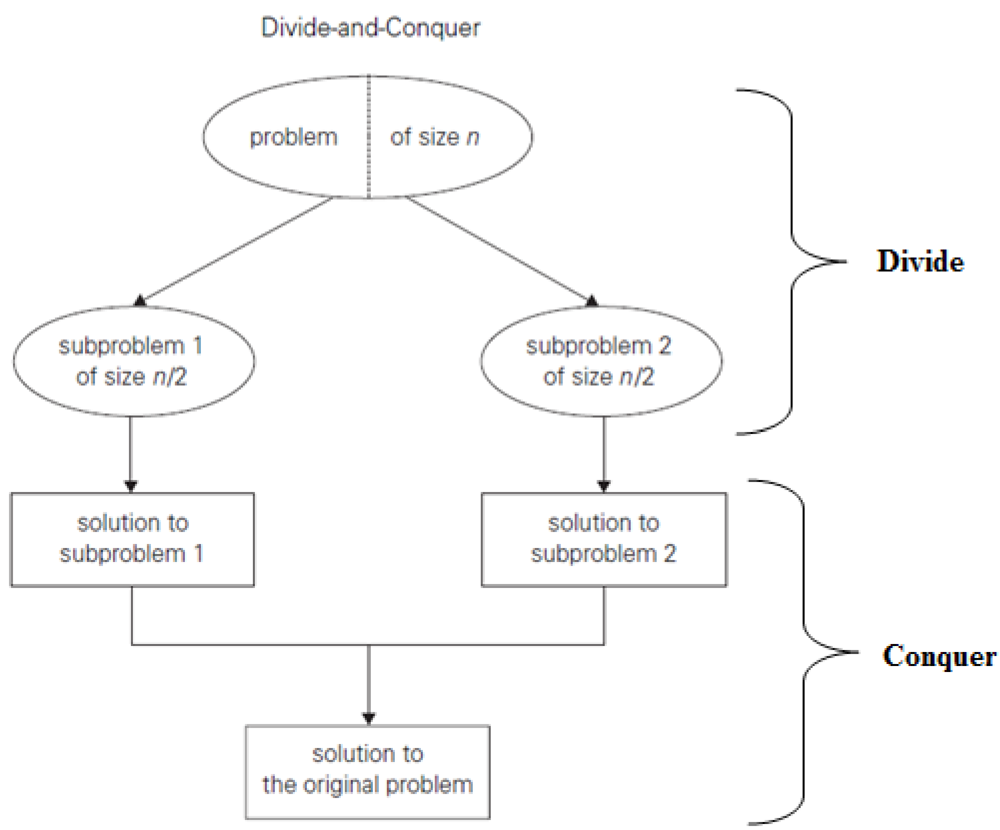
Unit-2

Divide-and-Conquer

- Divide-and-conquer is probably the best-known general algorithm design technique.
- Though its fame may have something to do with its catchy name, it is well deserved: quite a few very efficient algorithms are specific implementations of this general strategy.
- imp Divide-and-conquer algorithms work according to the following **general plan**:
  1. **A problem is divided into several subproblems of the same type, ideally of about equal size.**
  2. **The subproblems are solved (typically recursively, though sometimes a different algorithm is employed, especially when subproblems become small enough).**
  3. **If necessary, the solutions to the subproblems are combined to get a solution to the original problem.**
- The divide-and-conquer technique is diagrammed in Figure below, which depicts the case of dividing a problem into two smaller subproblems, by far the most widely occurring case



****

imp Master theorem

- Divide-and-conquer a problem's instance of size *n* can be divided into *b* instances of size *n/b*, with *a* of them needing to be solved
- Assuming that size *n* is a power of *b* to simplify our analysis, we get the following recurrence for the running time *T (n)*:

$$\boxed{T(n) = aT(n/b) + f(n)} \quad \ldots\ldots\ldots(1)$$

- where
  - *a* and *b* are constants; $a \geq 1$ and $b > 1$
  - *f (n)* is a function that accounts for the time spent on dividing an instance of size *n* into instances of size *n/b* and combining their solutions.
- The above recurrence is called the ***general divide-and-conquer recurrence***.
- Obviously, the order of growth of its solution *T (n)* depends on the values of the constants *a* and *b* and the order of growth of the function *f (n)*.
- The efficiency analysis of many divide-and-conquer algorithms is greatly simplified by the following theorem
- If $f(n) \in \Theta(n^d)$ where $d \geq 0$ in recurrence (1), then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_b n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

