Divide & Conquer.

Merge sort is based on an algorithmic design pattern called divide & conquer. The divide & conquer paradigm can be described in general terms as consisting of the following steps:

1. Divide: If the input size is smaller than a certain threshold, (say 1 or 2 elents), then solve the problem directly using a straight forward method & return the solution obtained. Otherwise divide the data into two or more distinct subsets.

2. Recur: Recursively solve the subproblem associated with the subset

3. Conquer: Take the solutions to the ~~subproblems~~ subproblems & merge them into a solution to the original problem.

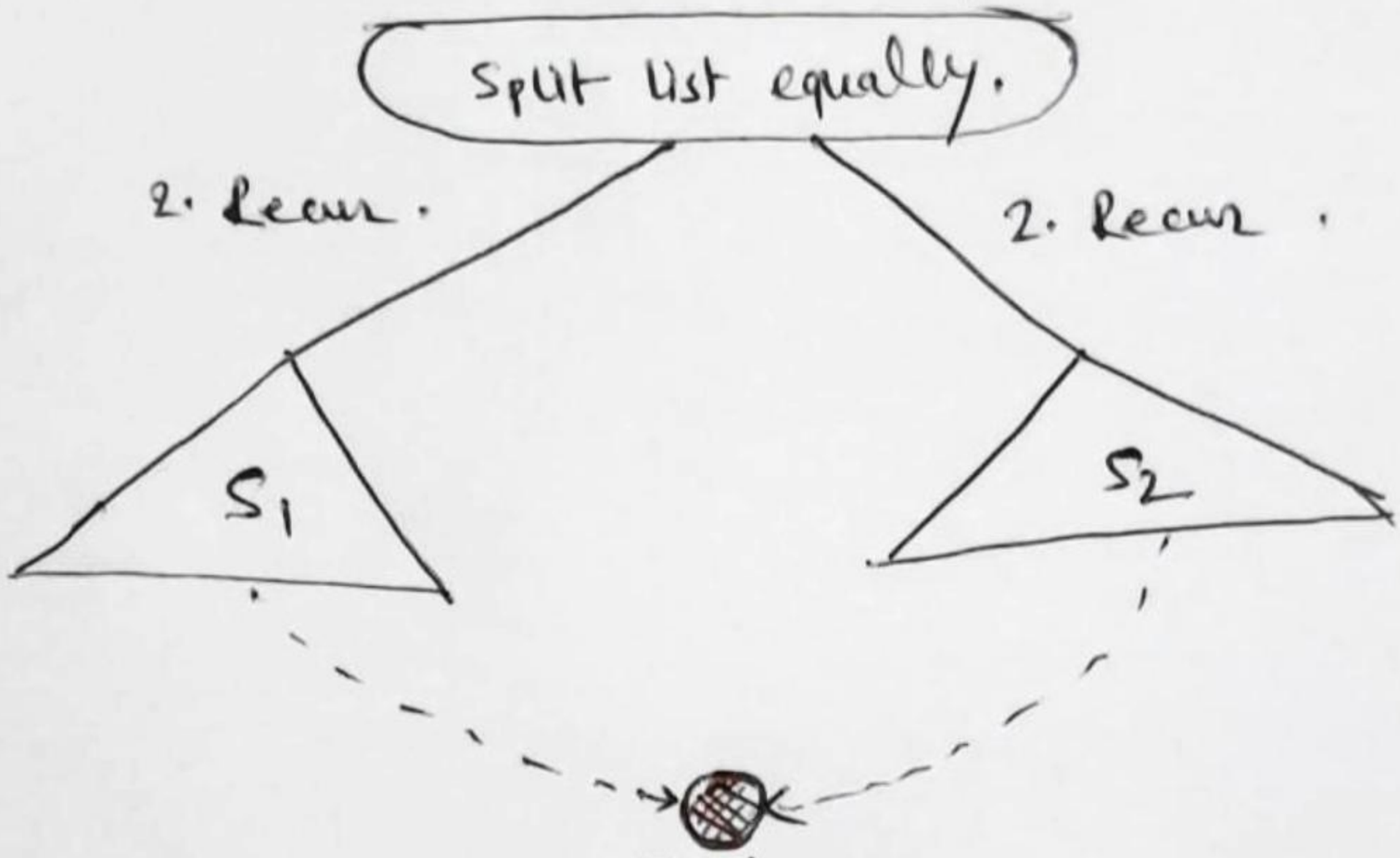
Using Divide & conquer for sorting:

Say we have to sort a collection(S) of n objects.

