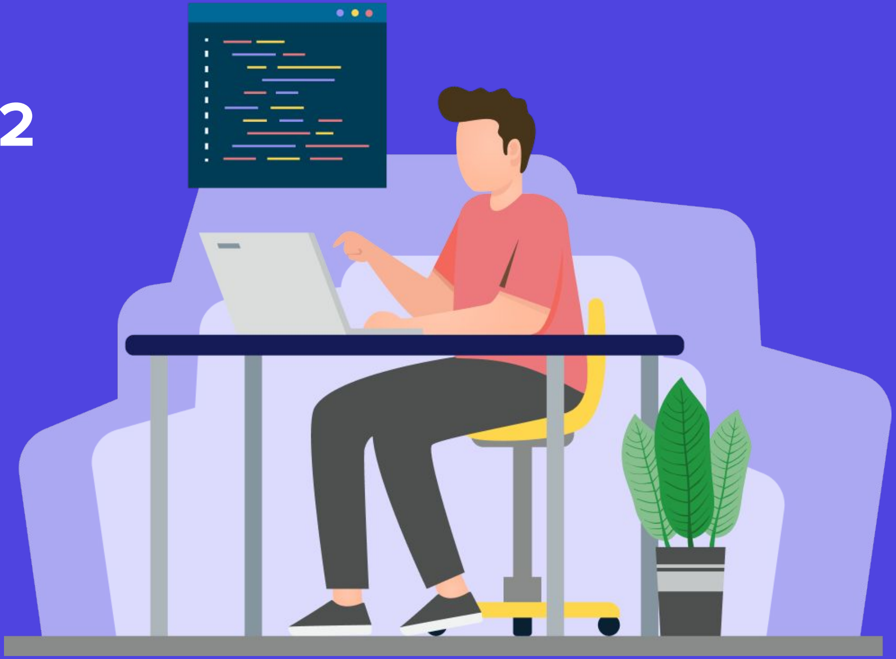
Algorithms: Merge sort -Part-2



Relevel
by Unacademy

What does it take?

As we would be concentrating on hands-on advanced problem-solving in this session, please revise the concepts covered in the last sessions:

- basic knowledge of sorting
- properties of sorting algorithms like stability, in-place sorting, etc.

List of Problems Involved

- Merge Sort Algorithm
- Problem-Solving On Merge Sort
    - Median of Arrays
    - Sort Array
    - Inversion Count
    - Double Inversion Count
    - Minimum and Maximum Element

Merge Sort

- We will divide the array into two halves, and we will assume that we will recursively sort the left and right half; then we will be in a state to merge the two sorted arrays.

Median of Arrays

Problem Statement
Given 2 sorted arrays. Your task is to find the median of both arrays

Input -
arr1 = [11, 12, 25, 26, 38];
arr2 = [2, 23, 17, 20, 45];

Output -
18.5

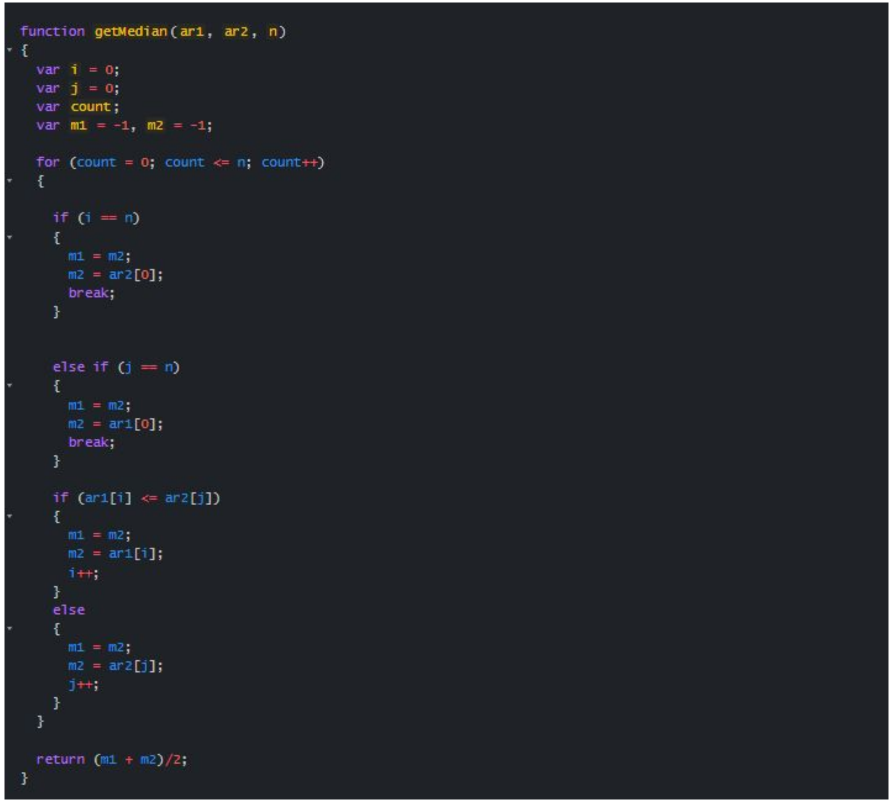
Steps -
1. Since this is related to finding the median, we will use merge sort here.
2. We need to keep track of the count while merging given arrays
3. If count = n where 2n = number of elements, calculate the average of elements at index n-1 and n

