AKTU

B.Tech 5th Sem

GATEWAY CLASSES

CS IT & CS Allied

DAA : Design & Analysis Of Algorithm

UNIT-1: Introduction

Lecture-1

Today's Target

- Introduction to Algorithms
- Characteristics of Algorithms
- Difference between a Priori and a Posteriori analysis
- Types of Time functions
- Order of growth of functions
- Comparison of functions

By Dr. Nidhi Parashar Ma'am

- M.Tech Gold Medalist
- Net Qualified

Course Details

1	Recorded Video Lectures (100 % Syllabus Coverage)
2	Pdf Notes
3	Lecture wise DPP(if required)
4	Unit wise set of PYQs
5	Important Questions according to new Syllabus

Paid Courses are available in Gateway Classes Application

Link in Description

Computer Science & Engineering , Information Technology & Computer Science Allied

(BCS-503 - Design and Analysis of Algorithm)

Unit-I : Introduction

AKTU : Syllabus

Introduction: Algorithms, Analyzing Algorithms, Complexity of Algorithms, Growth of Functions, Performance Measurements, Sorting and Order Statistics - Shell Sort, Quick Sort, Merge Sort, Heap Sort, Comparison of Sorting Algorithms, Sorting in Linear Time.

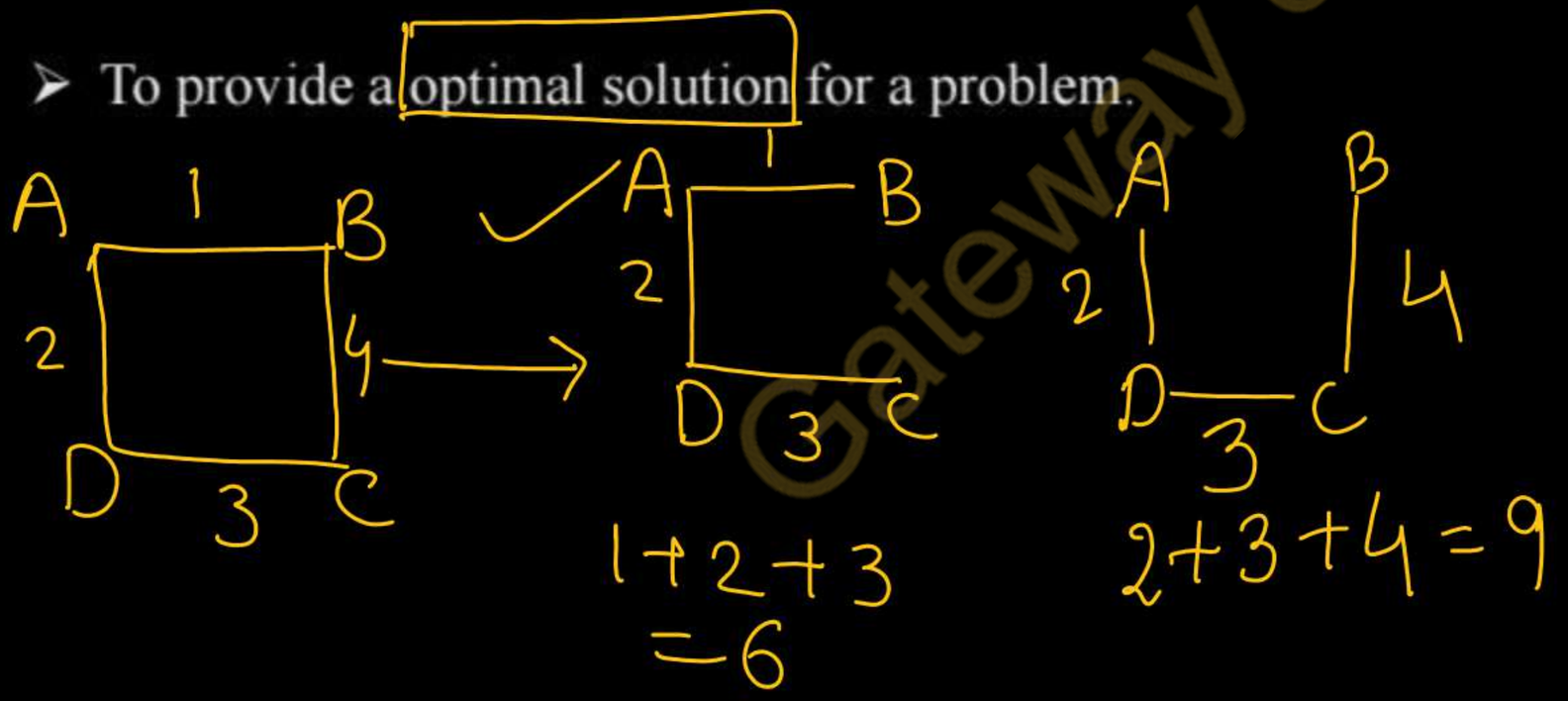
Why Design and Analysis of Algorithms?