- Here d is the power of n in f(n)
- Analogous results hold for the *O* and Ω notations, too

imp Sum of n numbers

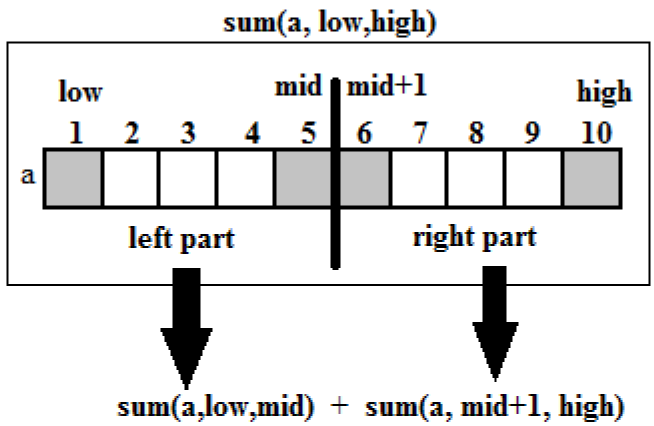
- Let us consider the problem of computing the sum of *n* numbers *a*0, *a1*,. . . , *an*−1.
- If $n > 1$, we can divide the problem into two instances of the same problem:
  - to compute the sum of the first [*n*/2] numbers
  - to compute the sum of the remaining [*n*/2] numbers.
- Of course, if $n = 1$, we simply return *a*0 as the answer.
- Once each of these two sums is computed by applying the same method recursively, we can add their values to get the sum in question:

$$a_0 + \cdots + a_{n-1} = (a_0 + \cdots + a_{\lfloor n/2 \rfloor - 1}) + (a_{\lfloor n/2 \rfloor} + \cdots + a_{n-1})$$

- So sum of all elements stored in the array a between 0 and n-1 (low and high respectively) can be calculated using following steps
  1. If low=high, it indicates that there is only one element in the array and return a[low] as the solution
  2. If low<high, we can divide the problem into 2 parts using the relation:

     mid ← (low+high)/2

  3. Recursively obtain the sum of left part ranging from low to mid
  4. Recursively obtain the sum of right part ranging from mid+1 to high
  5. Find the sum of left part and right part
- This can be represented as shown below:



- The recursive relation to find sum of n numbers can be written as

$$f(a, low, high) = \begin{cases} a[low] & \text{if } low=high \\ \boxed{\begin{array}{l} mid \leftarrow (low + high)/2 \\ f(a, low, mid) + f(a. mid+1, high) \end{array}} & \text{otherwise} \end{cases}$$

Algorithm:

```
/*Input : array a where n elements can be stored
Low and high represents the low index and high index*/
//Output: returns sum of all elements
If (low=high)
    Return a[low]
Else
{
Mid←(low+high)/2
Return sum(a, low, mid)+sum(a, mid+1, high) }
```

Analysis using substitution method:

The time efficiency to find sum of n elements is calculated as below:

$$T(n) = \begin{cases} 0 & \text{if } n=1 \\ T(n/2) + T(n/2) + 1 & \text{otherwise} \end{cases}$$

Time required to add elements in the left part of array

Time required to add elements in the right part of

Time required to add left part and right part

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + 1$$

$$= 2T\left(\frac{n}{2}\right) + 1$$

$$= 2\left[2T\left(\frac{n}{4}\right) + 1\right] + 1$$

$$= 2^2T\left(\frac{n}{2^2}\right) + 2 + 1$$

$$= 2^2\left[2T\left(\frac{n}{2^3}\right) + 1\right] + 2 + 1$$

$$= 2^3T\left(\frac{n}{2^3}\right) + 2^2 + 2 + 1$$

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n/2}{2}\right) + 1$$

$$= 2T\left(\frac{n}{2^2}\right) + 1$$

$$T\left(\frac{n}{2^2}\right) = 2T\left(\frac{n/2^2}{2}\right) + 1$$

$$= 2T\left(\frac{n}{2^3}\right) + 1$$

…………………

…………………

$$= 2^iT\left(\frac{n}{2^i}\right) + \left[2^{i-1} + 2^{i-2} + 2^{i-3} + \cdots + 2^3 + 2^2 + 2 + 1\right] \ldots\ldots \text{❶}$$