1. Divide :- → if S has '0' or '1' element then return S immediately it is already sorted.
→ else if S have at least '2' elements then remove all the elements from S & put them into two sequences S_1 & S_2 each containing about half of the elements of S, ie S_1 contains the first $n/2$ elements of S & S_2 contains the remaining $n/2$ elements.

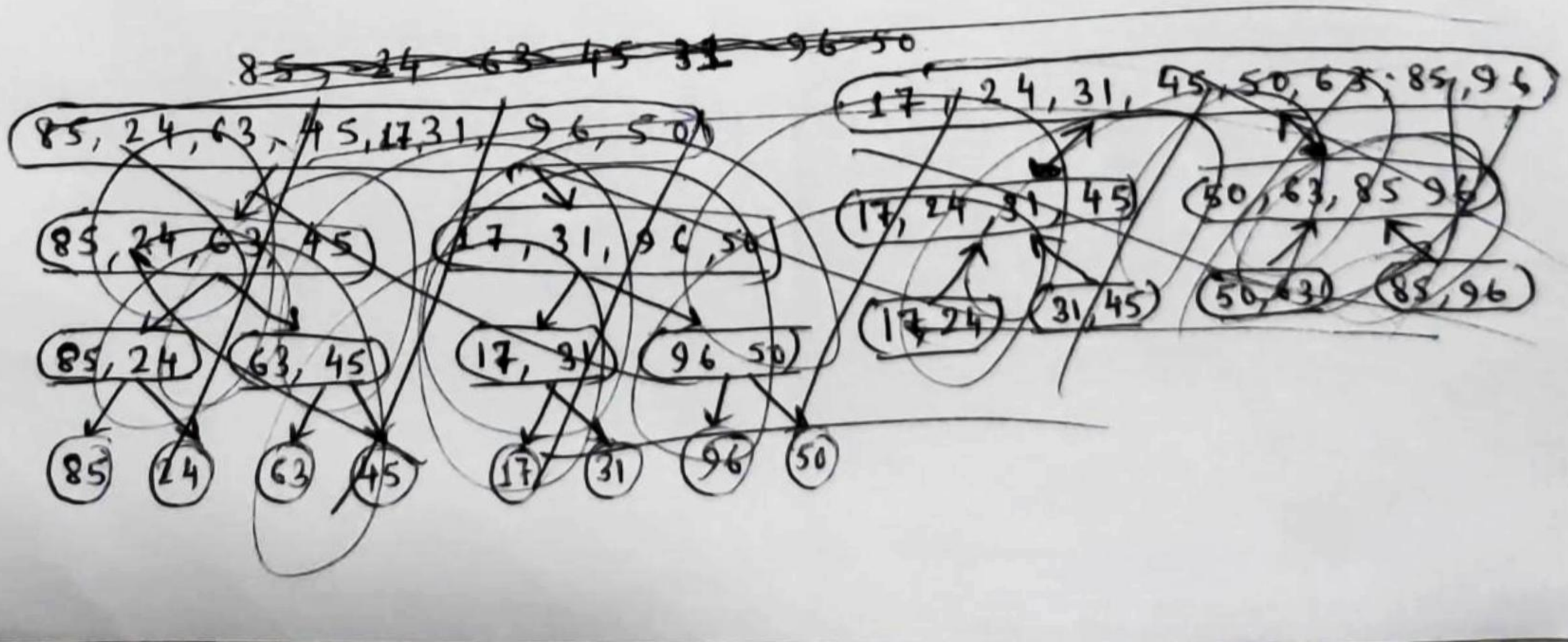
2. Recur: Recursively sort sequences S_1 & S_2.