Median of Arrays

Code Link - https://jsfiddle.net/3rLmjca6/

Time Complexity: O(m+n) where m and n are number of elements in arrays

Space Complexity: O(1)



```
function getMedian(ar1, ar2, n)
{
  var i = 0;
  var j = 0;
  var count;
  var m1 = -1, m2 = -1;

  for (count = 0; count <= n; count++)
  {

    if (i == n)
    {
      m1 = m2;
      m2 = ar2[0];
      break;
    }


    else if (j == n)
    {
      m1 = m2;
      m2 = ar1[0];
      break;
    }

    if (ar1[i] <= ar2[j])
    {
      m1 = m2;
      m2 = ar1[i];
      i++;
    }
    else
    {
      m1 = m2;
      m2 = ar2[j];
      j++;
    }
  }

  return (m1 + m2)/2;
}
```

Sort Array

Problem Statement

Given an array having 2 halves already sorted. Your task is to sort it

For example -

Input -

[1,4,5,2,3,6]

Output -

[1,2,3,4,5,6]

Steps -

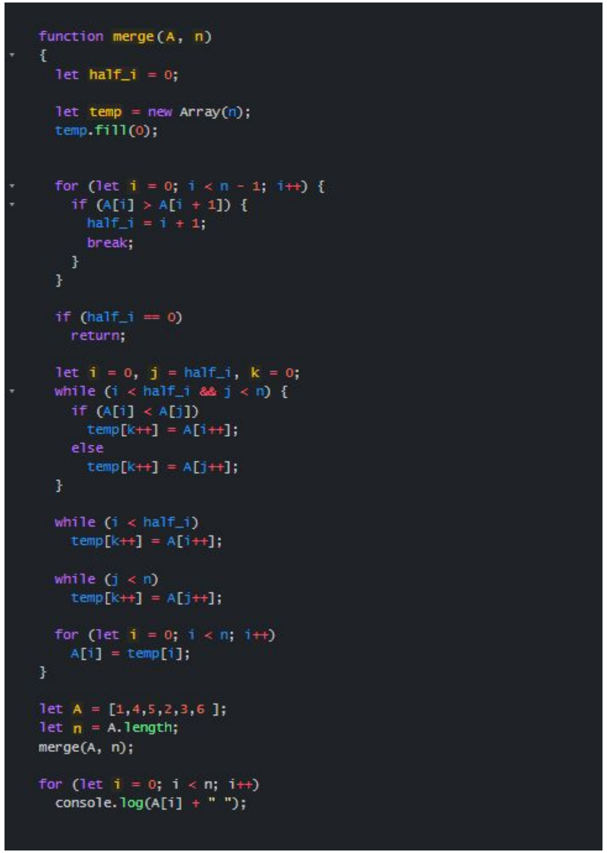
1. Naive approach is to use the inbuilt sort function of Arrays
2. Here, we will use an auxiliary array to save our sorted array after merging similar to merge sort algorithm

