DIVIDE AND CONQUER

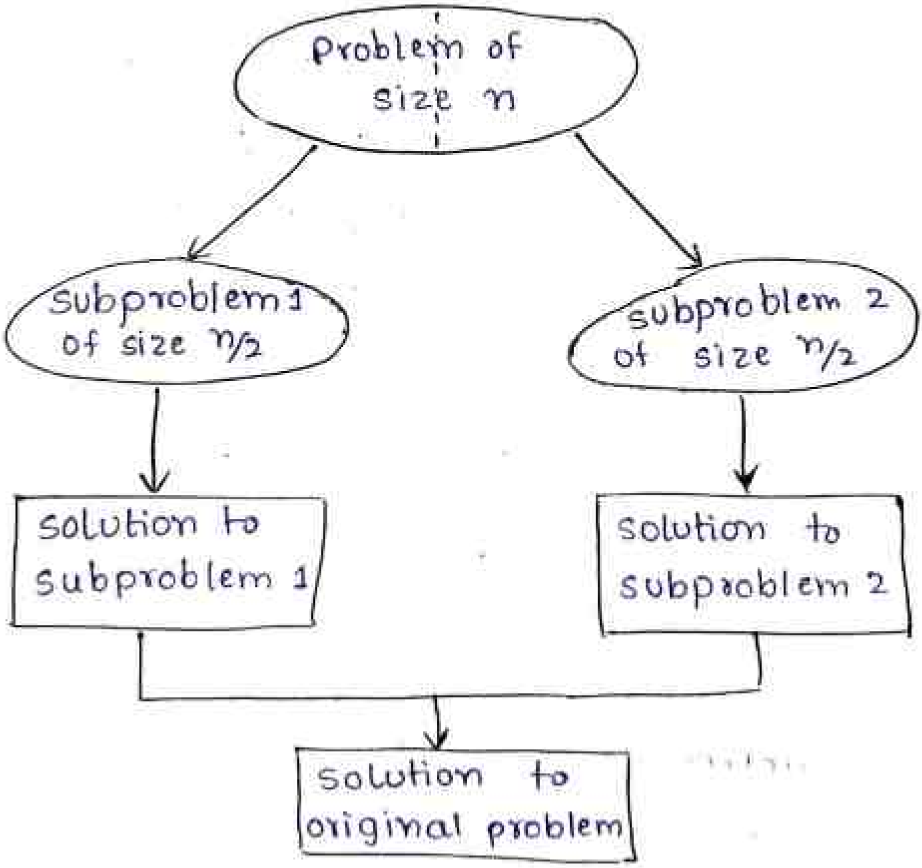
Divide and conquer is a technique of designing an algorithm that consists of dividing a problem into smaller subproblem hoping that solution for smaller subproblems are easier to find.

It involves the following steps for solving the problem.

1) Divide: A problem instance is divided into several smaller instances of the same problem, ideally of same size.

2) Conquer: Subproblems are conquered by solving them recursively.

3) Combine: Solutions of subproblems are combined to get a solution to original problem.



For eg, consider a problem of finding sum of n numbers from 1 to n.

$$S_n = a_0 + a_1 + \dots + a_{n-1}$$
$$= (a_0 + a_1 + \dots + a_{\lfloor n/2 \rfloor - 1}) + (a_{\lfloor n/2 \rfloor} + \dots + a_{n-1})$$

A problem instance of size n is divided into two problem instances of size $n/2$. In general, an instance of size n can be divided into b instances of size n/b, out of which a instances will be solved, $(a \geq 1, b \geq 1)$

This results in following recurrence relation,

$$T(n) = aT(n/b) + f(n)$$

where $f(n)$ is the function indicating time needed for division of problem into smaller parts and combining their solutions. This is the general divide and conquer recurrence relation. The order of growth of $T(n)$ depends on a, b & $f(n)$.

The order of growth computation for divide and conquer algorithms can be simplified using Master's Theorem.

