­­



Submitted to

:

Sir Asim Rehan

***BS***

***-***

***Software Engineering***

***3***

***rd***

***-***

***E***

Title: Merge Sort

DSA

Hamza Mehmood

Ahad Ali

Jibran Naeem

Saad Raja



**National University of Modern Languages**

DATA STRUCTURES & ALGORITHM

# MERGE SORT

Merge Sort is one of the most popular Sorting Algorithm 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.



# Divide & Conquer

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

Suppose we had to sort an array A. A sub-problem would be to sort a sub-section of this array starting at index p 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:**

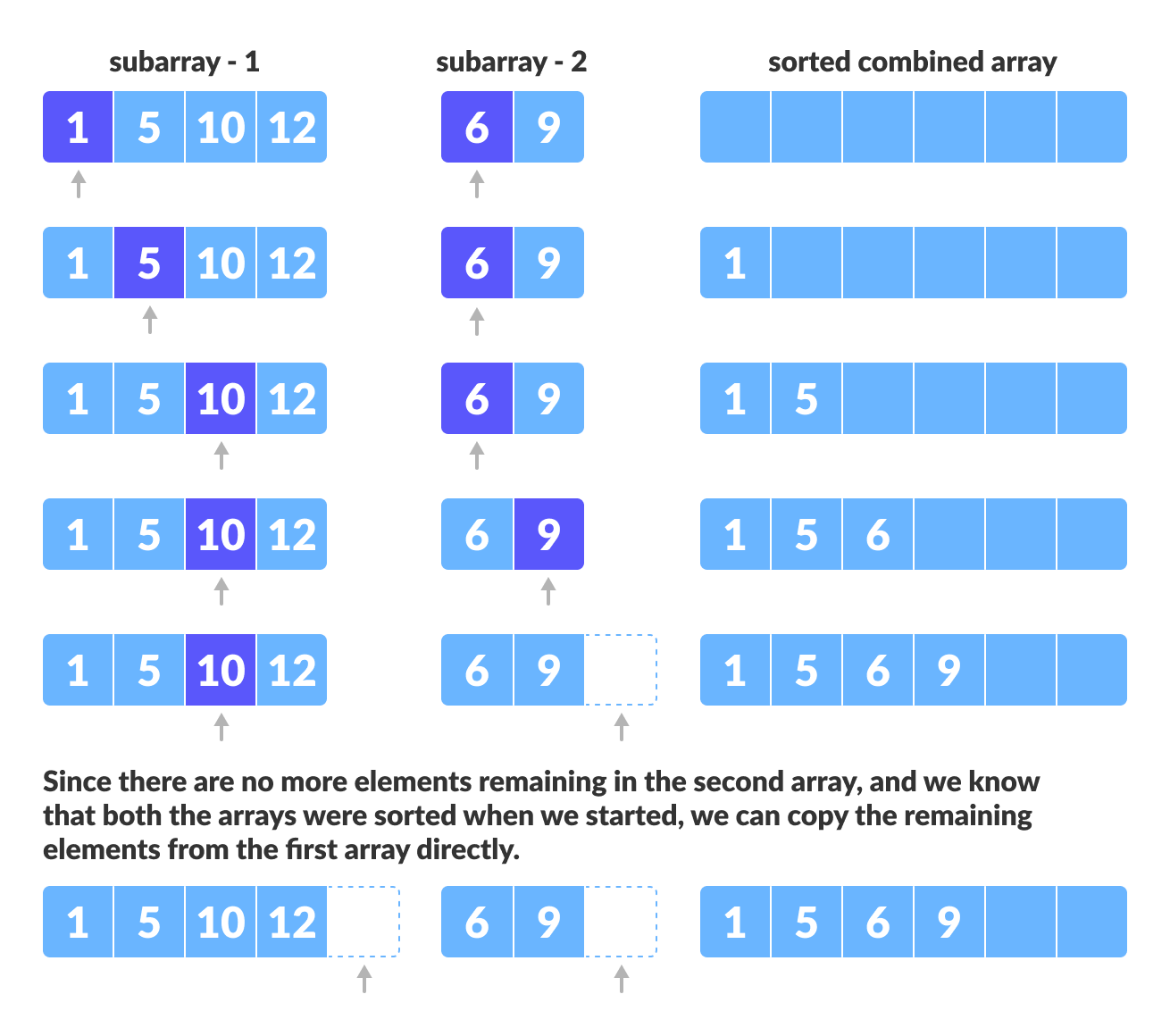
In the conquer step, we try to sort both the subarrays A[p..q] and A[q+1, r]. If we haven't yet reached the base case, we again divide both these subarrays and try to sort them.

