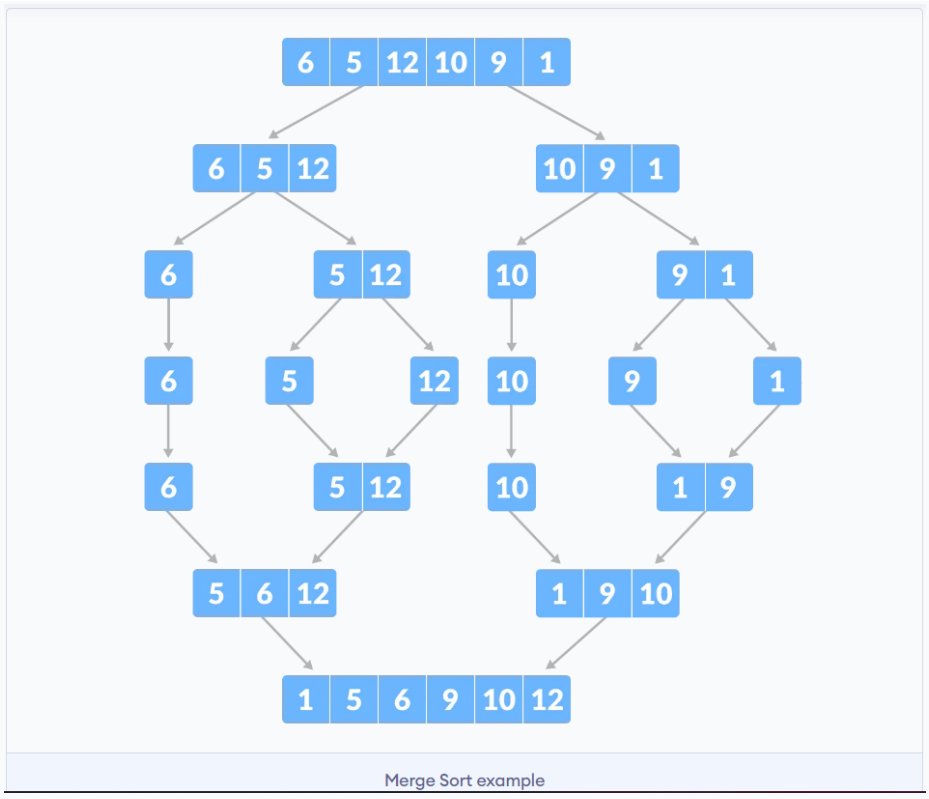
Insertion Sort Algorithm

```
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
```

Merge Sort

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 subsection 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].

Merge Sort Algorithm

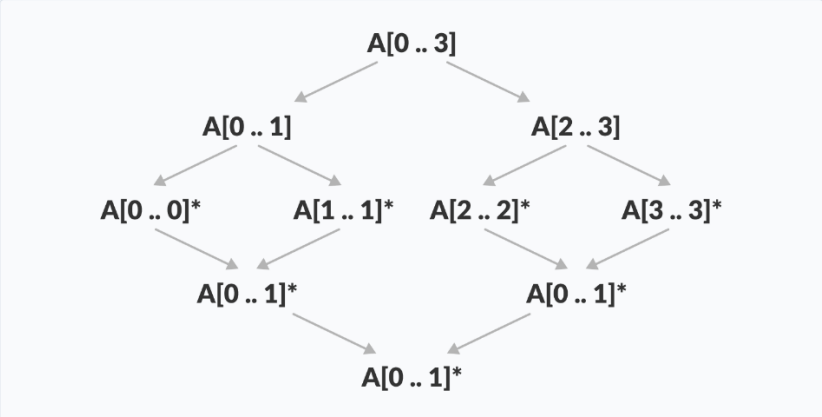
The MergeSort function repeatedly divides the array into two halves until we reach a stage where we try to perform MergeSort on a subarray of size 1 i.e. p == r.

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 the array with 1 element. After that, the merge function picks up the sorted subarrays and merges them to gradually sort the entire array.