We have $a^i + a^{i-1} + a^{i-2} + \cdots + a^3 + a^2 + a + 1 = a^{n+1} - 1$

Substitute that values in equation (1), we get

$$\therefore T(n) = 2^iT\left(\frac{n}{2^i}\right) + \left[2^{i-1+1} - 1\right]$$

$$= 2^iT\left(\frac{n}{2^i}\right) + \left[2^i - 1\right]$$

Inorder to get initial condition make $2^i = n$

$$\therefore T(n) = nT\left(\frac{n}{n}\right) + [n-1]$$

$$T(n) = nT(1) + [n-1]$$

$$\therefore T(n) = n * 0 + [n-1]$$

$$\therefore T(n) = n - 1$$

Represent it using Θ notation

We have the general relation as

$C1.g(n) \leq f(n) \leq C2.g(n)$ where $n \geq n_0$

$\frac{n}{4} \leq n - 1 \leq 4n \; where \; n \geq 2$

$C1 = \frac{n}{4}, C2 = 4, n_0 = 2, g(n) = n$

$\therefore T(n) \epsilon \; \theta(g(n))$

$\underline{\therefore \boldsymbol{T(n)\epsilon \; \theta(n)}}$

Analysis using Master theorem:

We have the master theorem as below:

T (n) = aT (n/b) + f (n) ……….(1)

The time complexity can be calculated using following relation:

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_b n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases} \quad \ldots\ldots\ldots\ldots(2)$$

For our problem we got the recurrence relation as:

$T(n) = 2T\left(\frac{n}{2}\right) + 1$……. (3)

Comparing eqn (1) and eqn (3), we can write, a=2, b=2, $f(n)=1=n^0$, d=0 [power of n in f(n)]

a **b^d**

2 2^0

2 > 1

So $a>b^d$

So from eqn (2) we get the relation as:

$$T(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n).$$

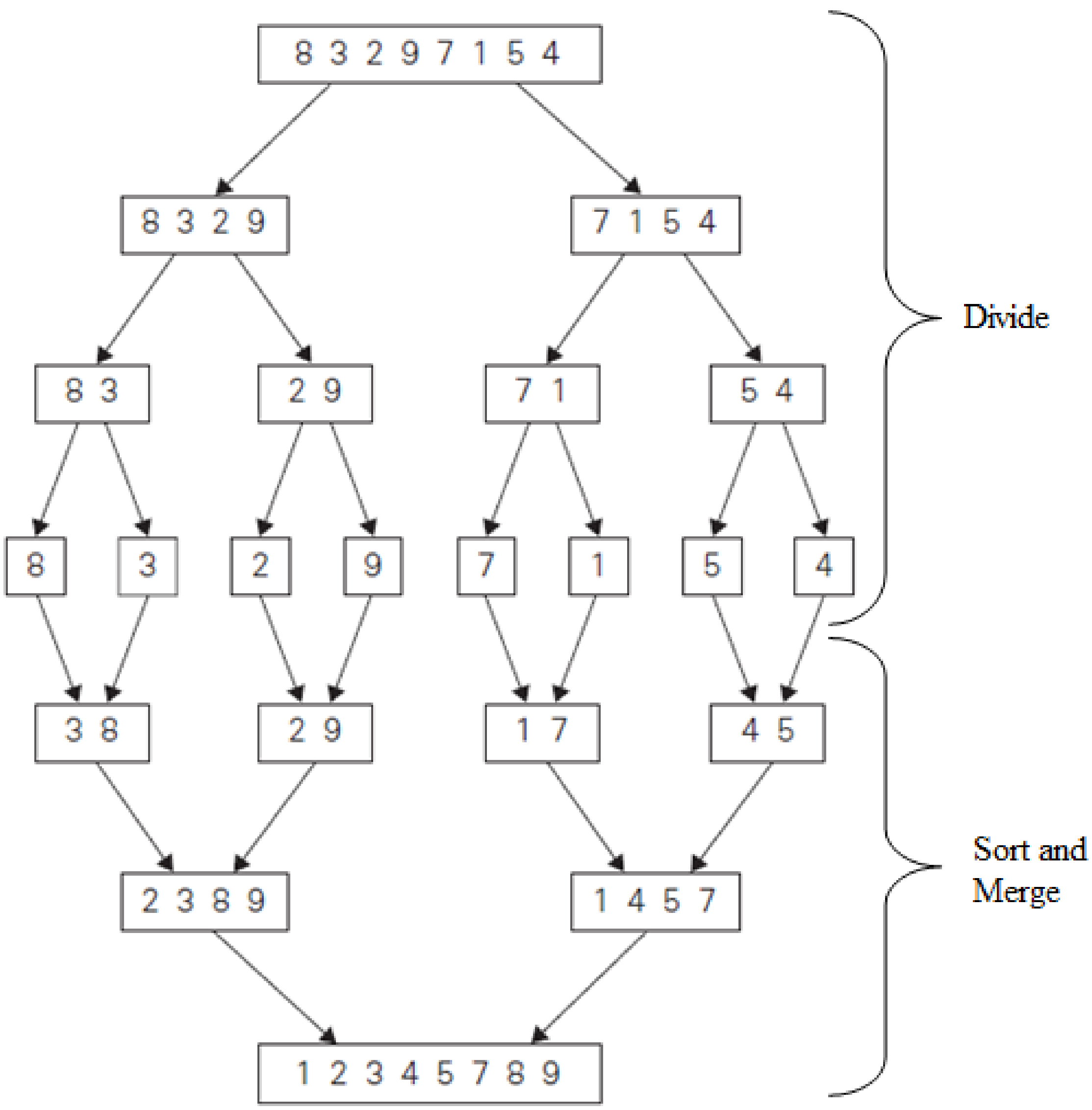
$\underline{\therefore \boldsymbol{T(n)\epsilon \; \theta(n)}}$

****

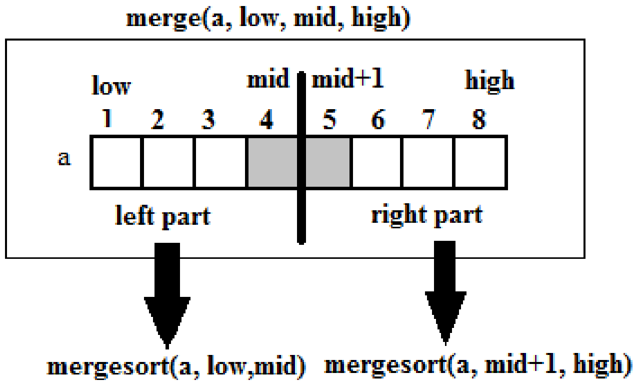
imp **Merge Sort**

- It uses divide and conquer method
- Mergesort is a perfect example of a successful application of the divide-and conquer technique.
- It sorts a given array A[0…n − 1] by dividing it into two halves A[0….(n/2)−1] and A[(n/2)…..n−1], sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.
- The various steps are:
    1. **Divide** the given array into 2 parts with n/2 elements each
    2. **Sort** left part and right part recursively
    3. **Merge** the sorted left part and sorted right part into single resultant array

Example:



Design:



So from the above figure, we can write the function for merge sort. The steps include:

- If (low<high)
  - Then divide the array into equal parts and find mid position
  - Sort the left part of the array recursively [mergesort(a, low,mid)]
  - Sort the left part of the array recursively [mergesort(a, mid+1, high)]
  - Merge the left part and right part [merge(a, low, mid, high)]

Algorithm for mergesort function is:

//Sorts array a[0..n − 1] by recursive mergesort

//Input: An array a[0..n − 1] of orderable elements

//Output: Array a[0..n − 1] sorted in ascending order

if low<high

mid←(low+high)/2

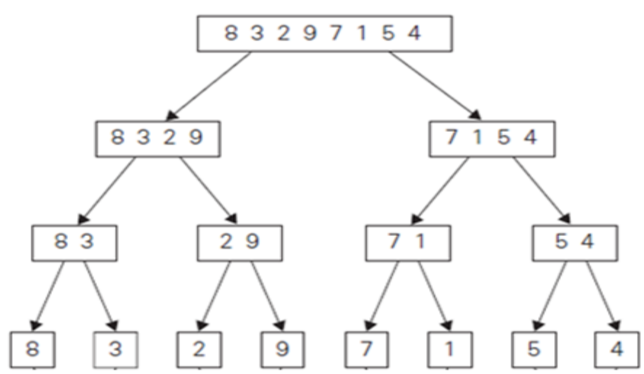
mergesort(a, low,mid)

mergesort(a, mid+1, high)

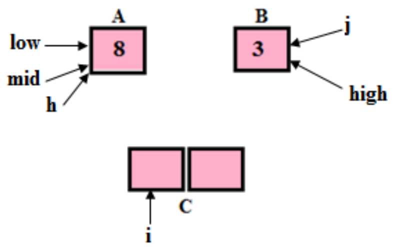
merge(a, low, mid, high)

end if

After the above algorithm the elements will be divided as shown in the figure →



- Next we need to sort it and merge it.
- Consider first 2 elements 8 and 3.
- Initially it will be as shown in the figure



Here we are initializing as below

h←low

i←low

j←mid+1

- Now compare hth and jth elements in the figure (8 and 3 respectively) and copy the lesser element into the ith position of array C and increment the value of i and h by 1so that they can point to the next position (C is having the capacity of size of A+ size of B arrays. i.e, if A is of size 2 and B is of size 2, then C is of size 4)
- We can write the code for the above comparison as shown below

  If A[h]<=B[j] then,

      C[i]←A[h]

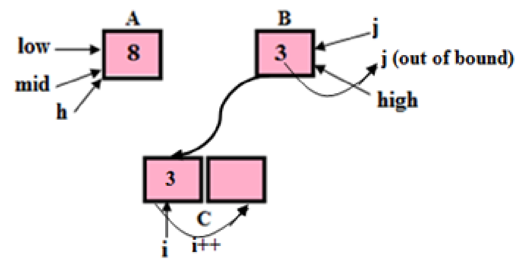
      h←h+1, i←i+1

  Else

      C[i]←B[j]

      j←j+1, i←i+1

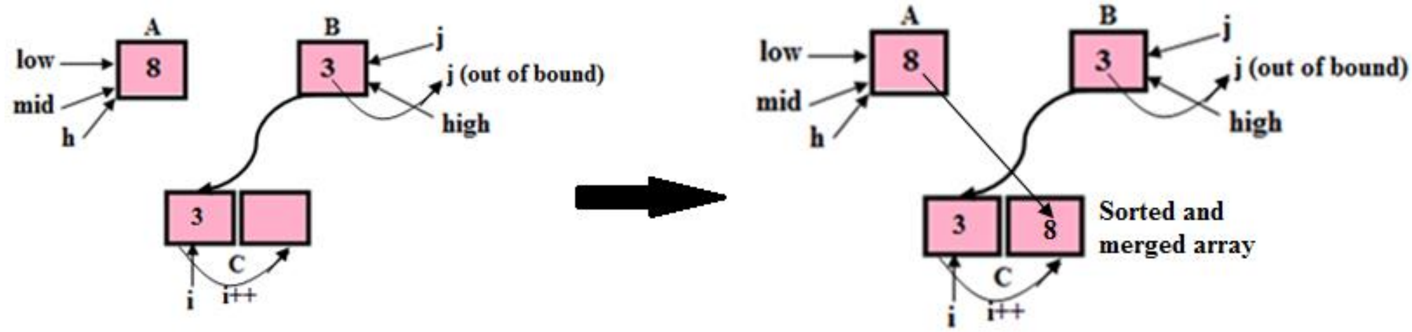
  here it will execute else part because A[h]>B[j] (8>3). So it will put B[j], i.e 3 to C[i]



- The above step should be repeated untill h is less than mid and j is less than high (i.e, h should not exceed the value of mid and j should not exceed the value of high. When it exceeds, the loop should stop)
- So we can write the code as:

```
While(h<=mid && j<=high)
        If A[h]<=B[j] then,
                C[i]←A[h]
                h←h+1, i←i+1
        Else
                C[i]←B[j]
                j←j+1, i←i+1
End while
```

- After that step, the remaining element in the left out array should be placed into the resultant array as it is as shown in the figure below.



