Analysis of Algorithm

Amypo Technologies

Agenda

- Divide and Conquer
- Quick sort
- Merge sort

Divide and Conquer

- A divide and conquer algorithm is a strategy of solving a large problem by

1. breaking the problem into smaller sub-problems
2. solving the sub-problems, and
3. combining them to get the desired output.

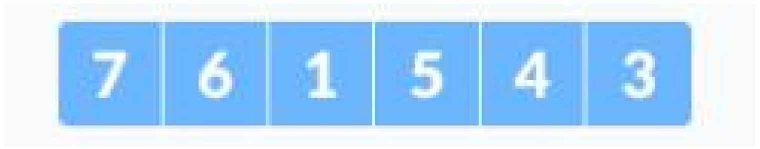
- To use the divide and conquer algorithm, **recursion** is used.

How Algorithms Work?

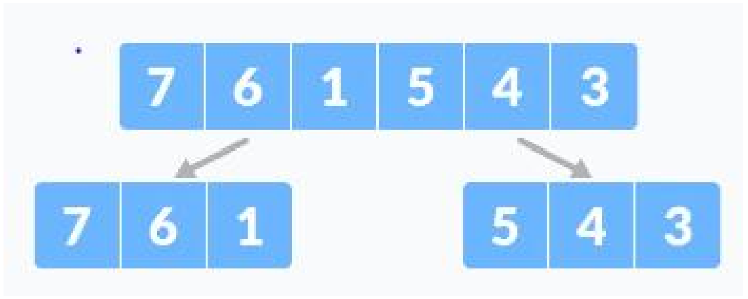
- **Divide**: Divide the given problem into sub-problems using recursion.
- **Conquer**: Solve the smaller sub-problems recursively. If the subproblem is small enough, then solve it directly.
- **Combine:** Combine the solutions of the sub-problems that are part of the recursive process to solve the actual problem.

Example

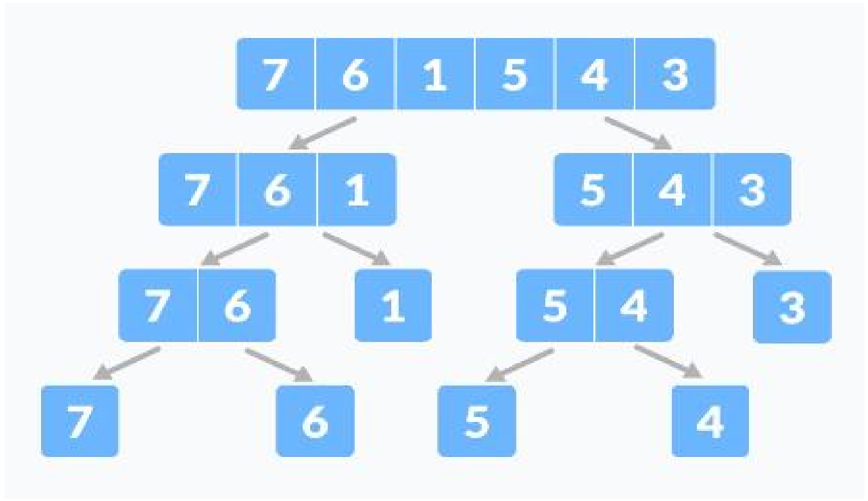
1.let the given array be:



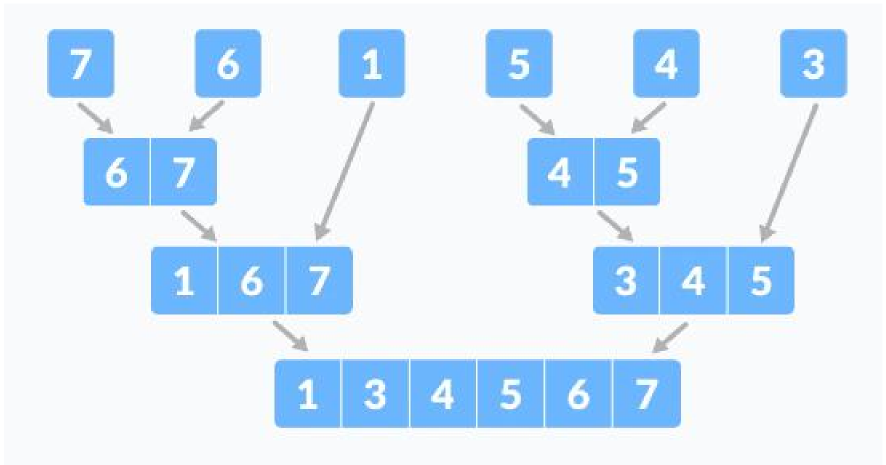
2.**Divide** the array into two halves.