Sort Array

Code Link - https://jsfiddle.net/evu81tky/

Time Complexity: O(n)

Space Complexity: O(n)



```
function merge(A, n)
{
  let half_i = 0;

  let temp = new Array(n);
  temp.fill(0);


  for (let i = 0; i < n - 1; i++) {
    if (A[i] > A[i + 1]) {
      half_i = i + 1;
      break;
    }
  }

  if (half_i == 0)
    return;

  let i = 0, j = half_i, k = 0;
  while (i < half_i && j < n) {
    if (A[i] < A[j])
      temp[k++] = A[i++];
    else
      temp[k++] = A[j++];
  }

  while (i < half_i)
    temp[k++] = A[i++];

  while (j < n)
    temp[k++] = A[j++];

  for (let i = 0; i < n; i++)
    A[i] = temp[i];
}

let A = [1,4,5,2,3,6 ];
let n = A.length;
merge(A, n);

for (let i = 0; i < n; i++)
  console.log(A[i] + " ");
```

Inversion Count

Let A[0...n - 1] be an array of n distinct positive integers. If i < j and A[i] > A[j] then the pair (i, j) is called an inversion of A. Given n and an array A, your task is to find the number of inversions of A.

Solution

Brute force: In the brute force algorithm we can try to find all the n^2 pairs from the given array and then check which of them follows the property of inversion.
Time complexity: O(n^2)
Space complexity: O(1)

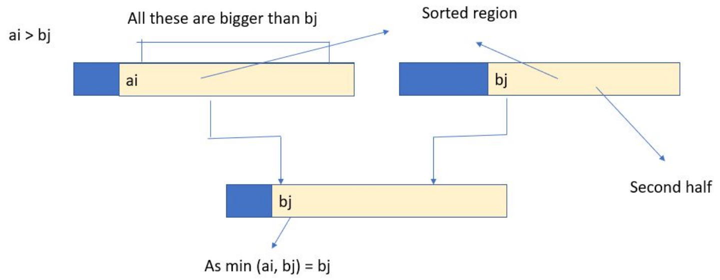
Can we optimize?
We can use merge sort in order to solve this problem more efficiently than the brute force solution.
We can see that if we can devise some recursive procedure for this then maybe the problem can be solved more easily. Let's say we write a function f(arr, start, end) that returns the number of inversions in the array arr[start..end], then we can say that if we divide the array into two parts, arr[start..mid] and arr[mid+1..end] then we have three different areas two get inversion pairs. Some inversion pairs come from only the left half, some come only from the right half and some pairs with one element will be in the first half and the second element in the second half i.e. pairs coming as a part of both. So, we can say that this problem seems now similar to merge sort and other dnc algorithms.

We assume that f(arr, start, mid) and f(arr, mid+1, end) works correctly, and finds the number of inversions in the left half and the number of inversions in the right half. Now we only need to solve the problem for pairs coming to form both the parts.

Inversion Count

Let's take an example
A1 = [2,14,16,19] , A2 = [1,7,10,13]

Here we can see that if we keep two pointers i and j on each corresponding array, then if i = 1 and j = 3 , if we just visualise this pair, then we can say that A2[j] < A1[i] and j > i, so we found one inversion pair, but wait there is more to it, we can see without comparison that A2[j] is lesser than all the elements to the right of i also because as all the elements to the right of i are greater than A1[i] and A1[i] > A2[j] so they are also greater than A2[j], so we got 2 more inversion pairs without even comparing !!! How cool is that.