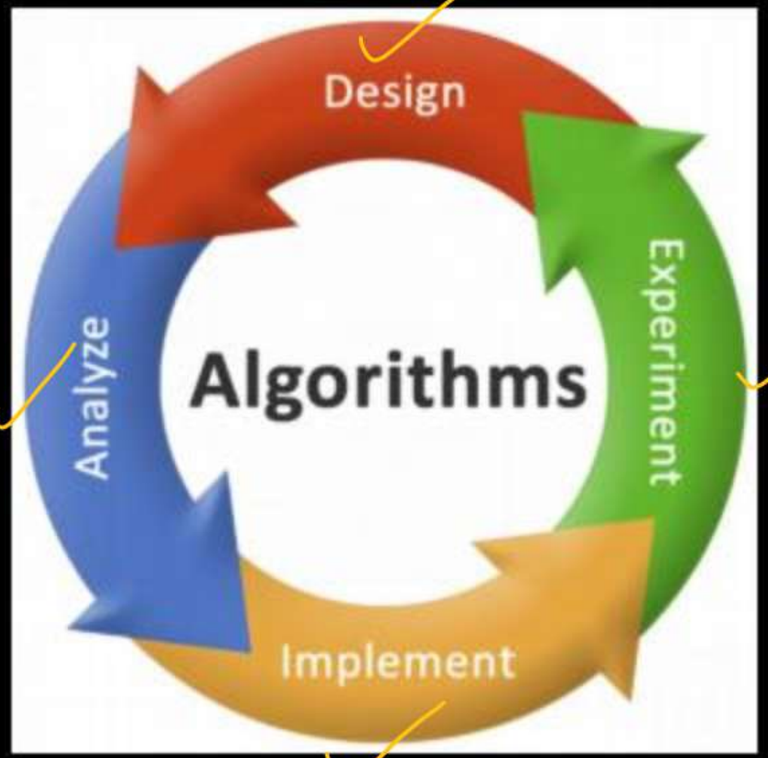
- First Design then implement policy.

Algorithm + PL = Program

- To provide feasible solutions for all types of problems based on efficient complexity (time and space complexity)

- To provide a optimal solution for a problem.





Algorithm

- A step-by-step procedure in a certain order to get the desired output.
- Independent of underlying languages.

Ex:-Algorithm to add two numbers

1. Start
2. Input A and B.
3. Calculate SUM := A + B.
4. Print SUM.
5. End.

Characteristics of Algorithm

- Input: Must take 0 or more input
- Output: Should always give one or more output
- Feasible: Can be run using available resources
- Finiteness: The process should terminate after finite number of steps
- Definiteness/Unambiguous: Each instruction should be clear
- Effectiveness: Every instruction must be basic enough to be carried out theoretically (Only necessary operations are included)
- Deterministic: Produce the same output for the same input

Difference between a Priori and a Posteriori analysis

A priori analysis	A Posteriori analysis
➢ Before execution of an algorithm.	➢ After execution of an algorithm.
➢ Independent of programming language and hardware.	➢ Dependent on programming language and hardware.
➢ Gives approximate answer.	➢ Give exact answer.
➢ Cheaper than Posteriori Analysis.	➢ Costlier than priori analysis.
➢ Maintenance Phase is not required to tune the algorithm.	➢ Maintenance Phase is required to tune the algorithm