- Again, divide each subpart recursively into two halves until you get individual elements.



- Now, combine the individual elements in a sorted manner. Here, **conquer** and **combine** steps go side by side.



Time Complexity

- The complexity of the divide and conquer algorithm is calculated using the master theorem.

$T(n) = aT(n/b) + f(n)$,

where,

n = size of input

a = number of subproblems in the recursion

n/b = size of each subproblem. All subproblems are assumed to have the same size.

$f(n)$ = cost of the work done outside the recursive call, which includes the cost of dividing the problem and cost of merging the solutions

Divide and Conquer Vs Dynamic approach

- The divide and conquer approach divides a problem into smaller subproblems; these subproblems are further solved recursively. The result of each subproblem is not stored for future reference, whereas, in a dynamic approach, the result of each subproblem is stored for future reference.

Divide and Conquer approach:

```
fib(n)
If n < 2, return 1
Else , return f(n - 1) + f(n -2)
```

Dynamic approach:

```
mem = []
fib(n)
If n in mem: return mem[n]
else,
If n < 2, f = 1
else , f =f(n - 1) + f(n -2)
mem[n] = f
return f
```

Advantages

- The complexity for the multiplication of two matrices using the naive method is $O(n^3)$, whereas using the divide and conquer approach (i.e. Strassen's matrix multiplication) is $O(n^{2.8074})$.
- This approach also simplifies other problems, such as the Tower of Hanoi.
- This approach is suitable for multiprocessing systems.
- It makes efficient use of memory caches.

Applications

- Binary Search
- Merge Sort
- Quick Sort
- Strassen's Matrix multiplication
- Karatsuba Algorithm

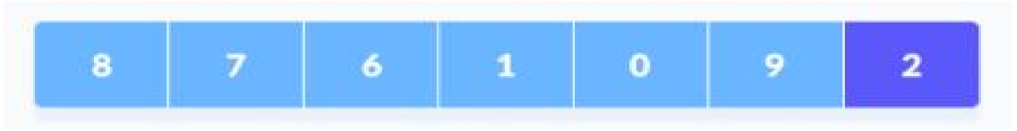
Quicksort Algorithm

- Quicksort is a sorting algorithm based on the **divide and conquer approach** where
- An array is divided into subarrays by selecting a **pivot element** (element selected from the array).
- The left and right subarrays are also divided using the same approach.
- This process continues until each subarray contains a single element.
- At this point, elements are already sorted. Finally, elements are combined to form a sorted array.
-

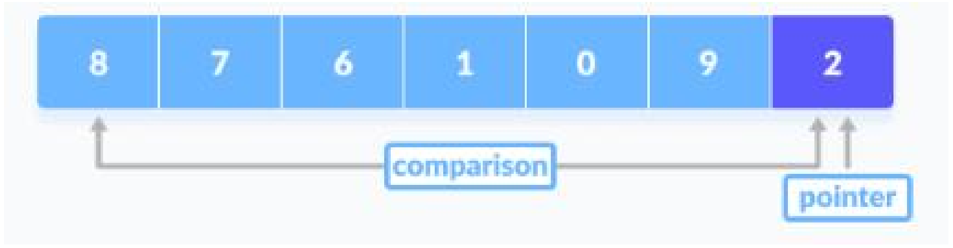
Working of Quicksort Algorithm

1. Select the Pivot Element

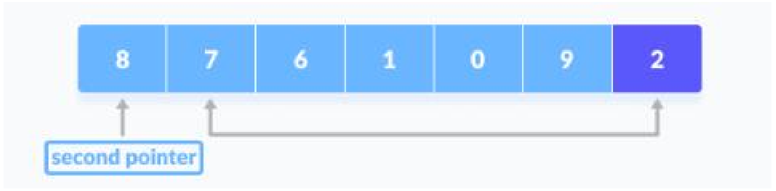
- There are different variations of quicksort where the pivot element is selected from different positions.
- Here, we will be selecting the rightmost element of the array as the pivot element.



