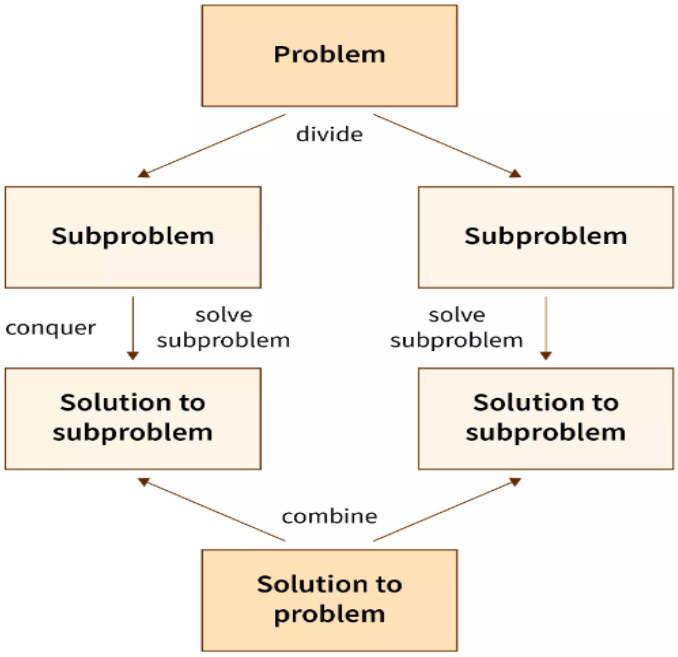
Merge Sort

Introduction:

Merge Sort Merge Sort is a sorting algorithm that follows a divide and conquer technique to achieve its goal.

In merge sort, the given array is divided into roughly two equal sub-arrays. These sub-arrays are divided over and over again into halves until we end up with arrays having only one element each. At last, we merge the sorted pairs of sub-arrays into a final sorted array.

Many algorithms use divide and conquer methods in their own way to sort or search the object or data from any entity. Developers cannot use any algorithm without any consultation with the prediction of the situation that the program can get the benefit from that algorithm. That's why developers cannot use merge sort instead of bubble sort everywhere nor can use bubble sort instead of merge sort or even any other sort.



How does Divide and Conquer work?

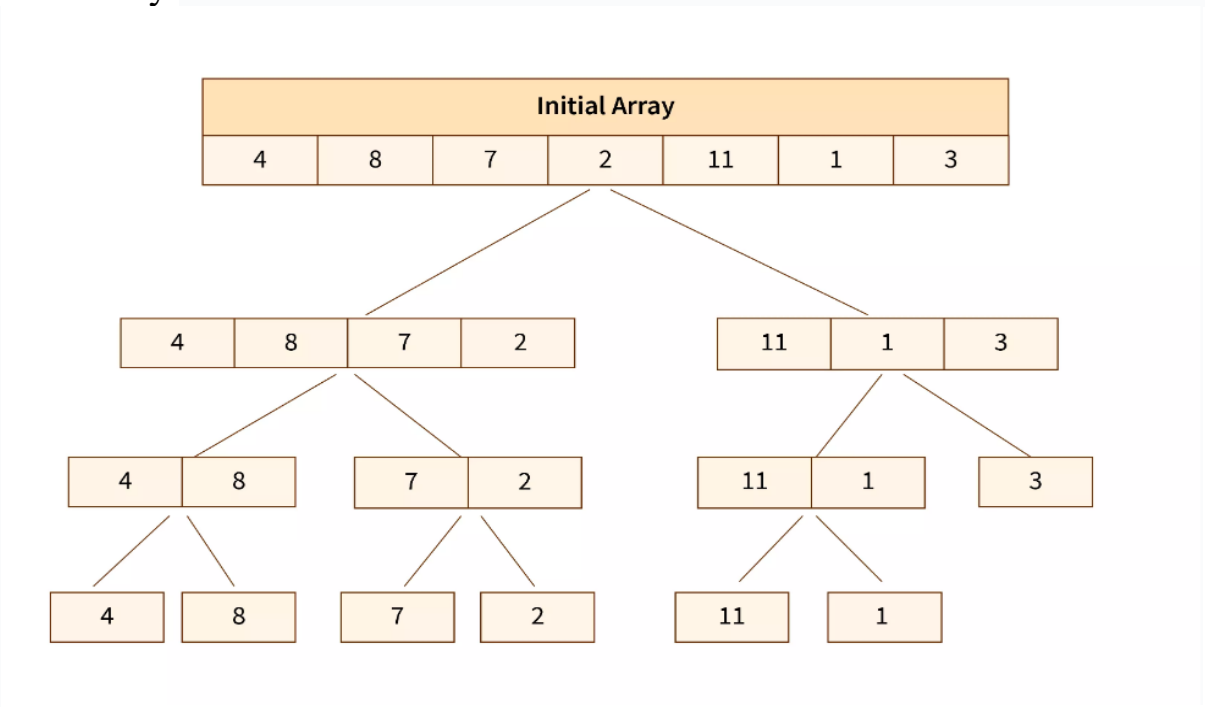
Let us now learn how divide and conquer technique is applied to sort an array of integers. Consider an array, that consists of n number of unsorted integers. Our result should be a sorted array.