**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.



CODE

#include <iostream>

Using namespace std;

void printArray(int \*A, int n)

{

for (int i = 0; i < n; i++)

{

Cout<<A[i];

}

Cout<<"\n";

}

void merge(int A[], int mid, int low, int high)

{

int i, j, k, B[100];

i = low;

j = mid + 1;

k = low;

while (i <= mid && j <= high)

{

if (A[i] < A[j])

{

B[k] = A[i];

i++;

k++;

}

else

{

B[k] = A[j];

j++;

k++;

}

}

while (i <= mid)

{

B[k] = A[i];

k++;

i++;

}

while (j <= high)

{

B[k] = A[j];

k++;

j++;

}

for (int i = low; i <= high; i++)

{

A[i] = B[i];

}

}

void mergeSort(int A[], int low, int high){

int mid;

if(low<high){

mid = (low + high) /2;

mergeSort(A, low, mid);

mergeSort(A, mid+1, high);

merge(A, mid, low, high);

}

}

int main()

{

int A[] = {9, 1, 4, 14, 4, 15, 6};

int n = 7;

printArray(A, n);

mergeSort(A, 0, 6);

printArray(A, n);

return 0;

}

# Complexity Analysis

We know that in order to perform sorting using merge sort, we first divide the array into two equal halves. This is represented by “log n” which is a logarithmic function and the number of steps taken is log (n+1) at the most.

Next to find the middle element of the array we require single step i.e. O(1).

Then to merge the sub-arrays into an array of n elements, we will take O (n) amount of running time.

Thus the total time to perform merge sort will be n (log n+1), which gives us the time complexity of O (n\*logn).

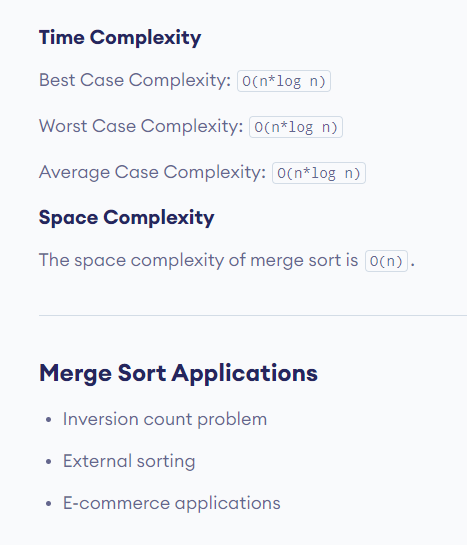
|  |  |
| --- | --- |
| Worst case time complexity | O(n\*log n) |
| Best case time complexity | O(n\*log n) |
| Average time complexity | O(n\*log n) |
| Space complexity | O(n) |

The time complexity for merge sort is the same in all three cases (worst, best and average) as it always divides the array into sub-arrays and then merges the sub-arrays taking linear time.

Merge sort always takes an equal amount of space as unsorted arrays. Hence when the list to be sorted is an array, merge sort should not be used for very large arrays. However, merge sort can be used more effectively for linked lists sorting.

# Conclusion

Merge sort uses the “divide and conquer” strategy which divides the array or list into numerous sub arrays and sorts them individually and then merges into a complete sorted array.

Merge sort performs faster than other sorting methods and also works efficiently for smaller and larger arrays likewise.