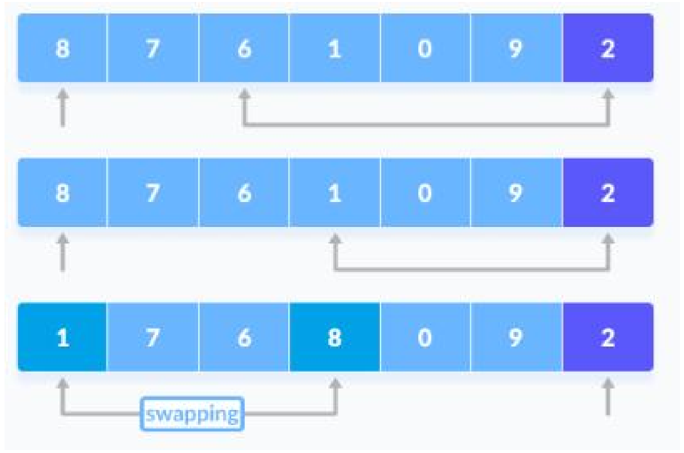
- Here's how we rearrange the array:
- A pointer is fixed at the pivot element. The pivot element is compared with the elements beginning from the first index.



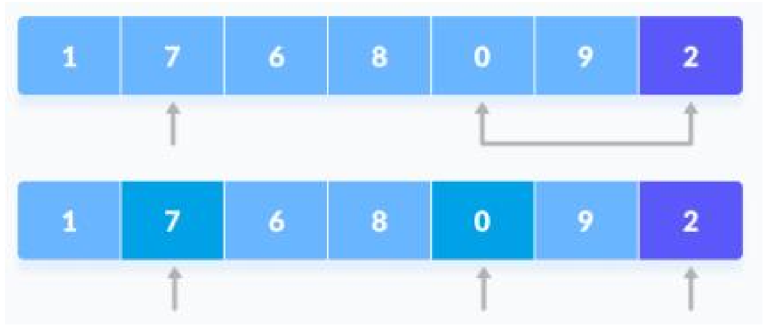
2. If the element is greater than the pivot element, a second pointer is set for that element.



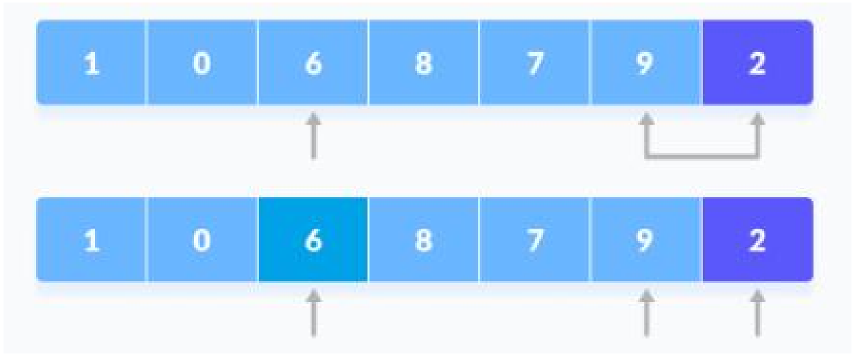
3. Now, pivot is compared with other elements. If an element smaller than the pivot element is reached, the smaller element is swapped with the greater element found earlier.



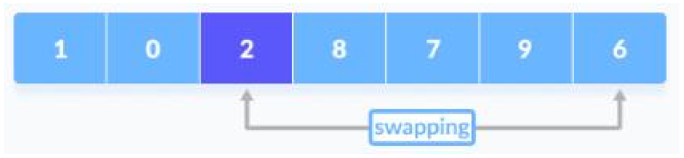
4. Again, the process is repeated to set the next greater element as the second pointer. And, swap it with another smaller element.