Let us consider an array = [4, 8, 7, 2, 11, 1, 3].

1. Divide For dividing the array, we calculate the midpoint of the array by using the formula r = len(array)//2

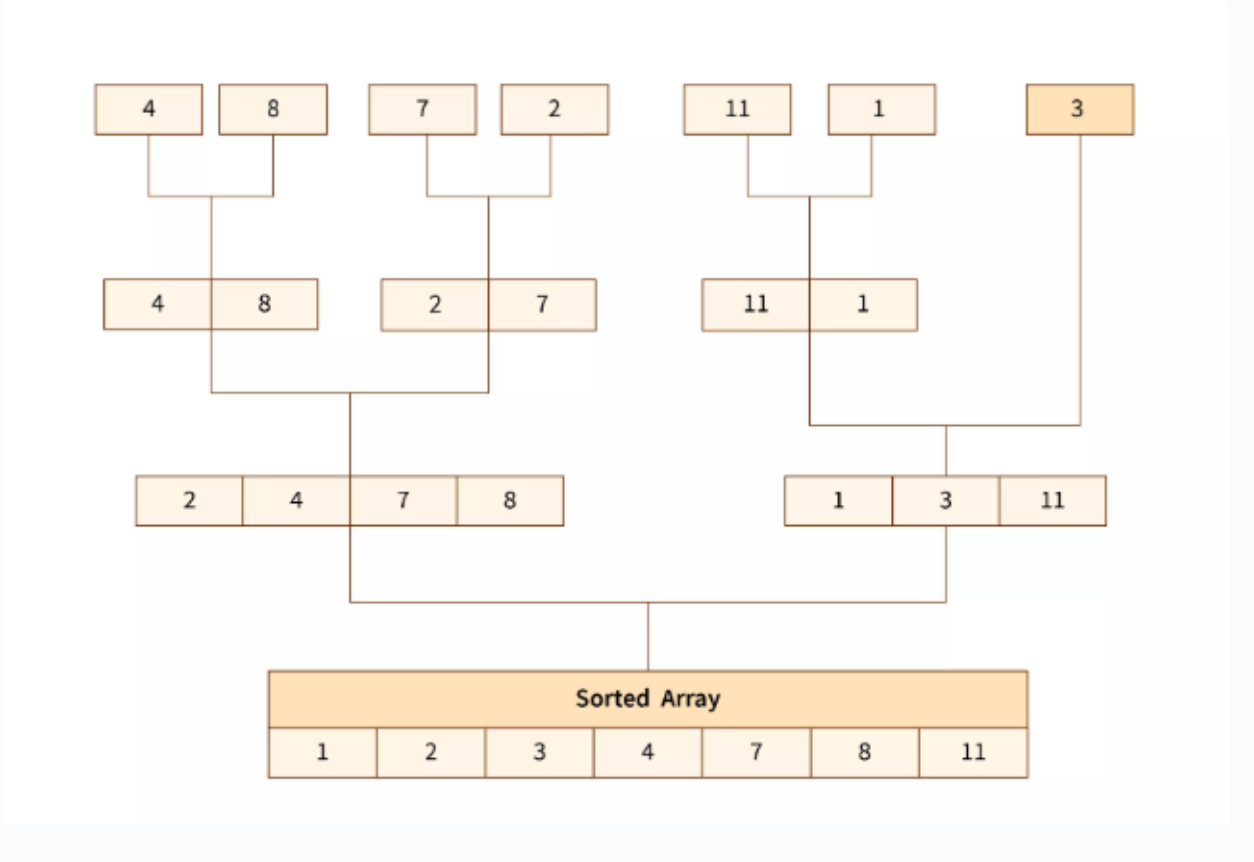
2. Conquer In this step, we divide the array into sub-arrays with the help of the midpoint calculated. This division into sub-arrays and calculation of midpoint is done recursively for the same. As we know, a single element in an array is always sorted. Therefore, we must continuously divide the sub-arrays until all elements in the array are single array elements.