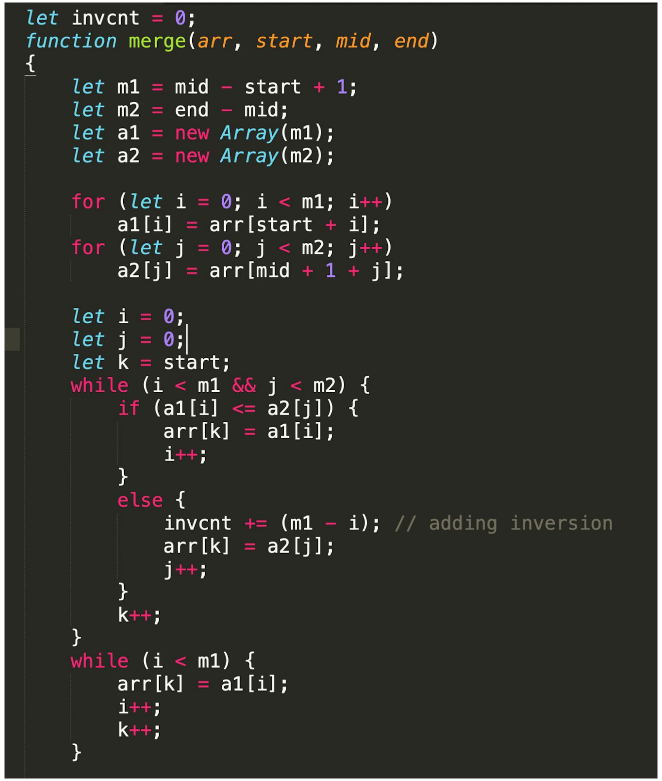


Inversion Count

But this property will be always true only for two sorted arrays. And guess what we know how to maintain two sorted halves from an unsorted array. Yes, mergesort.
So now f(arr, start, end) will not only return the number of inversion in the array arr[start..end] but will also merge sort the array arr[start..end] because we need to maintain the array sorted for above property to hold true. Now we assume f(arr, start, mid) returns the number of inversions in the left half and also sorts the left half correctly and f(arr, mid+1, end) does the same for the right half.
Now our recursive intuition is done, we will start the self-work.
We will start merging the two sorted halves as we need to maintain the array is sorted, and when merging whenever we get an element at index i in the left half greater than the element at index j in the right half we will add (mid-i) to the number of inversions as mid-i elements are greater (because left half was [start..mid]) than the element at index j of right half.
When the whole array is sorted, we have also counted the number of inversions in all possible subranges and hence we will print the answer.

Time Complexity: O(nlogn)
Space Complexity: O(n)

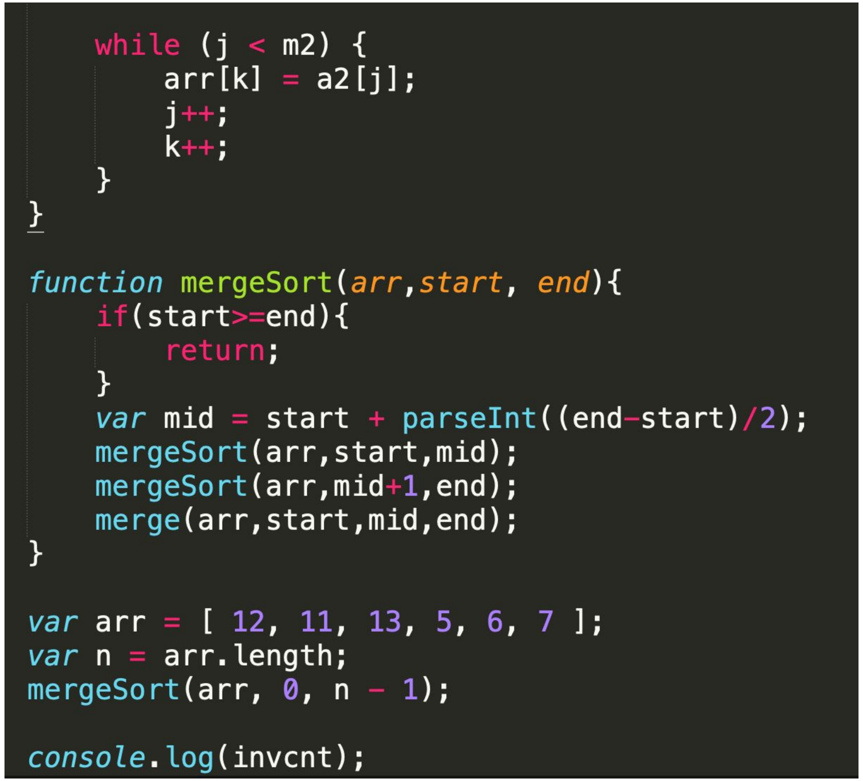
Inversion Count



```
let invcnt = 0;
function merge(arr, start, mid, end)
{
    let m1 = mid - start + 1;
    let m2 = end - mid;
    let a1 = new Array(m1);
    let a2 = new Array(m2);

    for (let i = 0; i < m1; i++)
        a1[i] = arr[start + i];
    for (let j = 0; j < m2; j++)
        a2[j] = arr[mid + 1 + j];

    let i = 0;
    let j = 0;
    let k = start;
    while (i < m1 && j < m2) {
        if (a1[i] <= a2[j]) {
            arr[k] = a1[i];
            i++;
        }
        else {
            invcnt += (m1 - i); // adding inversion
            arr[k] = a2[j];
            j++;
        }
        k++;
    }
    while (i < m1) {
        arr[k] = a1[i];
        i++;
        k++;
    }
```



```
    while (j < m2) {
        arr[k] = a2[j];
        j++;
        k++;
    }
}

function mergeSort(arr,start, end){
    if(start>=end){
        return;
    }
    var mid = start + parseInt((end-start)/2);
    mergeSort(arr,start,mid);
    mergeSort(arr,mid+1,end);
    merge(arr,start,mid,end);
}

var arr = [ 12, 11, 13, 5, 6, 7 ];
var n = arr.length;
mergeSort(arr, 0, n - 1);

console.log(invcnt);
```