MASTER THEOREM:

If $f(n) \in \Theta(n^d)$ with $d \geq 0$ in recurrence relation $T(n) = aT(n/b) + f(n)$, then

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

Eg i) If $A(n) = 2A\left(\frac{n}{2}\right) + 1$

then $a = 2$, $b = 2$ $f(n) = 1$ $d = 0$

$a = 2$ $b^d = 2^0 = 1$

$2 > 1$

$a > b^d$

$\therefore$ $A(n) \in \Theta(n^{\log_2 2})$

$A(n) \in \Theta(n)$

For Binary Recursion, $A(n) = A\left(\frac{n}{2}\right) + 1$

$a = 1$, $b = 2$, $f(n) = 1$, $d = 0$

$a = 1$, $b^d = 2^0 = 1$

$a = b^d$

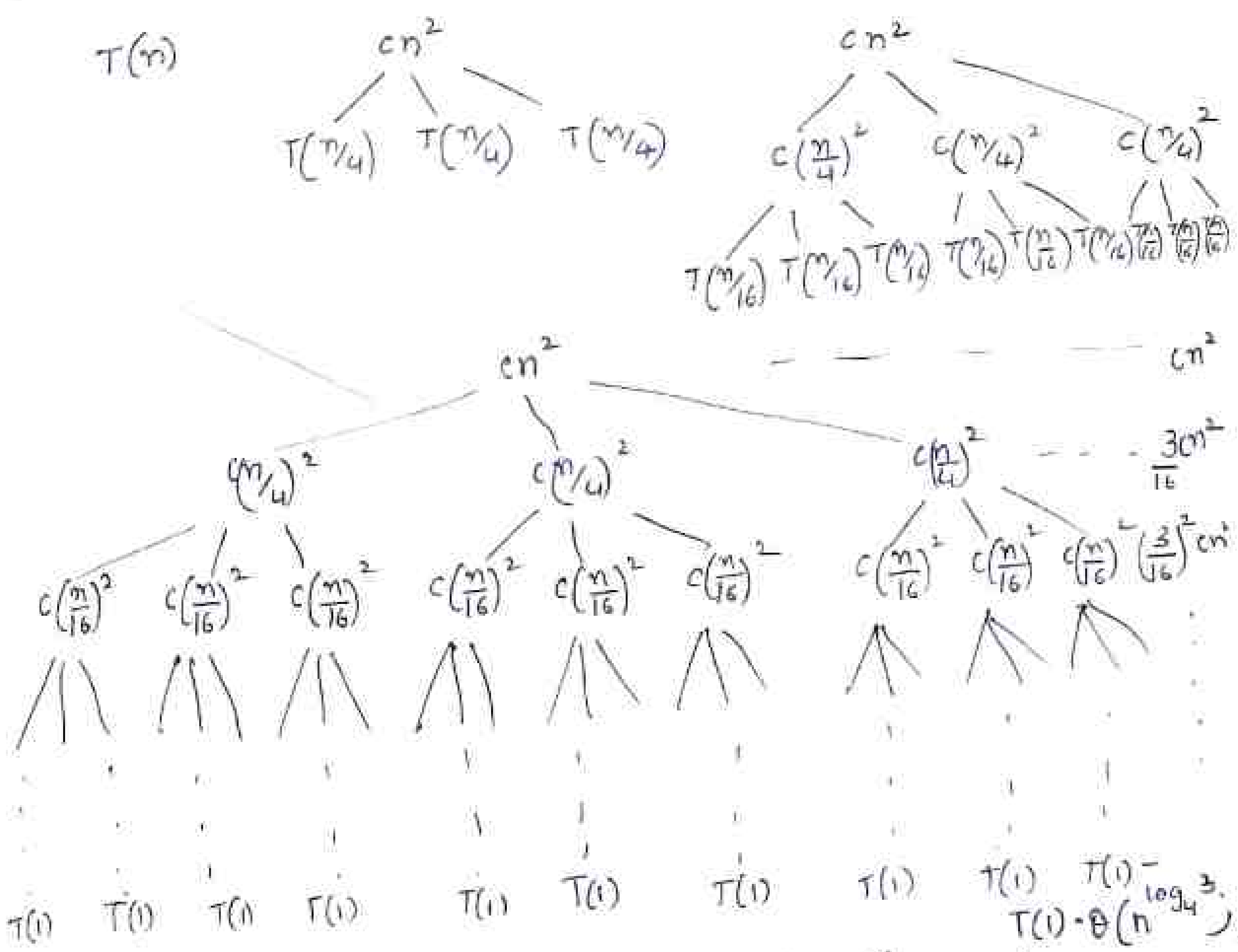
$\therefore$ $A(n) \in (n^d \log n)$

$A(n) \in (\log n)$

The recursion method for solving recurrences:

In the recursion tree, each node represents the cost of a single subproblem somewhere in the set of recursive function invocations. We sum the costs within each level of the tree to obtain a set of per-level costs, and then we sum per-level costs to determine the total cost.

$$T(n) = 3T(n/4) + \Theta(n^2)$$
$$= 3T(n/4) + cn^2 \quad , \; c > 0$$



The subproblem size at level i is $\frac{n}{4^i}$

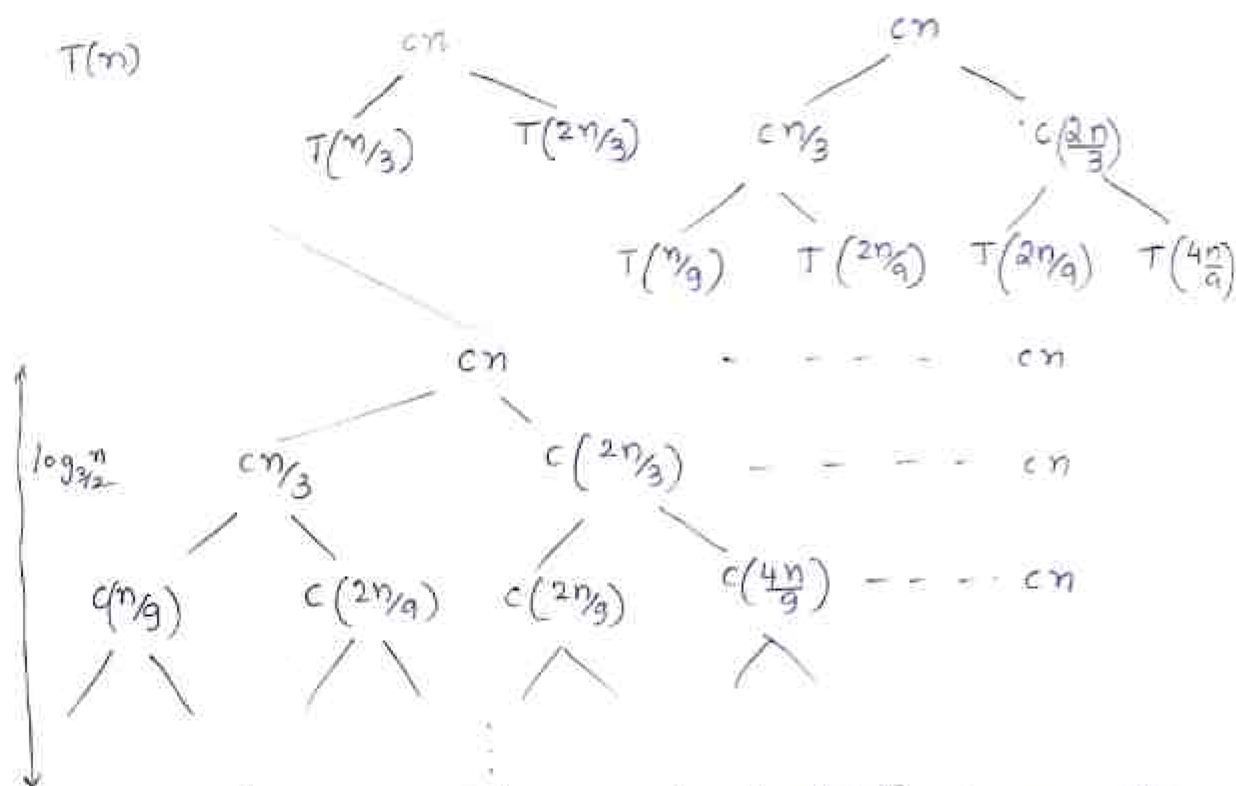
$$\frac{n}{4^i} = 1 \Rightarrow n = 4^i \Rightarrow i = \log_4 n$$