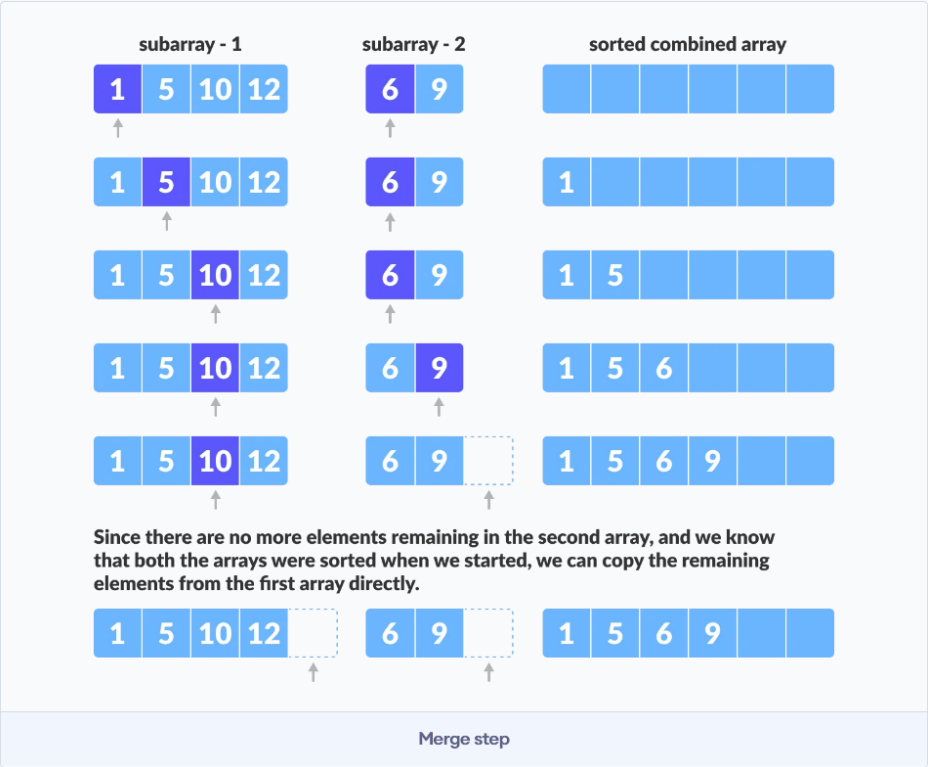
Merge sort in action

The merge Step of Merge Sort

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
```



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`

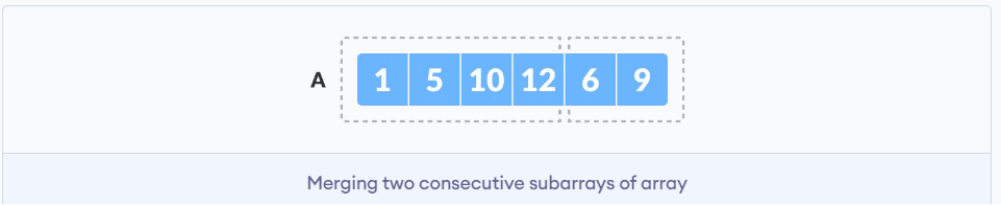
    1. `i` maintains current index of `L`, starting at 1
    2. `j` maintains current index of `M`, starting at 1
    3. `k` maintains the current index of `A[p..q]`, starting at `p`.
3. Until we reach the end of either `L` or `M`, pick the larger among the elements from `L` and `M` and place them in the 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++;
```

```
    // 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



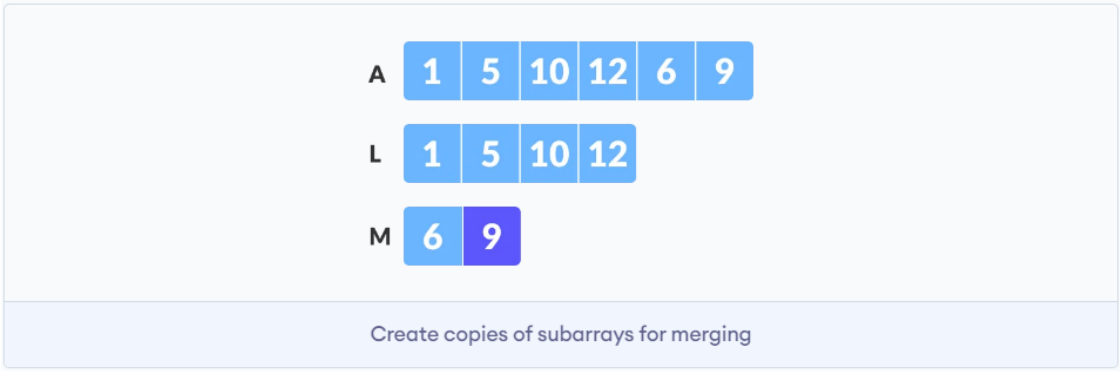
Merging two consecutive subarrays of array

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

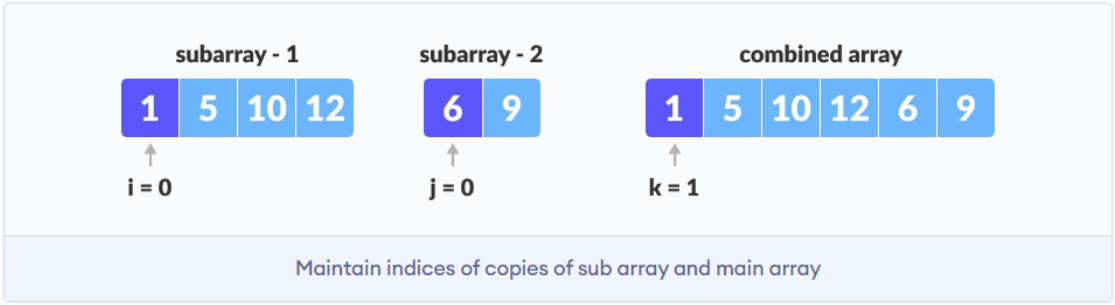
```
// Create L ← A[p..q] and M ← A[q+1..r]
int n1 = q - p + 1 = 3 - 0 + 1 = 4;
int n2 = r - q = 5 - 3 = 2;