3. Conquer: Put elements back into S by merging the sorted sequences S_1 & S_2 into a sorted sequence.

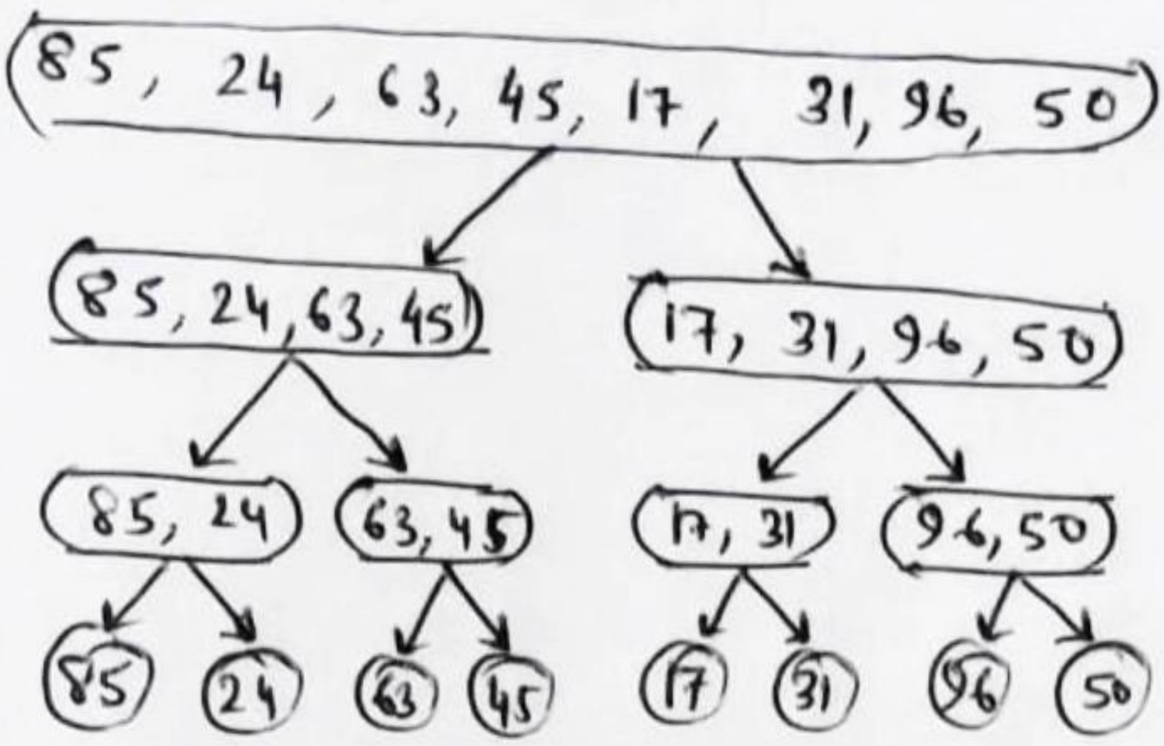


A visual scheme for merge sort Algorithm.

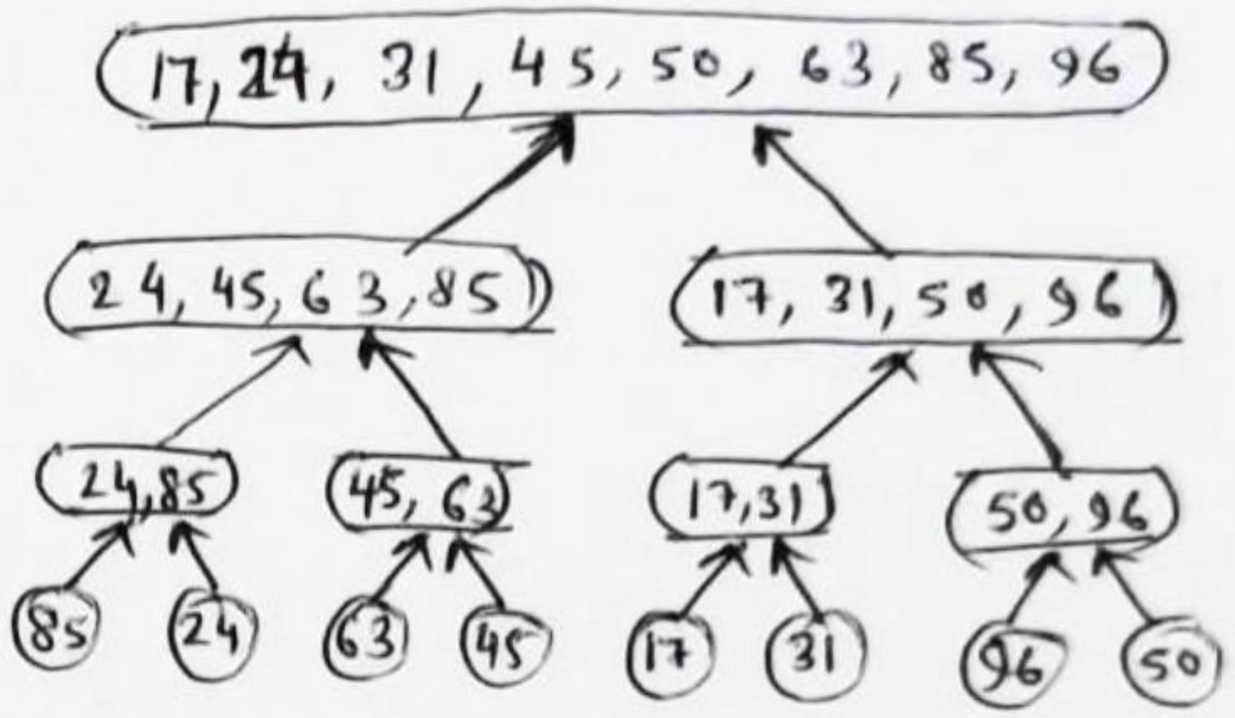
We can visualize an execution of the merge sort algorithm by means of a binary tree T. called merge-sort-tree. Each node of T represents a recursive invocation to the merge sort algorithm.