Once this step is completed, the array elements are merged back to form a final sorted array.



Sub-arrays, the last step is to combine them back in sorted order.

3. Combine Since, now we are done with the formation of sub-arrays, the last step is to combine them in sorted order.



Python implementation for Merge Sort

```
def merge(customList, l, m, r):
    # Dividing
    # l = first, m = middle, r = last
    n1 = m - l + 1 # First Subarray
    n2 = r - m # Second Subarray

    L = [0] * (n1) # First Array
    R = [0] * (n2) # Second Array

    # Copy the elements to the first Subarray
    for i in range(0, n1):
        L[i] = customList[l+i]

    # Copy the elements to the Second Subarray
    for j in range(0, n2):
```

```
        R[j] = customList[m+1+j]

    i = 0 # Initial index of First Subarray
    j = 0 # Initial index of Second Subarray
    k = l # Initial index of merge Subarray

    # Merging
    while i < n1 and j < n2:
        if L[i] <= R[j]:
            customList[k] = L[i]
            i += 1
        else:
            customList[k] = R[j]
            j += 1
        k += 1
    while i < n1:
        customList[k] = L[i]
        i += 1
        k += 1

    while j < n2:
        customList[k] = R[j]
        j += 1
        k += 1

def mergeSort(customList, l, r):
    if l < r:
        m = (l+(r-1))//2
        mergeSort(customList, l, m)
        mergeSort(customList, m+1, r)
        merge(customList, l, m, r)
    return customList
```

```
cList = [4, 8, 7, 2, 11, 1, 3]
print(f"Sorted array is : {mergeSort(cList, 0, 6)}")
```

Output:

```
Sorted array is : [1, 2, 3, 4, 7, 8, 11]
```

Merge-Sort and Recurrence Equations

There is another way to justify that the running time of the merge-sort algorithm is O ($n$log n) Namely, we can deal more directly with the recursive nature of the merge-sort algorithm. In this section, we present such an analysis of the running time of merge-sort, and in so doing, introduce the mathematical concept of a ***recurrence equation*** (also known as ***recurrence relation***).

Let the function $t(n)$ denote the worst-case running time of merge-sort on an input sequence of size n. Since merge-sort is recursive, we can characterize function $t(n)$ by means of an equation where the function $t(n)$ is recursively expressed in terms of itself. In order to simplify our characterization of $t(n)$, let us restrict our attention to the case when n is a power of 2. (We leave the problem of showing that our asymptotic characterization still holds in the general case as an exercise.)
In this case, we can specify the definition of $t(n)$ as

$$t(n) = \begin{cases} b & if\ n \leq 1 \\ 2t(n/2) + cn & otherwise, \end{cases}$$

An expression such as the one above is called a recurrence equation, since the function appears on both the left- and right-hand sides of the equal sign. Although such a characterization is correct and accurate, what we really desire is a big-Oh type of characterization of $t(n)$ that does not involve the function $t(n)$ itself. That is, we want a ***closed-form*** characterization of $t(n)$.

We can obtain a closed-form solution by applying the definition of a recurrence equation, assuming n is relatively large. For example, after one more application of the equation above, we can write a new recurrence for $t(n)$ as

$$\begin{aligned} t(n) &= 2\left(2t\left(\frac{n}{2^2}\right) + (\frac{cn}{2})\right) \\ &= 2^2 t\left(\frac{n}{2^2}\right) + 2\left(\frac{cn}{2}\right) + cn \\ &= 2^2 t\left(\frac{n}{2^2}\right) + 2cn \end{aligned}$$

If we apply the equation again, we get $t(n) = 2^3 t\left(\frac{n}{2^3}\right) + 3cn$. At this point, we should see a pattern emerging, so that after applying this equation *i* times, we get

$$t(n) = 2^i t\left(\frac{n}{2^i}\right) + icn$$

The issue that remains, then, is to determine when to stop this process. To see when to stop, recall that we switch to the closed form $t(n) = b$ when $n \leq 1$, which will occur when $2i = n$. In other words, this will occur when $i = \log n$. Making this substitution, then, yields

$$\begin{aligned} t(n) &= 2^{logn} t\left(\frac{n}{2^{logn}}\right) + (logn)cn \\ &= nt(1) + cnlogn \\ &= nb + cnlogn \end{aligned}$$

That is, we get an alternative justification of the fact that $t(n)$ is O ($n$log n).

THE END