Double Inversion Count

This problem changes the definition of the inversion pair from the above problem, Now the inversion pair is a pair such that A[i] > 2*A[j] and i < j, and let's say this is a double inversion pair. Now we need to count the number of double inversion counts.

Solution:

In this problem, we can put exactly the above problem's solution intuition. In the above problem, we say that if A1[i] > A2[j] then everything to the right if i is also greater than A2[j]. Now just think about it if A1[i] > 2*A2[j] then everything to the right of i is also greater than 2*A2[j].

For example:

A1 = [2,4,6,8] A2 = [2,5,7]

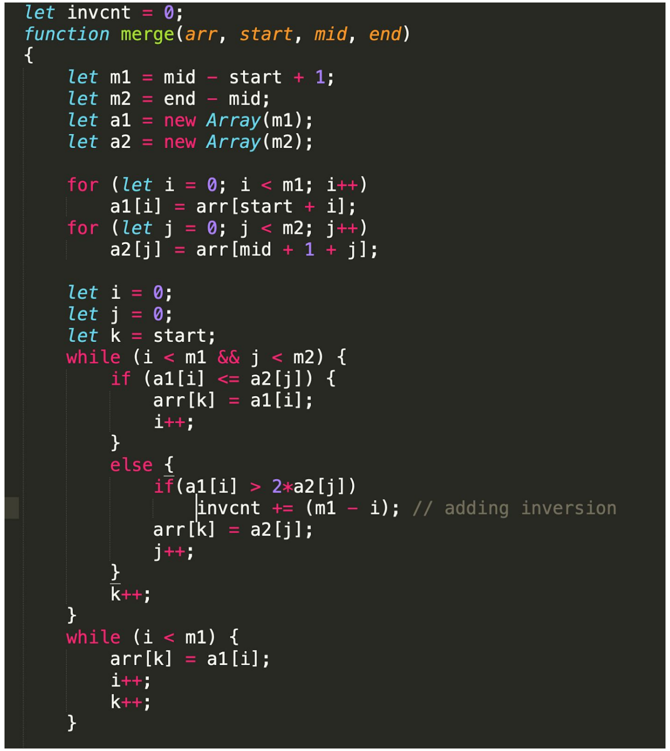
Now let's say i = 2, j = 0, here we can see that A1[i] > 2*A2[j] and we can see everything to the right of A1[i] are also greater than A2[j].

So we can put in the same strategy as the above problem of DNC, We try to get the double inversion pair from the left and right half using recursion, and while merging the two sorted halves, we will find an index i in the left half such that A1[i] > 2*A2[j] instead of A1[i] > A2[j] of the previous problem, and rest everything is the same.