The last level corresponds to level $\log_4 n$ and no. of nodes in last level is $3^{\log_4 n} = n^{\log_4 3}$

$$T(n) = cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3})$$
$$= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3})$$
$$= cn^2 \left[\frac{(3/16)^{\log_4 n} - 1}{3/16 - 1}\right] + \Theta(n^{\log_4 3})$$
$$< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3})$$
$$= \frac{1}{1 - 3/16} cn^2 + \Theta(n^{\log_4 3})$$
$$= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3})$$
$$T(n) \leq 16/13\, cn^2 + \Theta(n^{\log_4 3}) = \underline{\underline{O(n^2)}}$$

$$T(n) = T(n/3) + T(2n/3) + O(n)$$

$$T(n) = T(n/3) + T(2n/3) + cn$$



The longest path from root to leaf is $n \to \frac{2n}{3} \to \left(\frac{2}{3}\right)^2 n \to \cdots \to 1$

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

$$n = \left(3/2\right)^k$$

$$\Rightarrow k = \log_{3/2} n$$

The height of the tree is $\log_{3/2} n$

Since the cost of each level is 'cn', the total cost would be $cn \cdot \log_{3/2} n$

$$T(n) = O(cn \log_{3/2} n)$$

$$= \underline{\underline{O(n \lg n)}}$$

The master method for solving recurrences:

$$T(n) = a\,T(n/b) + f(n)$$

where $a \geq 1$, $b > 1$ are constants and $f(n)$ is an asymptotically positive function.

Master's Theorem:

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function and let $T(n)$ be defined on the non-negative integers by the recurrence

$T(n) = aT(n/b) + f(n)$

where we interpret n/b as either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds

1) If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$

2) If $f(n) = \Theta(n^{\log_b a})$ then $T(n) = \Theta(n^{\log_b a} \lg n)$

3) If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \le cf(n)$ for some constant $c < 1$, and all sufficiently large n, then $T(n) = \Theta(f(n))$

1) $T(n) = 9T(n/3) + n$

$a = 9, \quad b = 3, \quad f(n) = n$

$n^{\log_3 9} = n^2 = \Theta(n^2)$

$f(n) = O(n^{\log_3 9 - \epsilon})$ where $\epsilon = 1$

$\Rightarrow T(n) = \Theta(n^2)$

2) $T(n) = T(2n/3) + 1$

$a = 1, \quad b = 3/2, \quad f(n) = 1$

$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$

$f(n) = \Theta(n^{\log_b a})$

$\Rightarrow T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(\lg n)$

3) $T(n) = 3T(n/4) + n\lg n$

$a = 3, \quad b = 4, \quad f(n) = n\lg n$

$n^{\log_b a} = n^{\log_4 3} = n^{0.793}$

$f(n) = \Omega(n^{\log_4 3 + \epsilon})$, where $\epsilon \approx 0.2$

Also, $3(f(n/4)) \le cf(n)$

$3 \cdot (n/4) \lg(n/4) \le (3/4) n\lg n, \quad c = 3/4 < 1$

$\Rightarrow T(n) = \Theta(n\lg n)$

4) $T(n) = 2T(n/2) + n\lg n$

$a = 2, \quad b = 2, \quad f(n) = n\lg n$

$n^{\log_b a} = n^{\log_2 2} = n$

$f(n) = \Omega(n^{\log_2 2 + \varepsilon})$ = ε should be > 0

But $\frac{f(n)}{n^{\log_b a}} = \frac{n\lg n}{n} = \lg n$ and it is asymptotically less that n^{ε} for any $\epsilon > 0$

Hence it falls between case 2 & 3.

HW:

$T(n) = 2T(n/2) + \theta(n)$, $T(n) = 8T(n/2) + \theta(n^2)$

$T(n) = 7T(n/2) + \theta(n^2)$

Merge Sort:

Mergesort sorts a given array $A[0 \dots n-1]$ by dividing it into two halves $A[0 \dots \lfloor n/2 \rfloor - 1]$ and $A[\lfloor n/2 \rfloor \dots n-1]$, sorting each of them recursively and the merging the two smaller sorted arrays into a single sorted one.