A



B



Merge-Sort tree T for an execution of the merge-sort algorithm on a sequence of eight elements:

A: input sequences processed at each node of T.

B. output sequences generated at each node of T.

Algorithm : merge (S_1, S_2, S) .

Input: sequences S_1 & S_2 sorted in non decreasing order & an empty sequence S.

output: Sequence S containing the elements from S_1 & S_2 sorted in non decreasing order with S_1 & S_2 becoming empty.

```
While ( not (S1. isEmpty () or S2. isEmpty ()) do
    if S1. first(). element() ≤ S2. first(). element() then.
        { move the first element of S1 at the end of S}.
        S. insertLast ( S1. remove ( S1. first ()))
    else
        { move the first elemt of S2 at the end of S}.
        S. insertLast ( S2. remove ( S2. first ()))
 { move the remaing elemts of S1 to S}.
while (not S1. isEmpty ()) do
    S. insertLast ( S1. remove ( S1. first ()))
 { move the remaining elements of S2 to S}.
 while (not S2. isEmpty ()) do
    S. insertLast ( S2. remove ( S2. first ())) .
```

Analysis of merging operation

Let n_1 & n_2 be the number of elements of S_1 & S_2 respectively. Also let us assume that the sequences S_1, S_2 & S are implemented in such way, that the insertion & deletion from their first & last position will take $O(1)$ time.