Performance Analysis of Algorithm

1 unit ⇒ 4 bytes
4 × 4 = 16 byts

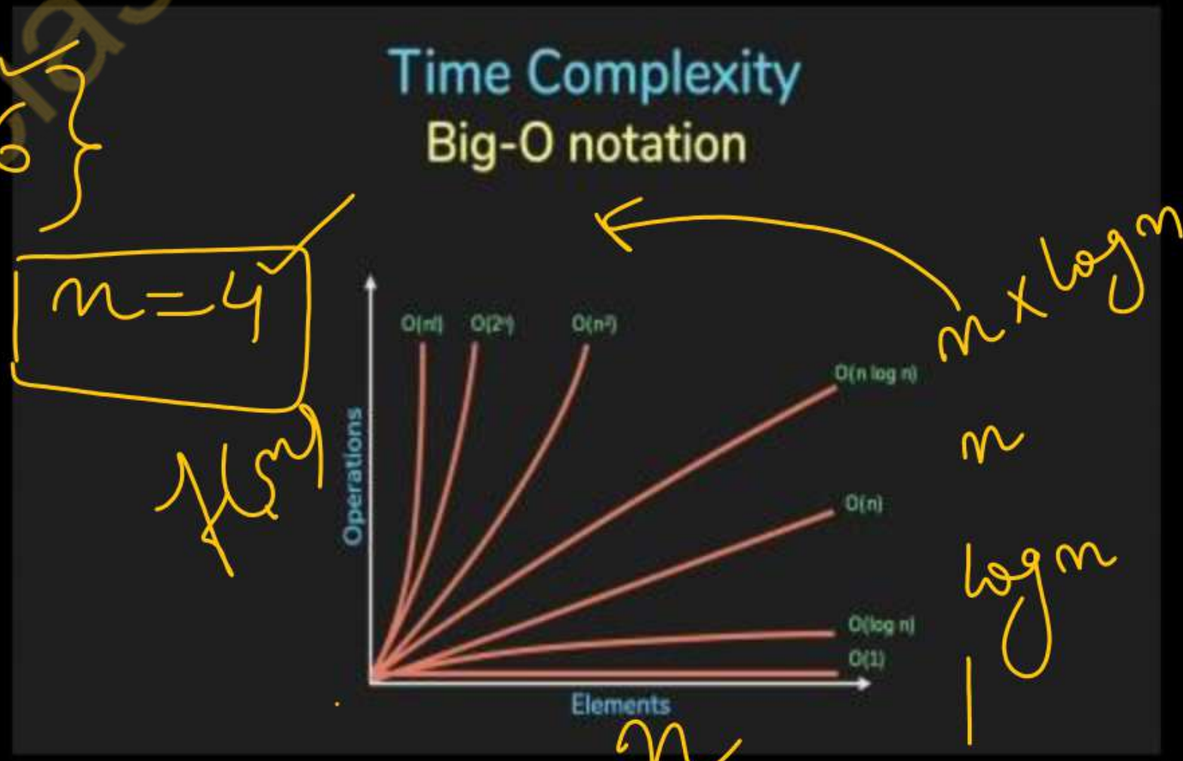
There are two types are: Time complexity and space complexity

- **Time Complexity** : It is the amount of time taken by an algorithm to run, as a function of the length of the input. It measures the time taken to execute each statement of code in an algorithm.

- **Space Complexity**: The amount of memory used by a program to execute it is represented by its space complexity as a program requires memory to store input data and temporal values while running.

a[4] = {1, 8, 3, 6}

n = 4



Time and space complexity are represented using functions.

Growth of Function

- Resources for an algorithm are usually expressed as a function regarding input.
- To study Function growth efficiently, we reduce the function down to the important part.

Let $f(n) = an^2+bn+c$

- Here n^2 term dominates the function when n gets sufficiently large.
- Dominant terms (highest order term concerning n) are considered and all constants and coefficients are ignored.
- $f(n)$ is said to be **asymptotically equivalent** to n^2 as $n \rightarrow \infty$", and here is written symbolically as $f(n) \sim n^2$.