Algorithm Mergesort $(A[0 \dots n-1])$
// Sorts array $A[0 \dots n-1]$ by recursive mergesort
// Input: An array $A[0 \dots n-1]$ of orderable elements
// Output: Array $A[0 \dots n-1]$ sorted in increasing
// order

```
if n > 1
    copy A[0 ... ⌊n/2⌋ - 1] to B[0 ... ⌊n/2⌋ - 1]
    copy A[⌊n/2⌋ ... n-1] to C[0 ... ⌈n/2⌉ - 1]
    Mergesort (B[0 ... ⌊n/2⌋ - 1])
    Mergesort (C[0 ... ⌈n/2⌉ - 1])
    Merge (B, C, A)
```

Algorithm Merge$(B[0 \dots p-1], C[0 \dots q-1], A[0 \dots p+q-1])$
// merges two sorted arrays into one sorted array
// Input: Array $B[0 \dots p-1]$ and $C[0 \dots q-1]$ both
// being sorted
// Output: Sorted array $A[0 \dots p+q-1]$ of elements
// of B and C

```
i ← 0
j ← 0
k ← 0
while i < p and j < q do
    if B[i] ≤ C[j]
            A[k] ← B[i]
            i ← i + 1
    else
```

$A[k] \leftarrow C[j]$

$j \leftarrow j+1$

$k \leftarrow k+1$

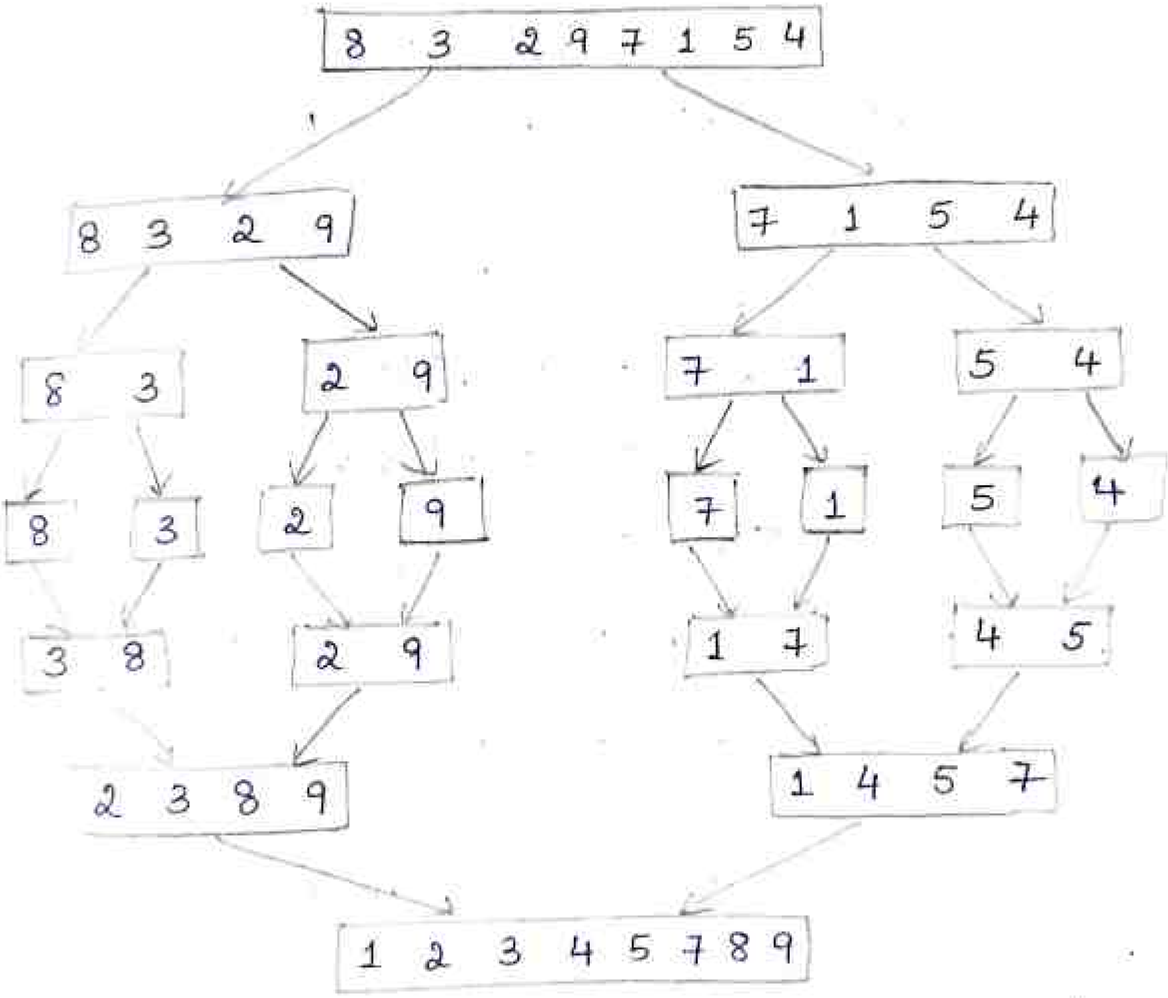
if $i = p$

copy $C[j \ldots q-1]$ to $A[k \ldots p+q-1]$