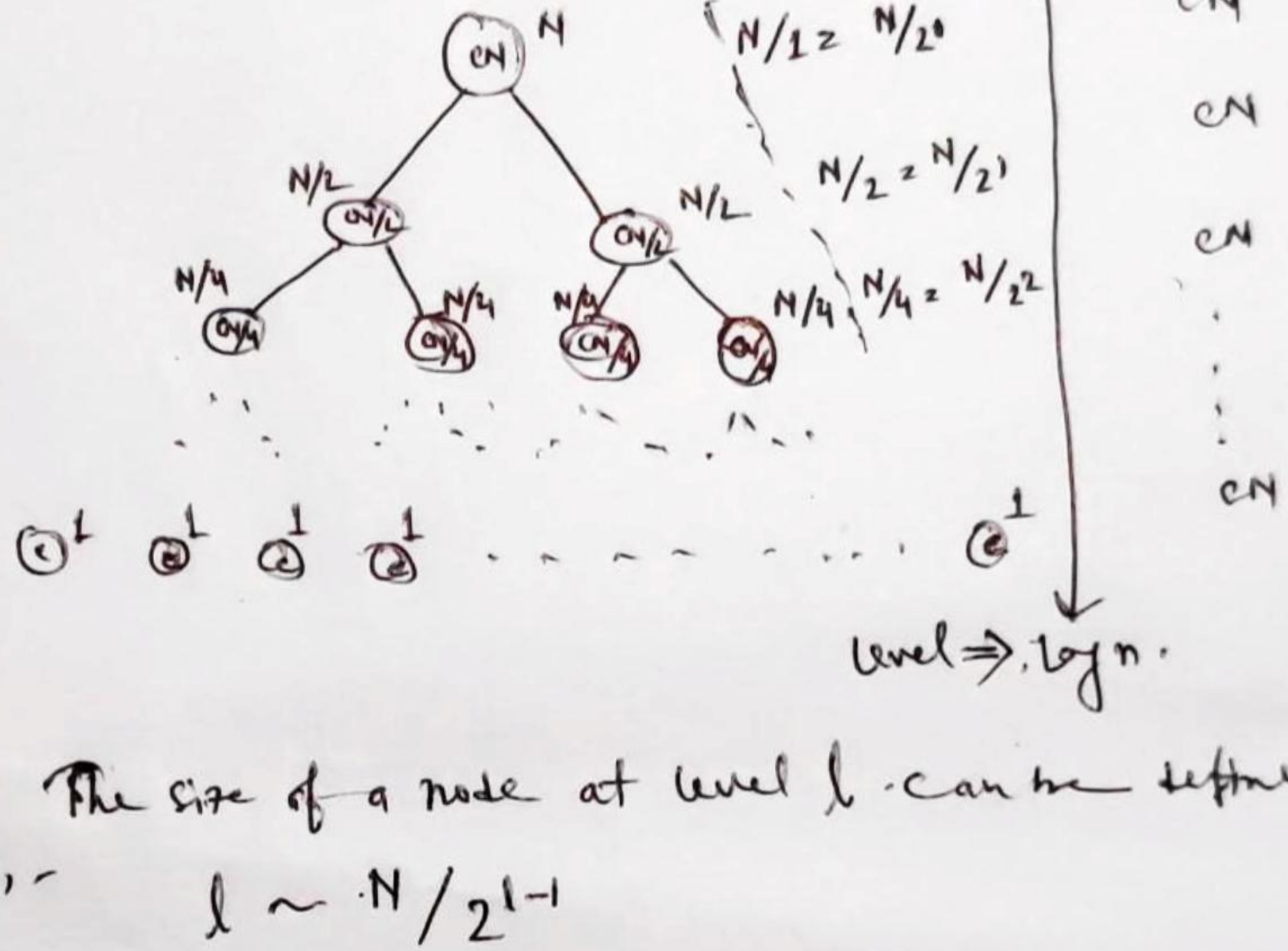
Algorithm merge has three while loops. & according to our assumption the operations inside the loop takes $O(1)$ time each.

From the observation we can see that in each iteration one element is removed from S_1 or S_2. Since no insertion is performed into S_1 or S_2. Thus the overall iteration of three loops in (n_1+n_2). Therefore overall time of the merge Algorithm is $O(n_1+n_2)$.

The Running time of Merge Sort.

Recursion tree Method:

$T(n) = 2T(n/2) + cn$ / $T(1) = O(1)$.



The size of a node at level l can be defined

@, - $l \sim N/2^{l-1}$

at the bottom the size will be 1.

thus, - $N/2^{l-1} = 1$

$N = 2^{l-1}$

$l - 1 = \lg N$.

$l \approx \lg N$.

Thus $T(n) = cN + cN + \ldots$ upto l terms.

$= cNl = cN \lg N$.

$T(n) = O(N \lg N)$.

Merge Sort & Recurrence Equations:

Since merge sort is recursive, we can characterize function $t(n)$ by means of following equalities, where function $t(n)$ is recursively expressed in terms of itself.

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

where $b > 0$ & $c > 0$ are constants.

In order to provide a closed form characterization of $t(n)$ let us restrict our attention to the case where n is power of 2. we can simply define $t(n)$ as:

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

But even so, we must try to characterize this recurrance equation in a closed form way. One way to do this is to iteratively apply this equation assuming n is relatively large.

for example after one more application of this equation we can write a new recurrance for $t(n)$ as follows:

$$t(n) = 2(2t(n/2^2) + c(n/2)) + cn.$$
$$= 2^2 t(n/2^2) + 2cn.$$

if we apply it again

$$t(n) = 2^2(2t(n/2^3) + c(n/2^2)) + 2cn$$
$$= 2^3 t(n/2^3) + 3cn.$$

after ith iteration

$$t(n) = 2^i t(n/2i) + icn.$$

The issue that remains, then, is to determine when to stop this iterative process.

The answer is when $t(n) \geq b \cdot n$, $n \geq 1$.

This occurs when $2^i \geq n$.

or $i \geq \lg n$.

if we substitute it in the last equation then

$$t(n) = 2^{\lg n} t\left(n/2^{\lg n}\right) + (\lg n)\, cn.$$

$$= n t(1) + cn \lg n.$$

$$= nb + cn \lg n.$$

$$= bn + cn \lg n.$$

i.e. The complexity is $\underline{O(n \lg n)}$.