- This can be done by copying A[h] on to C[i] until h goes out of bound, or the other case is copying B[j] to C[i] until j goes out of bound. This can be coded as below:

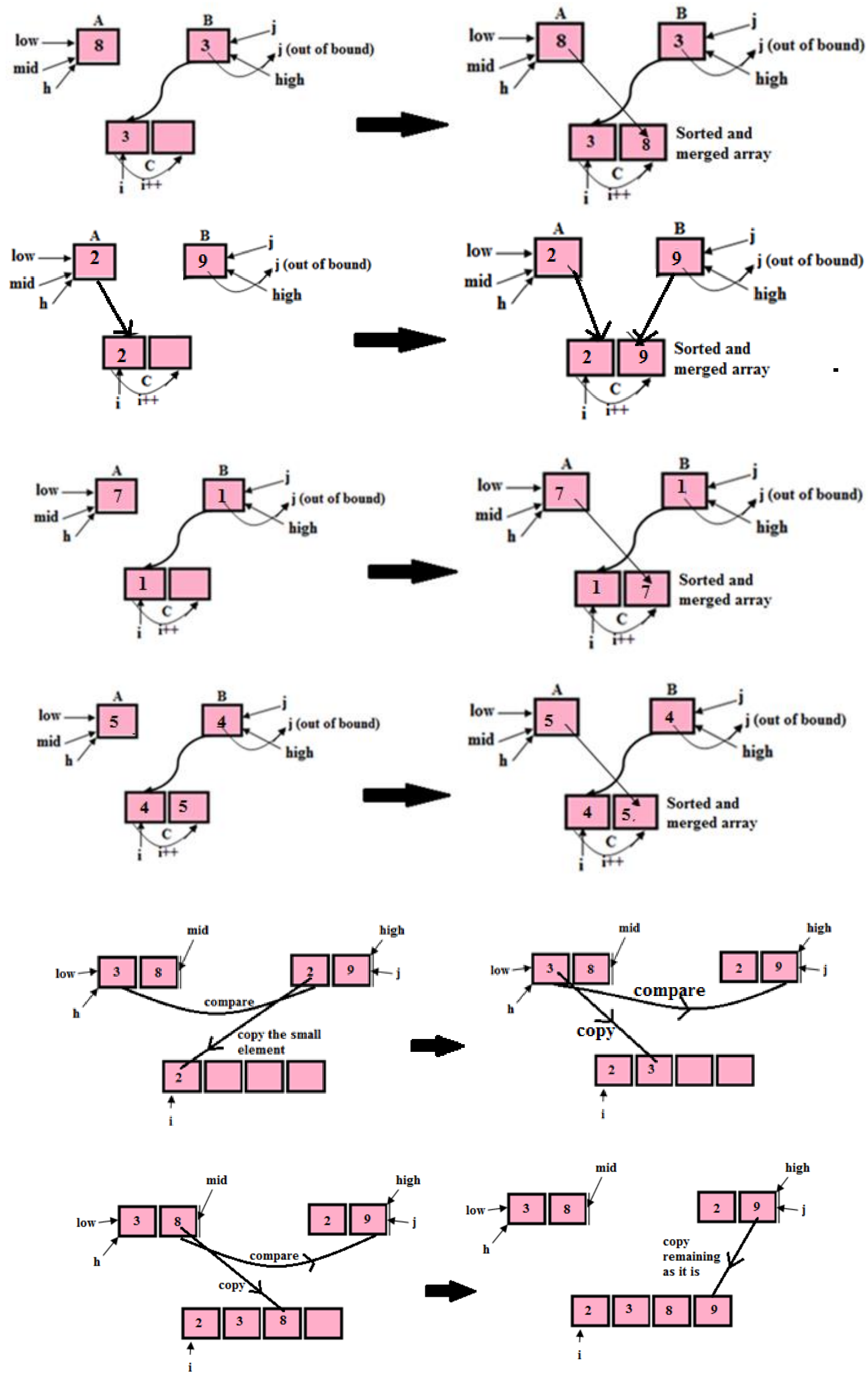
```
If h>mid then
     while j<=high then,
               C[i]←B[j]
               i←i+1
     end while
else
     while h<=mid
               C[i]=A[h]
               i←i+1
     end while
```

