Sorting Algorithms

Content

Merge Sort

About

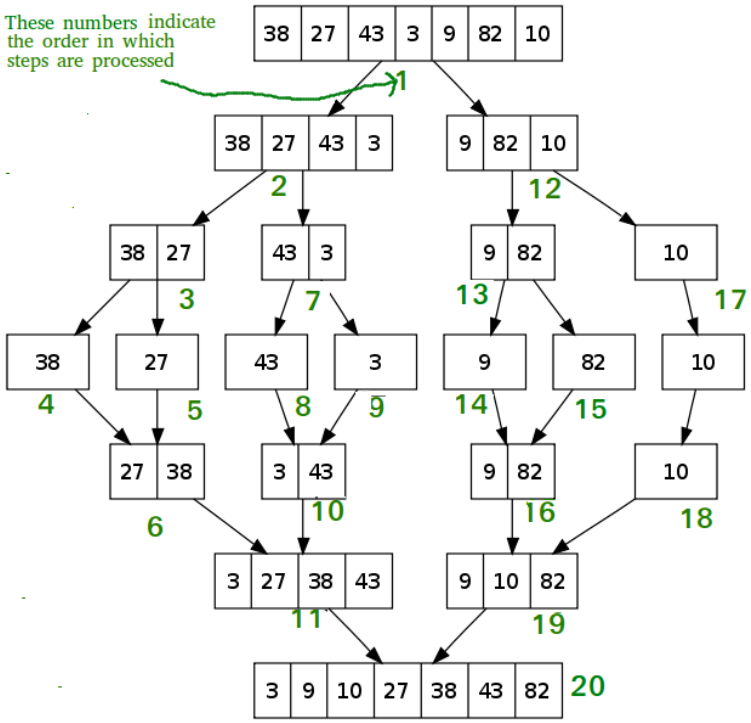
Like QuickSort, Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted subarrays into one.

PseudoAlgorithm

MergeSort(arr[], l, r)

If r > l

1. Find the middle point to divide the array into two halves:
   middle m = l+ (r-l)/2
2. Call mergeSort for first half:
   Call mergeSort(arr, l, m)
3. Call mergeSort for second half:
   Call mergeSort(arr, m+1, r)
4. Merge the two halves sorted in step 2 and 3:
   Call merge(arr, l, m, r)



Implementation

https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Sorting/Codes/MergeSort.cpp

Time Complexity - Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

$T(n) = 2T(n/2) + \theta(n)$

The above recurrence can be solved either using the Recurrence Tree method or the Master method. It falls in case II of the Master Method and the solution of the recurrence is $\theta(n\log(n))$. The time complexity of Merge Sort is $\theta(n\text{Log}n)$ in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.

Auxiliary Space - O(n)

Algorithmic Paradigm - Divide and Conquer

Sorting In Place - No in a typical implementation

Stable - Yes

Selection Sort

About

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from the unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

1. The subarray is already sorted.
2. The remaining subarray is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

The following example explains the above steps:

arr[] = 64 25 12 22 11

// Find the minimum element in arr[0...4]
// and place it at beginning
11 25 12 22 64

// Find the minimum element in arr[1...4]
// and place it at beginning of arr[1...4]
11 **12** 25 22 64

// Find the minimum element in arr[2...4]

// and place it at beginning of arr[2...4]

11 12 **22** 25 64

// Find the minimum element in arr[3...4]

// and place it at beginning of arr[3...4]

11 12 22 **25** 64

Implementation

https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Sorting/Codes/SelectionSort.cpp

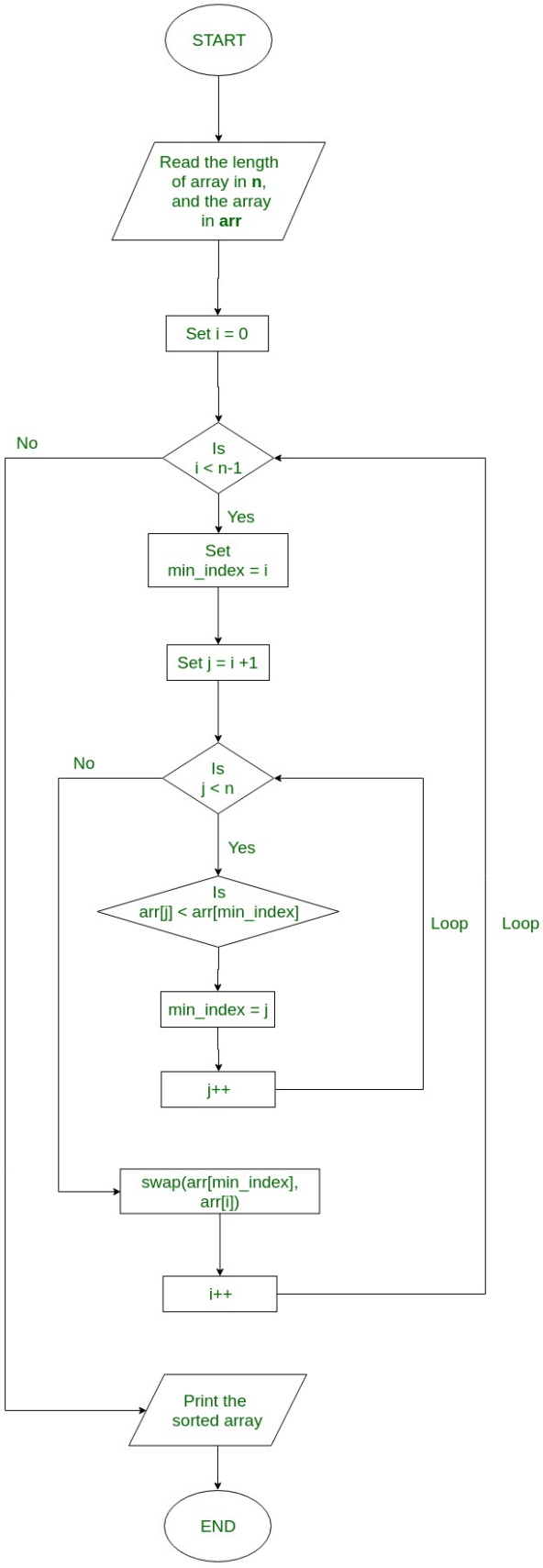
Time Complexity - O(n^2) as there are two nested loops.

Auxiliary Space - O(1)

The good thing about selection sort is it never makes more than O(n) swaps and can be useful when memory writing is a costly operation.

Stability - The default implementation is not stable. However, it can be made stable.

In-Place - Yes, it does not require extra space.


START
Read the length of array in n, and the array in arr
Set i = 0
No
Is i < n-1
Yes
Set min_index = i
Set j = i +1
No
Is j < n
Yes
Is arr[j] < arr[min_index]
Loop
Loop
min_index = j
j++
swap(arr[min_index], arr[i])
i++
Print the sorted array
END