Merge Sort

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being **O(n log n)**, it is one of the most used and approached algorithms. Merge sort first divides the array into equal halves and then combines them in a sorted manner.

What Is a Divide and Conquer Algorithm?

Divide-and-conquer recursively solves subproblems; each subproblem must be smaller than the original problem, and each must have a base case. A divide-and-conquer algorithm has three parts:

Divide: Divide the list or array recursively into two halves until it can no more be divided.

Conquer: Each subarray is sorted individually using the merge sort algorithm.

Merge: The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.

Merge Sort Algorithm

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

```
MERGE(A, p, q, r)
n1 = q - p + 1
n2 = r - q
let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i - 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

Example: A=[2, 4, 5, 7, 1, 2, 3, 6]

Quick Sort

Quick Sort is a sorting algorithm based on the Divide and Conquer algorithm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

The following procedure implements quicksort.

```
QUICKSORT(A, p, r)
if p < r
    then q ← PARTITION(A, p, r)
         QUICKSORT(A, p, q - 1)
         QUICKSORT(A, q + 1, r)
```

To sort an entire array A, the initial call is QUICKSORT(A, 1, *length*[A]).

Partitioning the array

The key to the algorithm is the PARTITION procedure, which rearranges the subarray $A[p$ □ $r]$ in place.

```
PARTITION(A, p, r)
x ← A[r]
i ← p - 1
for j ← p to r - 1
     do if A[j] ≤ x
           then i ← i + 1
                exchange A[i] ↔ A[j]
exchange A[i + 1] ↔ A[r]
return i + 1
```

Example: 2, 8, 7 , 1, 3, 5, 6, 4

Heap Sort

```
HEAPSORT(A)
  BUILD-MAX-HEAP(A)
  for i = A.length downto 2
      exchange A[1] with A[i]
      A.heap-size = A.heap-size − 1
      MAX-HEAPIFY(A, 1)
```

// Implementation of Heap Sort in C

```
#include <stdio.h>

// Function to swap the position of two elements in an array
void swap(int *a, int *b) {
  int tempvar = *a;
  *a = *b;
  *b = tempvar;
}

void heapify(int arr[], int n, int i) {
  // Finding the greatest among root, leftSide child, and rightSide child of the tree
  int greatest = i;
  int leftSide = 2 * i + 1;
  int rightSide = 2 * i + 2;
```

```
  if (leftSide < n && arr[leftSide] > arr[greatest])
    greatest = leftSide;

  if (rightSide < n && arr[rightSide] > arr[greatest])
    greatest = rightSide;

  // Swap and continue heapifying if the root is not the greatest
  if (greatest != i) {
    swap(&arr[i], &arr[greatest]);
    heapify(arr, n, greatest);
  }
}

// Main function
void heapSort(int arr[], int n) {
  // Build max heap
  for (int i = n / 2 - 1; i >= 0; i--)
    heapify(arr, n, i);



  for (int i = n - 1; i >= 0; i--) {
    swap(&arr[0], &arr[i]);

    heapify(arr, i, 0);
  }
}
```

```
// Printing the array
void printArray(int arr[], int n) {
  for (int i = 0; i < n; ++i)
    printf("%d ", arr[i]);
  printf("\n");
}


int main() {
  int arr[] = {1, 12, 9, 5, 6, 10};
  int n = sizeof(arr) / sizeof(arr[0]);

  heapSort(arr, n);

  printf("Sorted array is \n");
  printArray(arr, n);
}
```