Types of Time Functions

- Constant — $O(1)$
- Logarithmic — $O(\log_2 n)$
- Linear — $O(n)$
- Quadratic — $O(n^2)$
- Cubic — $O(n^3)$
- Exponential — $O(2^n)$

$f(n)=2$
$f(n)=100$
$f(n)=3420$
$\Rightarrow O(1)$

$f(n)=2n+1$
$f(n)=40n+1000$
$f(n)=200n+n/20+40$
$\Rightarrow O(n)$

$f(n)=2n^2+1$
$f(n)=200n^2+10n+40$
$\Rightarrow O(n^2)$

$f(n)=5n^3+200n^2+10n+40$

$f(n)=2^n+20n^2+10n+40$

$f(n)=2\log n+1$
$f(n)=O(\log n)$

Order of Growth of Functions

- **Order of growth** is how the time of execution depends on the length of the input.
- Order of growth will help to compute the running time with ease.

$$1 < \log\log n < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \ldots\ldots < 2^n < 3^n < n^n$$

	log n	n	n^2	2^n
n = 1	0 $\log_2 1$	1	1	2
n = 2	$\log_2 2 = 1$	2	4	4
n = 4	$\log_2 4 = 2$	4	16	$2^4 = 16$
n = 8	$\log_2 8 = 3$	8	64	$2^8 = 256$

$n^2 < 2^n$

$n^{100} < 2^n$

$n^{1000} < 2^n$

$\frac{100}{n} < 10 < \log n < \sqrt{n} < n$

Find order of growth for the functions

	10	n	$\sqrt{n}$	$\log_2 n$	$100/n$
$n=1$	10	1	1	$\log 1 = 0$	$100/1 = 100$
$n=2$	10	2	>1	1	$100/2 = 50$
$n=4$	10	4	2	$\log_2 4 = 2$	$100/4 = 25$
$n=16$	10	16	4	$\log_2 16 = 4$	$100/16 =$
$n=64$	10	64	8	$\log_2 64 = 6$	$100/64 =$

AKTU

B.Tech 5th Sem

GATEWAY CLASSES

CS IT & CS Allied

DAA : Design & Analysis Of Algorithm

UNIT-1: Introduction

Lecture-2

Today's Target

- Comparison of functions
- Complexity analysis of algorithms using:
  Frequency count method

By Dr. Nidhi Parashar Ma'am
- M.Tech Gold Medalist
- Net Qualified

Comparison of functions

$f(n) = n^2$ and $g(n) = n^3$

Prove that f(n) is asymptotically smaller than g(n).

(I)

n	n^2	n^3
n=1	1	1
n=2	4	8
n=3	8	27
n=4		

$n^2 < n^3$

II

$\log(n^2) = 2\log n$

$\log(n^3) = 3\log n$

$2 < 3$

$n^2 < n^3$

III

n^2 $\quad$ n^3

n^2 $\quad$ $n \cdot n^2$

$1 < n$

$n^2 < n^3$

Comparison of functions

$f(n) = n^2$ and $g(n) = n \log n$
Prove that $f(n)$ is asymptotically larger than $g(n)$.

		n^2	$n \log n$
$n=1$	$\log n = 0$	$n \cdot n$	$n \log n$
	$\log_2 1 = 0$	n	$\log n$
$n=2$	$\log_2 2 =$		
$n=4$	$\log_2 4 \Rightarrow \log_2 2^2$		