Time Complexity: O(nlogn)

Space Complexity: O(n)

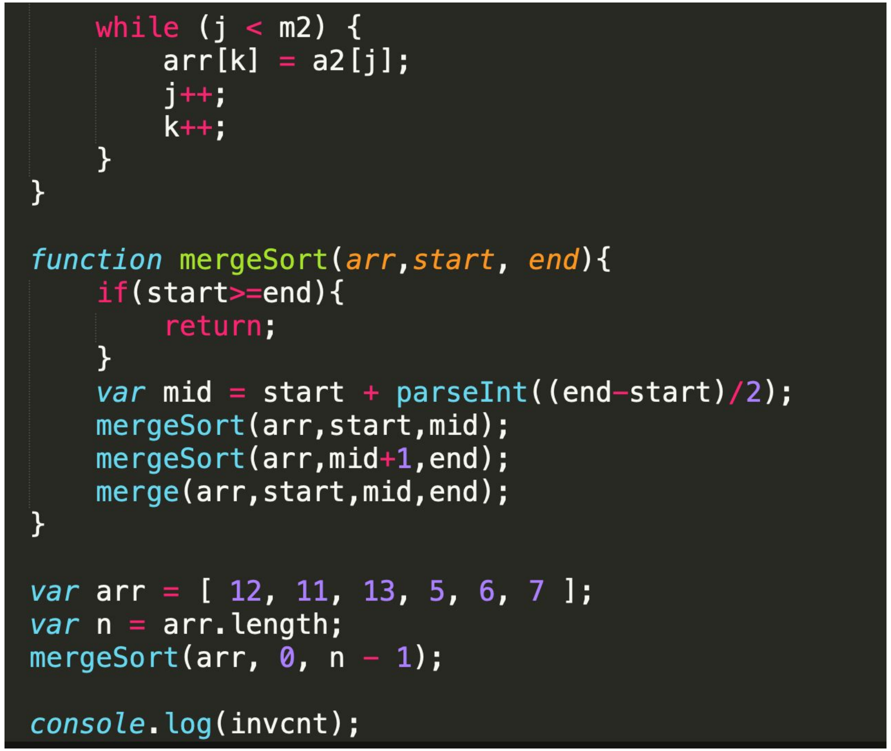
Double Inversion Count



```
let invcnt = 0;
function merge(arr, start, mid, end)
{
    let m1 = mid - start + 1;
    let m2 = end - mid;
    let a1 = new Array(m1);
    let a2 = new Array(m2);

    for (let i = 0; i < m1; i++)
        a1[i] = arr[start + i];
    for (let j = 0; j < m2; j++)
        a2[j] = arr[mid + 1 + j];

    let i = 0;
    let j = 0;
    let k = start;
    while (i < m1 && j < m2) {
        if (a1[i] <= a2[j]) {
            arr[k] = a1[i];
            i++;
        }
        else {
            if(a1[i] > 2*a2[j])
                invcnt += (m1 - i); // adding inversion
            arr[k] = a2[j];
            j++;
        }
        k++;
    }
    while (i < m1) {
        arr[k] = a1[i];
        i++;
        k++;
    }
```



```
    while (j < m2) {
        arr[k] = a2[j];
        j++;
        k++;
    }
}

function mergeSort(arr,start, end){
    if(start>=end){
        return;
    }
    var mid = start + parseInt((end-start)/2);
    mergeSort(arr,start,mid);
    mergeSort(arr,mid+1,end);
    merge(arr,start,mid,end);
}

var arr = [ 12, 11, 13, 5, 6, 7 ];
var n = arr.length;
mergeSort(arr, 0, n - 1);

console.log(invcnt);
```

Using D&C, find the maximum and minimum elements from the given array

Note: This problem can be solved iteratively with fairly simpler code and in optimal run time complexity but to make a be understanding of the topic, we will learn it through Divide and Conquer Approach.

Solution.

1. Function should have base case

   a) if given input array length is 1, then max and min are same i.e. return the same element as max and min.

   b). if the given input array length is 2, then compare both the element's return max and min accordingly.