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

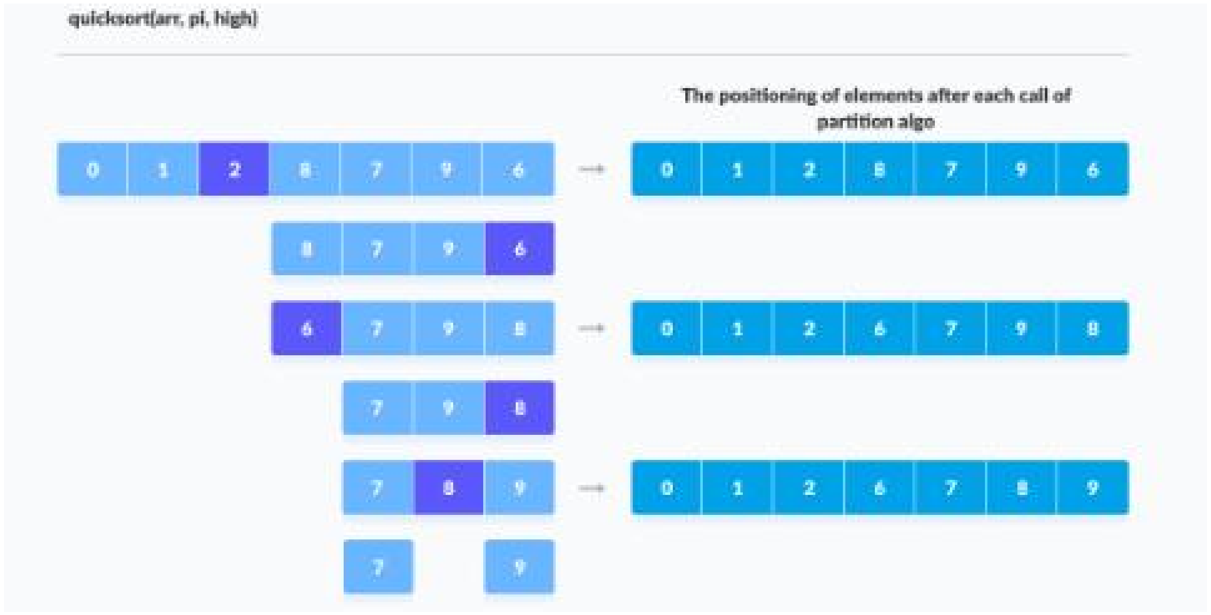


6. Finally, the pivot element is swapped with the second pointer.



3. Divide Subarrays

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



Quick Sort 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
```

Example

```
#include <iostream>
using namespace std;
void swap(int *a, int *b) {
  int t = *a;
  *a = *b;
  *b = t;          }
void printArray(int array[], int size) {
  int i;
  for (i = 0; i < size; i++)
    cout << array[i] << " ";
  cout << endl;              }
int partition(int array[], int low, int high) {
int pivot = array[high];
int i = (low - 1);
for (int j = low; j < high; j++) {
    if (array[j] <= pivot) {
i++;
swap(&array[i], &array[j]);
    }  }
swap(&array[i + 1], &array[high]);
return (i + 1);
}
```