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

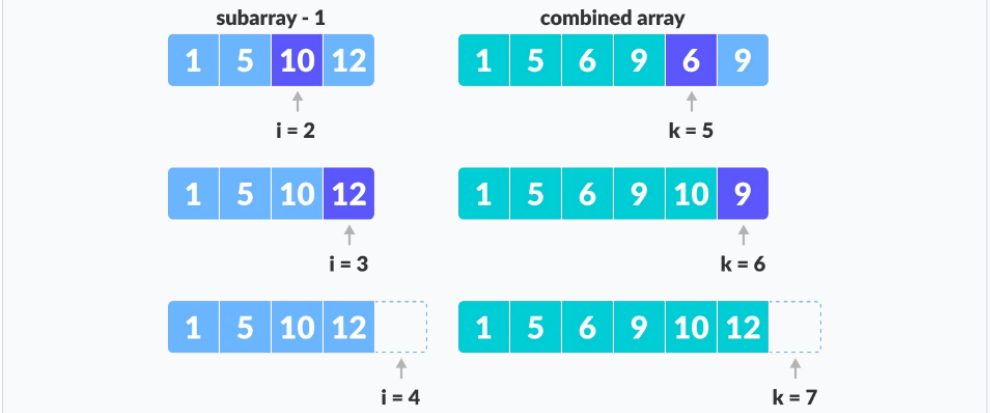
Step 3: Until we reach the end of either L or M, pick larger among elements L and M and place them in the correct position at A[p..r]

```
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

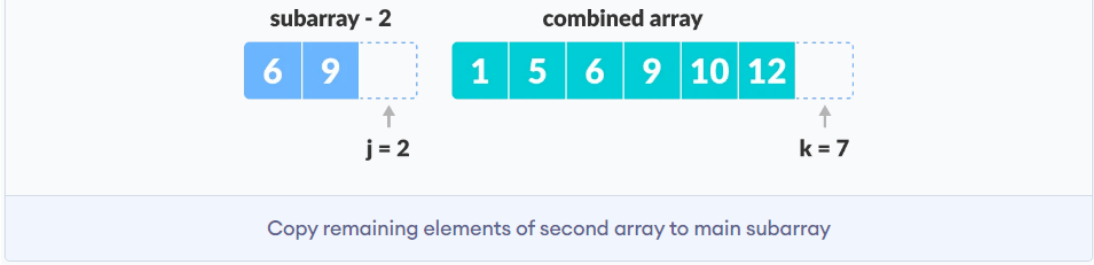
Step 4: When we run out of elements in either L or M, pick up the remaining elements and put in A[p..r]

```
    // 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.