else copy $B[i \ldots p-1]$ to $A[k \ldots p+q-1]$

Problem Tracing:

List: 8 3 2 9 7 1 5 4



Analysis:

It is clear from the algorithm that problem instance is divided into 2 parts. The recurrence relation for the algorithm can be written as,

$$C(n) = 2\,C(N/2) + C_{merge}(n) \quad \text{for } n > 1,$$

$$C(1) = 0$$

In the worst case, $C_{merge}(n) = n - 1$

Hence.

$$C_{worst}(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2C_{worst}\left(\frac{n}{2}\right) + n - 1 & \text{if } n > 1 \end{cases}$$

According to master's Theorem.

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

By comparison,

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

$$a = 2, \quad b = 2, \quad f(n) = n - 1$$

$$f(n) \in \Theta(n) \quad \Rightarrow \quad d = 1$$

$$a = 2 \qquad b^d = 2^1 = 2$$

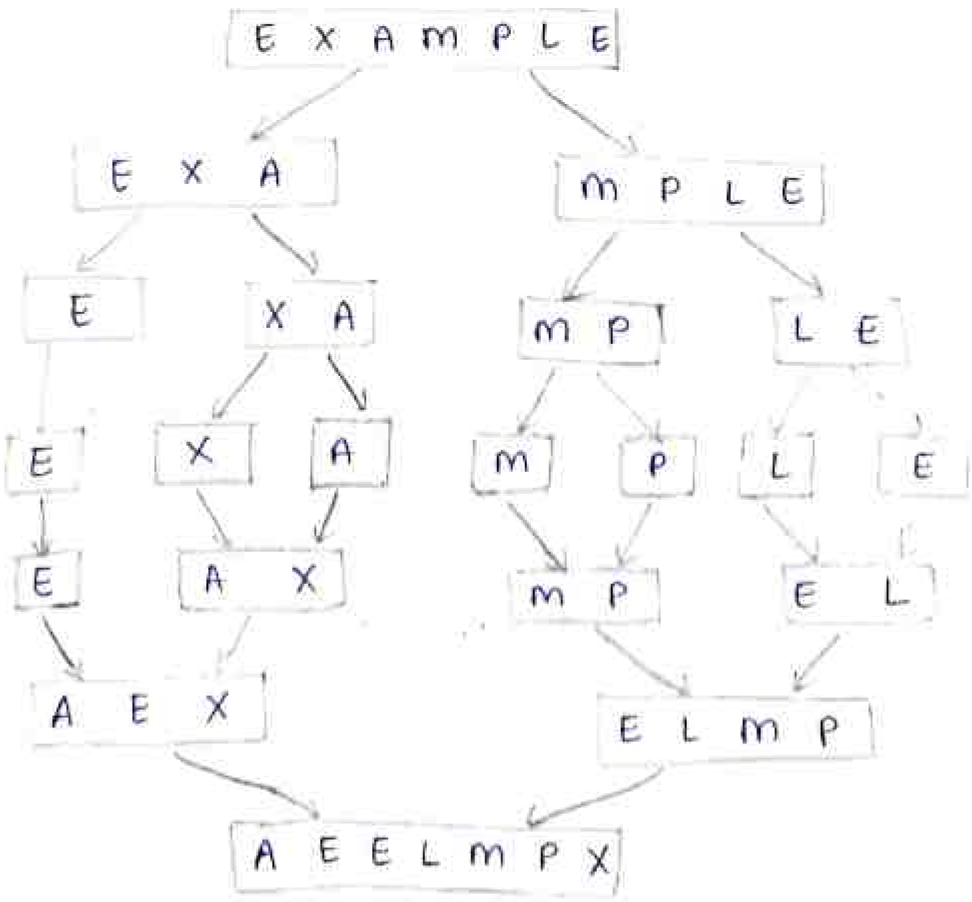
$$a = b^d$$

$$\Rightarrow \quad T(n) \in \Theta(n^d \log n)$$

$$\in \Theta(n^1 \log n)$$

$$T(n) \in \Theta(n \log n)$$

HW : Sort E, X, A, M, P, L, E by mergesort.