```
void quickSort(int array[], int low, int high) {
  if (low < high) {
int pi = partition(array, low, high);
quickSort(array, low, pi - 1);
quickSort(array, pi + 1, high);
  }
}
int main() {
  int data[] = {8, 7, 6, 1, 0, 9, 2};
  int n = sizeof(data) / sizeof(data[0]);
  cout << "Unsorted Array: \n";
  printArray(data, n);
  quickSort(data, 0, n - 1);
  cout << "Sorted array in ascending order: \n";
  printArray(data, n);
}
```

Quicksort Complexity

Time Complexity	
Best	O(n*log n)
Worst	$O(n^2)$
Average	O(n*log n)
Space Complexity	O(log n)
Stability	No

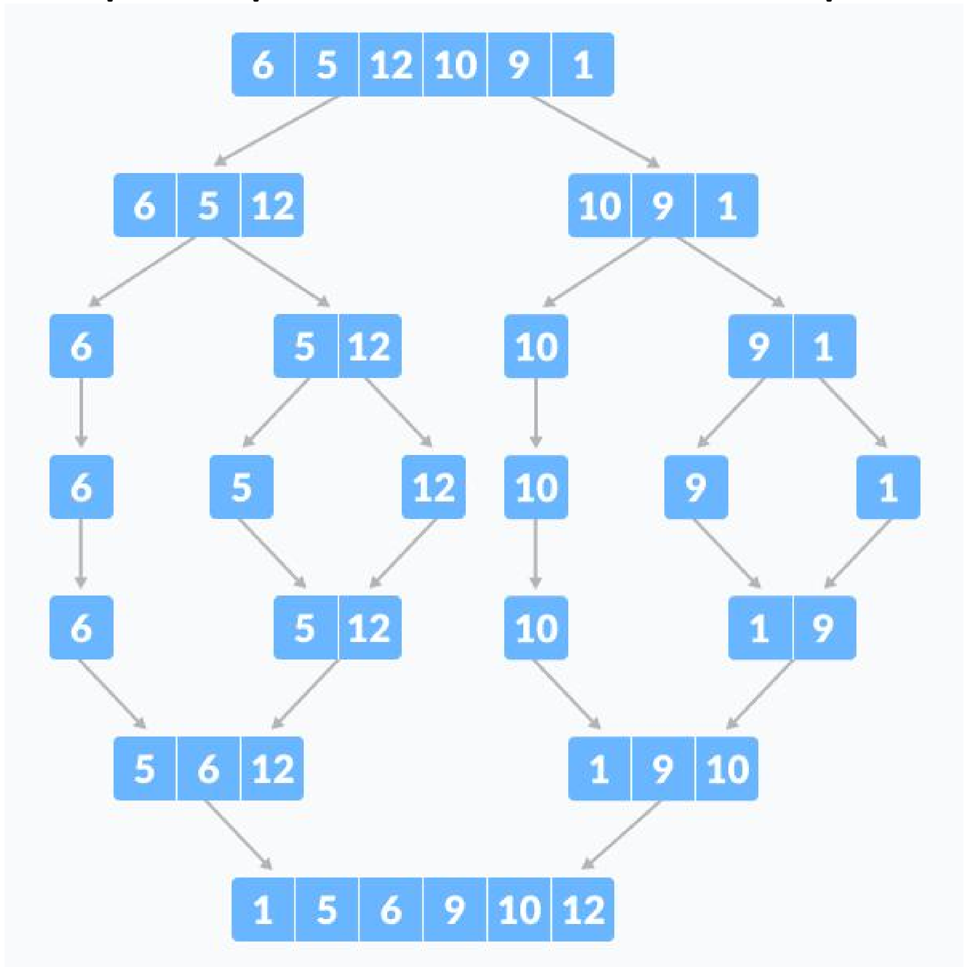
Quicksort Applications

Quicksort algorithm is used when

- the programming language is good for recursion
- time complexity matters
- space complexity matters

Merge Sort Algorithm

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



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)
```

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:

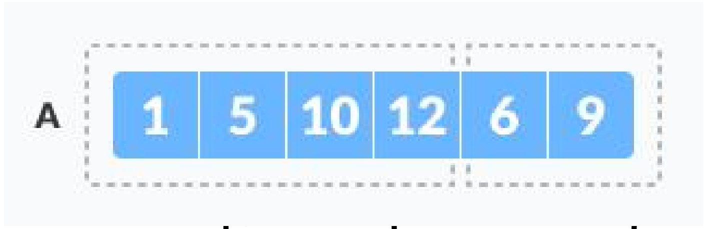
1. Create copies of the subarrays L <- A[p..q] and M <- A[q+1..r].
2. Create three pointers i, j and k
   - i maintains current index of L, starting at 1
   - j maintains current index of M, starting at 1
   - k maintains the current index of A[p..q], starting at p.
3. Until we reach the end of either 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]

Example

```
void merge(int arr[], int p, int q, int 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];
int i, j, k;
   i = 0;
   j = 0;
   k = p;
while (i < n1 && j < n2) {
      if (L[i] <= M[j]) {
         arr[k] = L[i];
         i++;
      } else {
         arr[k] = M[j];
         j++;          }
      k++;        }
while (i < n1) {
      arr[k] = L[i];
      i++;
      k++;       }
while (j < n2) {
      arr[k] = M[j];
      j++;
      k++;       }
}
```

Merge( ) Function Explained Step-By-Step

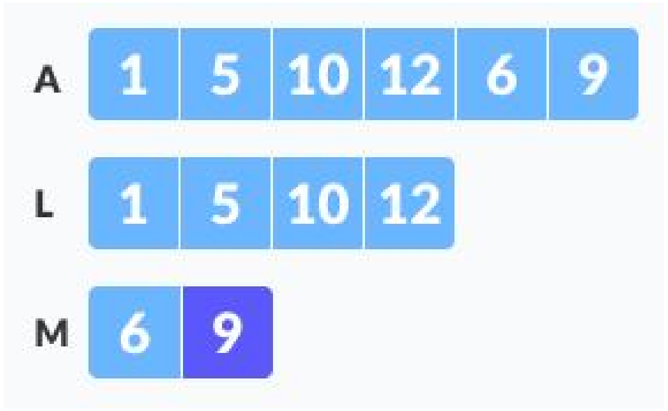
- A lot is happening in this function, so let's take an example to see how this would work.
- As usual, a picture speaks a thousand words.