So we can write the algorithm for the merge function as below:

```
h←low, i←low, j←mid+1
While(h<=mid && j<=high)
          If A[h]<=B[j] then,
                    C[i]←A[h]
                    h←h+1, i←i+1
          Else
                    C[i]←B[j]
                    j←j+1, i←i+1
End while
```

```
If h>mid then
     while j<=high then,
               C[i]←B[j]
               i←i+1
     end while
else
     while h<=mid
               C[i]=A[h]
               i←i+1
     end while
```

Tracing using the algorithm:



Continue the same process…..

Analysis:

- It is clear from the algorithm that the problem is divided into 2 equal parts
- So we can write the recurrence relation for the algorithm as below:

$$T(n) = \begin{cases} 0 & \text{if } n=1 \\ T(n/2) + T(n/2) + n & \text{otherwise} \end{cases}$$

Time required to Sort elements in the left part of array

Time required to Sort elements in the right part of

Time required to merge the left part and right part

- If n=1, then only one element will be there in the list, and no need of sorting it. So time required is 0
- The above relation can also be written as below:

$$T(n) = \begin{cases} 0 & \text{if } n=1 \\ 2T(n/2) + n & \text{otherwise} \end{cases}$$

Time complexity using master theorem:

We have the master theorem as below:

T (n) = aT (n/b) + f (n) ……….(1)

The time complexity can be calculated using following relation:

$$\boxed{T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_b n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}}$$ …………(2)

For our problem we got the recurrence relation as:

$T(n) = 2T\left(\frac{n}{2}\right) + n$……. (3)

Comparing eqn (1) and eqn (3), we can write, a=2, b=2, $f(n)=1=n^1$, d=1 [power of n in f(n)]

a **b^d**

2 2^1

2 = 2

So $a=b^d$

So from eqn (2) we get the relation as:

$T(n) = \Theta(n^d \log n)$

$= \theta(n^1 log_2 n)$

$= \theta(n \, log_2 n)$

$\therefore$ T(n)ϵ Θ(n $\log_2$n)

Time complexity using substitution method:

$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n$

$= 2T\left(\frac{n}{2}\right) + n$

$= 2\left[2T\left(\frac{n}{4}\right) + \frac{n}{2}\right] + n$

$= 2^2 T\left(\frac{n}{2^2}\right) + n + n$

$= 2^2\left[2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\right] + 2n$

$= 2^3 T\left(\frac{n}{2^3}\right) + 3n$

…………………

…………………

$= 2^i T\left(\frac{n}{2^i}\right) + i.n \ldots\ldots(1)$

$T\left(\frac{n}{2}\right) = 2T\left(\frac{n/2}{2}\right) + \frac{n}{2}$

$= 2T\left(\frac{n}{2^2}\right) + \frac{n}{2}$

$T\left(\frac{n}{2^2}\right) = 2T\left(\frac{n/2^2}{2}\right) + \frac{n}{2^2}$

$= 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}$

Inorder to get initial condition make $2^i = n$…..(2)

$2^i = n$

Taking log_2 on both sides,

$log_2 2^i = log_2 n$

i $log_2 2 = log_2$ n

i=log_2 n …..(3)

equating (2) and (3) in (1) we will get,

$\therefore T(n) = nT\left(\frac{n}{n}\right) + nlog_2$ n

$T(n) = nT(1) + nlog_2$ n

$\therefore T(n) = n * 0 + nlog_2$ n

$\therefore T(n) = nlog_2$ n

Represent it using Θ notation

$\therefore T(n) \epsilon \, \theta\big(g(n)\big)$ ➔ **$\therefore$ T(n)ϵ Θ(n$\log_2$ n)**