QuickSort:

Quicksort is another sorting algorithm that is based on divide and conquer approach. While mergesort divides the array elements according to their position in the array, quicksort divided them according to their value. It rearranges the elements of a given array A[0...n-1] to achieve partition, where all elements before partition position s are smaller than or equal to A[s] and all elements after position s are larger than or equal to A[s].

$$\underbrace{A[0] \;\dots\; A[s-1]}_{\leq A[s]} \quad A[s] \quad \underbrace{A[s+1] \;\dots\; A[n-1]}_{\geq A[s]}$$

After the partition is achieved, A[s] will be in its final position in sorted array and sorting is continued on the two subarrays preceding and succeeding A[s] independently.

Algorithm Quicksort (A[l...r])
// sorts a subarray by quicksort
// I/P: A subarray A[l..r] of A[0..n-1] defined by
// left and right indices l & r
// O/P: Subarray A[l..r] sorted in increasing order

```
if l < r
    s ← Partition(A[l...r]) // s is the partition position
    Quicksort (A[l ... s-1])
    Quicksort (A[s+1 ... r])
```

Partition of A[0...n-1] for its subarray A[l...r] can be obtained using the following algorithm. First, we select a value wrto which we are going to divide the array, called the pivot element. It is the first element of the subarray. P ← A[l].

Two scans are performed on the subarray. one from left to right and other from right to left comparing subarray elements with pivot. In left to right scan, we scan until an element larger than pivot is encountered and in right to left scan, we scan until element smaller than pivot is encountered. If $i < j$ then swapping of elements of $A[i]$ and $A[j]$ is performed and if $i > j$ then swap $A[l]$ with $A[j]$ to get fixed position for pivot $A[l]$

<u>Algorithm</u> Partition $(A[l \ldots r])$

// Partitions a subarray using first element as pivot
// Input: A subarray $A[l \ldots r]$ of $A[0 \ldots n-1]$ defined by
// left and right index l & r.
// O/P: A partition of $A[l \ldots r]$, with split position
// returned as value

```
p ← A[l]
i ← l
j ← r + 1
repeat
    repeat i ← i + 1 until A[i] ≥ p
    repeat j ← j - 1 until A[j] ≤ p
    swap (A[i], A[j])
until i ≥ j
swap (A[i], A[j]) // undo last swap when i ≥ j
swap (A[l], A[j])
return j
```

<u>Problem</u>: Sort the array 5, 3, 1, 9, 8, 2, 4, 7 using quicksort

0	1	2	3	4	5	6	7	
l	i						j	j
5	3	1	9	8	2	4	7	
p			i			j		
$\boxed{5}$	3	1	9	8	2	4	7	
			i			j		
$\boxed{5}$	3	1	4	8	2	9	7	// swap A[i] & A[j]
				i	j			
$\boxed{5}$	3	1	4	8	2	9	7	// swap A[i] & A[j]
				i	j			
$\boxed{5}$	3	1	4	2	8	9	7	
				j	i			
$\boxed{5}$	3	1	4	2	8	9	7	// swap p and A[j]
2	3	1	4	5	8	9	7	

p i j

$\boxed{2}$ 3 1 4

i j

$\boxed{2}$ 3 1 4 // swap A[i] and A[j]

i j

$\boxed{2}$ 1 3 4

j i

$\boxed{2}$ 1 3 4 // swap p and A[j]

1 $\boxed{2}$ 3 4

p ij

$\boxed{3}$ 4

j i

$\boxed{3}$ 4 // swap p with A[j]