- Merging two consecutive subarrays of arrayThe 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)

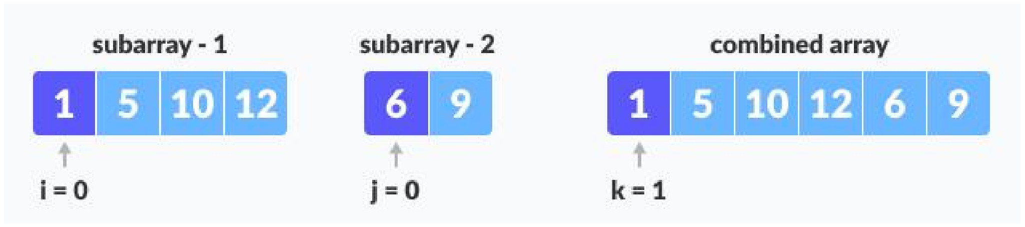
Step 1: Create duplicate copies of sub-arrays to be sorted

```
// 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]
```



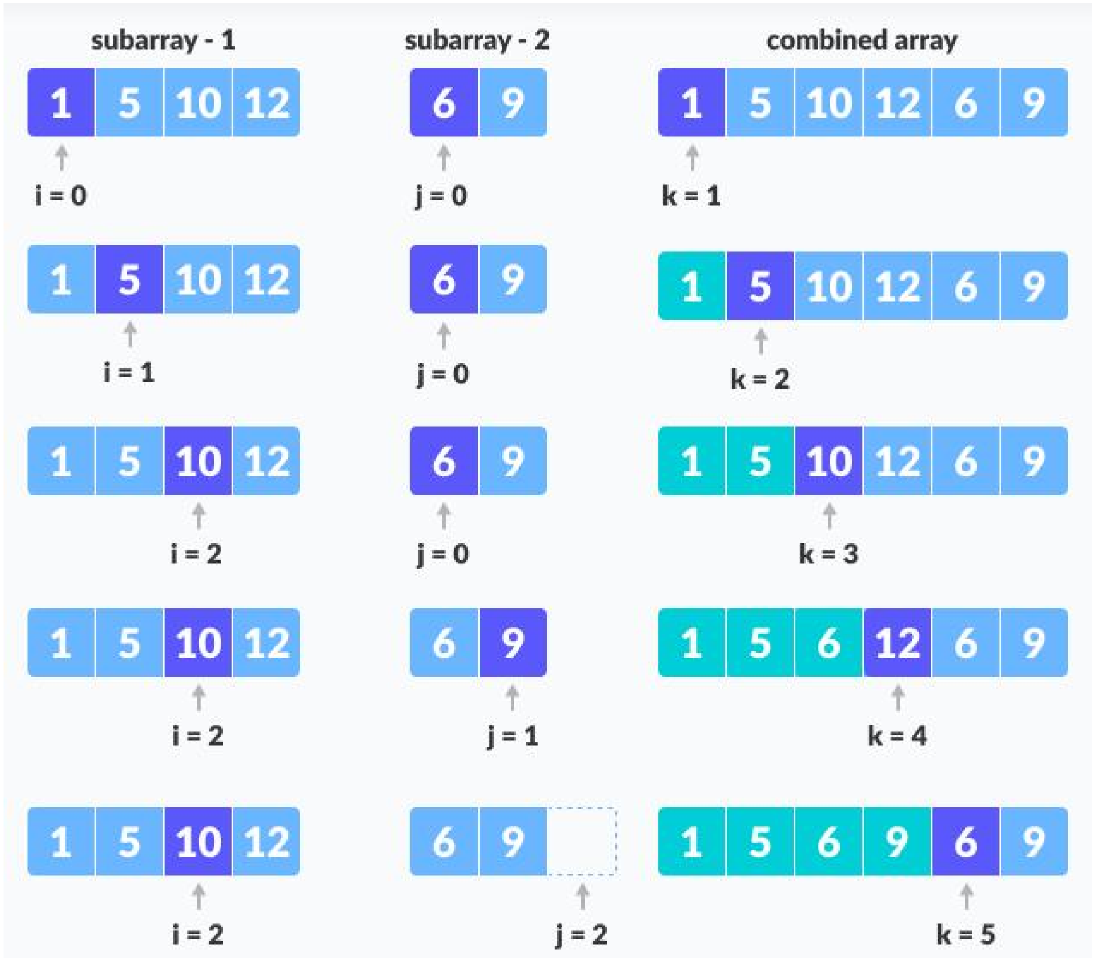
Step 2: Maintain current index of sub-arrays and main array