2. Get Min and Max for left and right halves.
3. Compare left and right halves.
4. Return the results. i.e. Min and Max.

Calculate run time complexity:

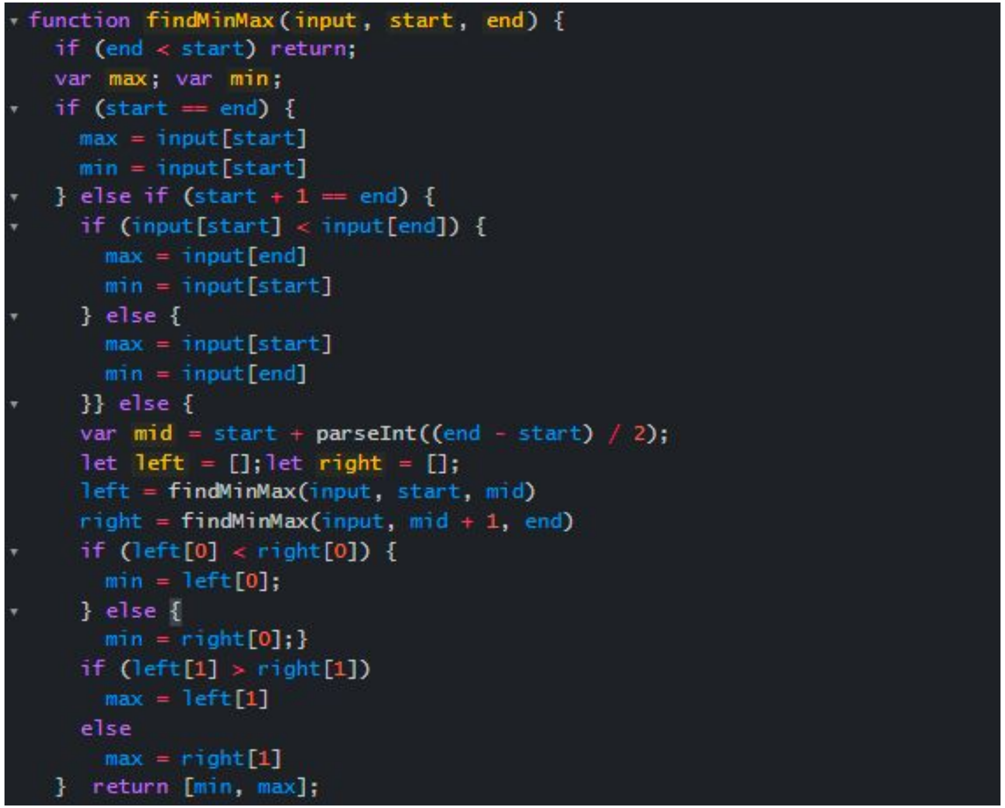
O(n).

Here is the explanation.

T(n) = 2T(n/2) + 2 [constant time for comparsion for an array of length 2]

Using D&C, find the maximum and minimum elements from the given array

Code Link - https://jsfiddle.net/Lzcjrvot/



```
function findMinMax(input, start, end) {
  if (end < start) return;
  var max; var min;
  if (start == end) {
    max = input[start]
    min = input[start]
  } else if (start + 1 == end) {
    if (input[start] < input[end]) {
      max = input[end]
      min = input[start]
    } else {
      max = input[start]
      min = input[end]
    }} else {
    var mid = start + parseInt((end - start) / 2);
    let left = [];let right = [];
    left = findMinMax(input, start, mid)
    right = findMinMax(input, mid + 1, end)
    if (left[0] < right[0]) {
      min = left[0];
    } else {
      min = right[0];}
    if (left[1] > right[1])
      max = left[1]
    else
      max = right[1]
  }  return [min, max];
```

Practice Questions

1) Given two sorted arrays. You need to merge them using merge sort
   Input -> a=[1,3,5,6], b=[2,4,7,8,9]
   Output -> [1,2,3,4,5,6,7,8,9]
1) Given a linkedlist. You need to sort linkedlist using merge sort
   Input -> [115,110,32,10,3,21]
   Output -> [3,10,21,32,110,115]

THANK YOU