3 4

p i j

$\boxed{8}$ 9 7 //swap A[i] & A[j]

i j

$\boxed{8}$ 7 9

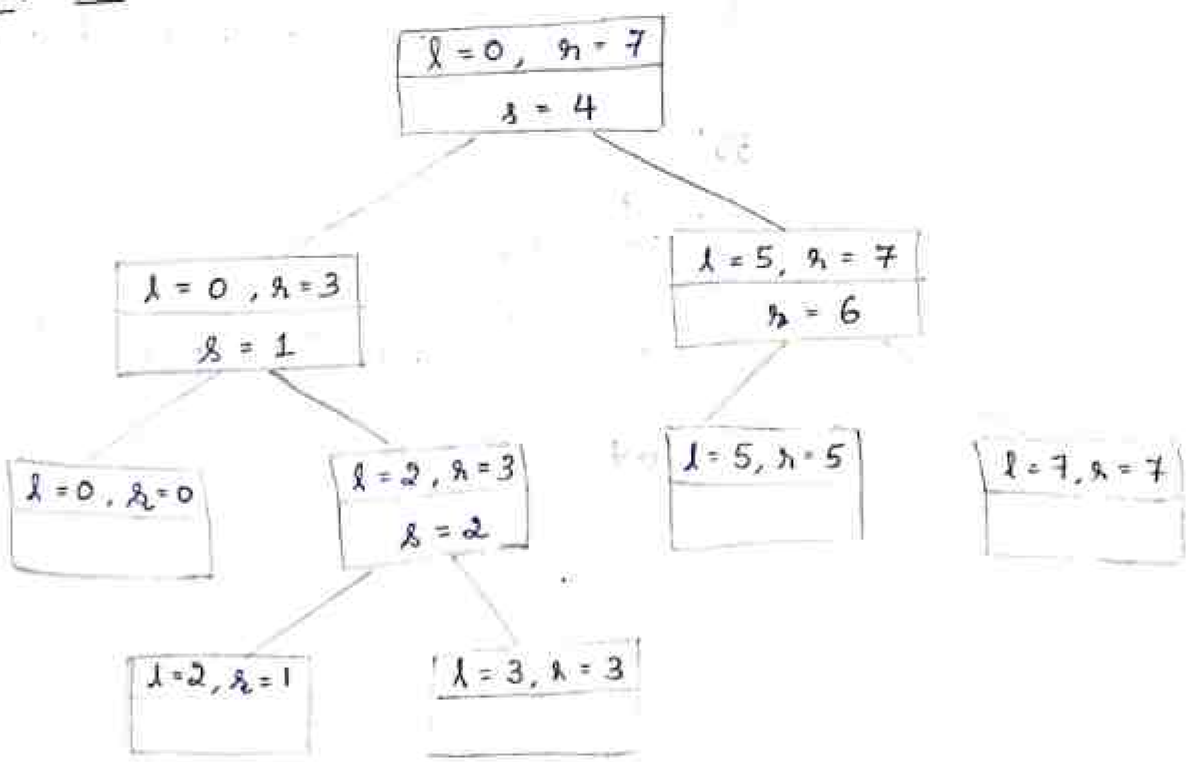
j i

$\boxed{8}$ 7 9 // swap p with A[j]

7 8 9

∴ Sorted array : 1 2 3 4 5 7 8 9

<u>Tree of recursive calls:</u>



Problem 2: Sort 44, 75, 23, 43, 55, 12, 64, 77, 33 in ascending order using Quicksort

0	1	2	3	4	5	6	7	8	
p	i							j	
$\boxed{44}$	75	23	43	55	12	64	77	33	// swap A[i] & A[j]
	i							j	
44	33	23	43	55	12	64	77	75	
				i	j				
44	33	23	43	55	12	64	77	75	// swap A[i] & A[j]
				i	j				
44	33	23	43	12	55	64	77	75	
				j	i				
44	33	23	43	12	55	64	77	75	//swap P & A[j]
12	33	23	43	44	55	64	77	75	

p i j

$\boxed{12}$ 33 23 43

j i

12 33 23 43 // swap p with A[j]

12 33 23 43

p i j

$\boxed{33}$ 23 43

j i

33 23 43 // swap p with A[j]

23 33 43

p i j

$\boxed{55}$ 64 77 75

j i

55 64 77 75 // swap p with A[j]

55 64 77 75

p i j

$\boxed{64}$ 77 75

j i

64 77 75 // swap p with A[j]

64 77 75

p ij

$\boxed{77}$ 75

j i

77 75 // swap p with A[j]

75 77

Sorted Array: 12, 23, 33, 43, 44, 55, 64, 75, 77