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



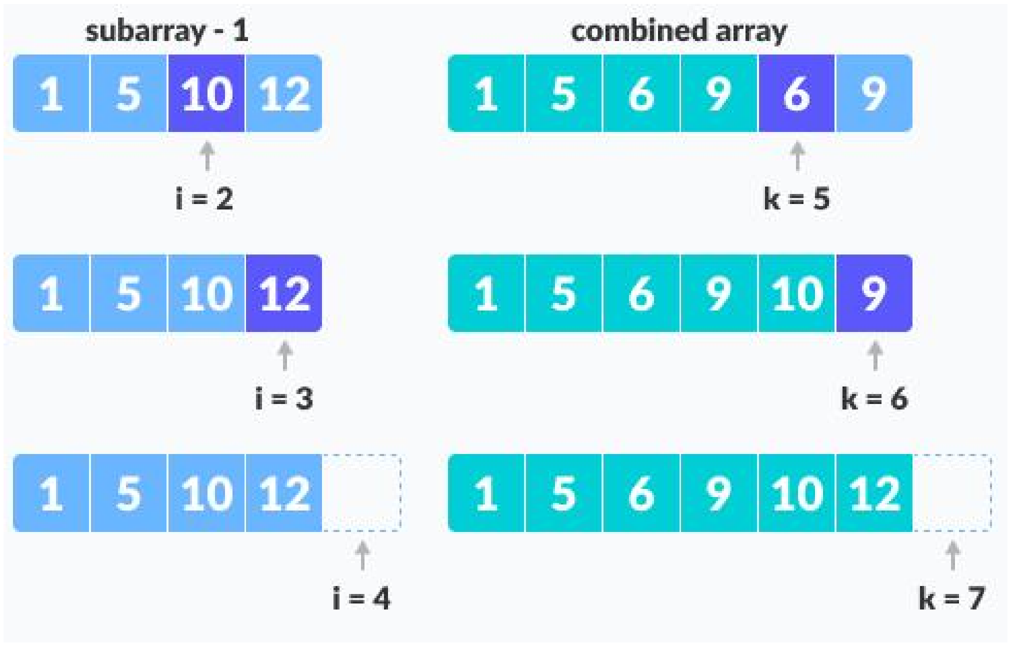
Step 3: Until we reach the end of either L or M, pick larger among elements L and M and place them in the correct position at A[p..r]

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



- **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++;
}
```



Example

```
#include <iostream>
using namespace std;
void merge(int arr[], int p, int q, int 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];
int i=0, j=0, k=p;
while (i < n1 && j < n2) {
   if (L[i] <= M[j]) {
     arr[k] = L[i];
     i++;
   } else {
     arr[k] = M[j];
     j++;          }
   k++;          }
while (i < n1) {
   arr[k] = L[i];
   i++;
   k++;          }
while (j < n2) {
   arr[k] = M[j];
   j++;
   k++;       }      }
```

```
void mergeSort(int arr[], int l, int r) {
 if (l < r) {
int m = l + (r - l) / 2;
   mergeSort(arr, l, m);
   mergeSort(arr, m + 1, r);
merge(arr, l, m, r);
 }
}
void printArray(int arr[], int size) {
 for (int i = 0; i < size; i++)
   cout << arr[i] << " ";
 cout << endl;
}
int main() {
 int arr[] = {6, 5, 12, 10, 9, 1};
 int size = sizeof(arr) / sizeof(arr[0]);
 mergeSort(arr, 0, size - 1);
 cout << "Sorted array: \n";
 printArray(arr, size);
 return 0;
}
```

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

Merge Sort Applications

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

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).