$= 2 \log_2 2$

$= 2 \times 1$

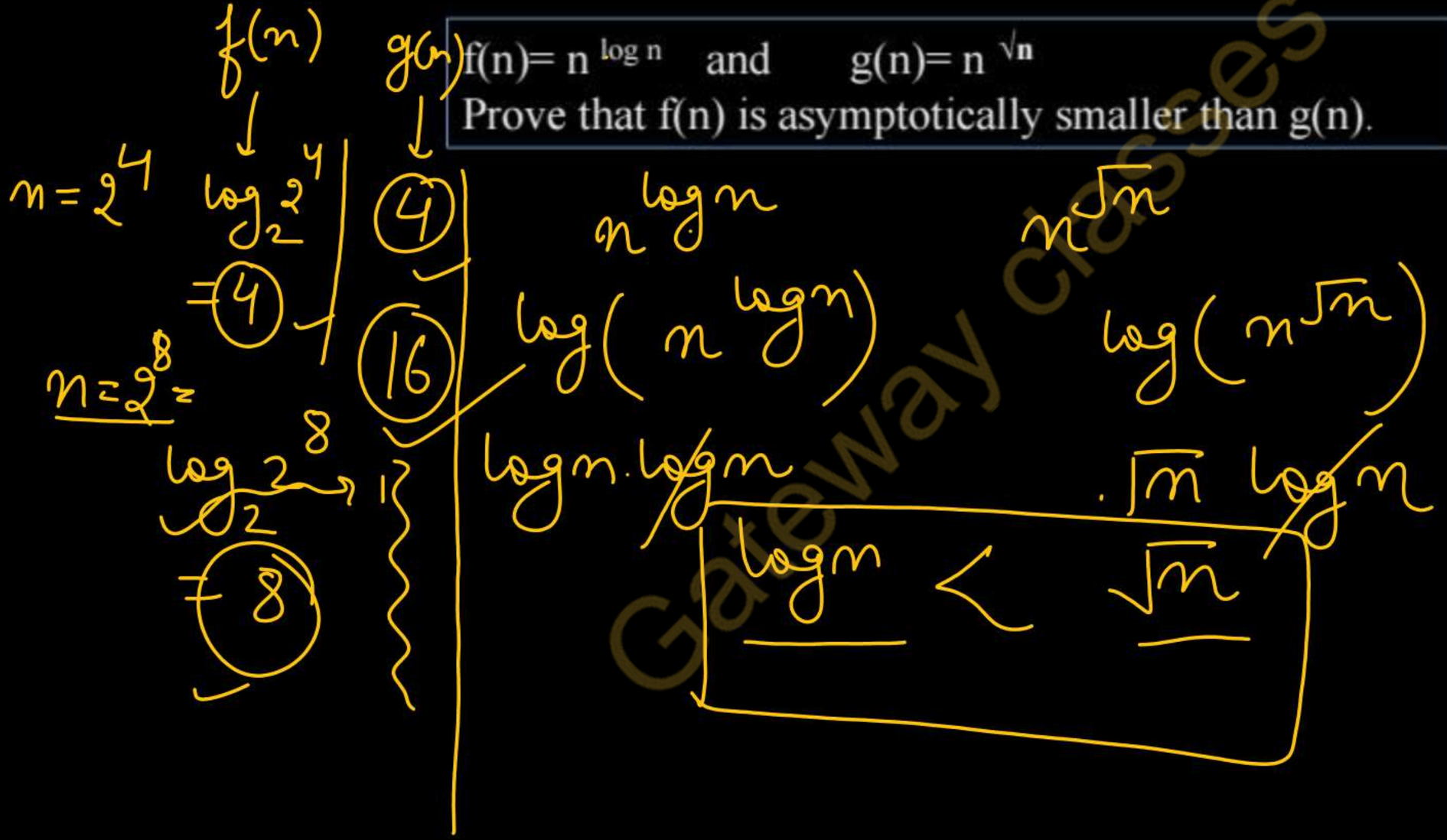
$= 2$

$n > \log n$

Comparison of functions

$f(n) = n^{\log n}$ and $g(n) = n^{\sqrt{n}}$

Prove that f(n) is asymptotically smaller than g(n).



Comparison of functions

$f(n)= 3n^{\sqrt{n}}$ and $g(n)= 2^{\sqrt{n}\log n}$

$\log n^a = a \log n$

$f(n) = 3\, n^{\sqrt{n}}$

$g(n) = 2^{\sqrt{n}\log n}$

$= 2^{\log_2 (n^{\sqrt{n}})}$

$= (n^{\sqrt{n}})^{\log_2 2}$ $\quad \log_2 2 = 1$

$3\, n^{\sqrt{n}}$

$3 > 1$

$n^{\log_a b} = b^{\log_a n}$

Complexity Analysis of Algorithms

- To analyze a programming code or algorithm, we must notice that each instruction affects the overall performance of the algorithm and therefore, each instruction must be analyzed separately to analyze overall performance.
- Finding time and space complexity

Frequency Count Method

- It counts the number of times each instruction is executed.
- It keeps track of how many times a computer follows each instruction. Each step gets a time unit, starting from 1. The time it takes for the algorithm to run is described by a function called f(n), where n is the size of the data.

Example:

Start
Input a and b.
Temp=a
a=b
b=temp
End

1 1 1 1

$f(n) = O(4) = O(1)$

```
for ( i =0; i < n; i++)
{
   printf ("%d", i);
}
```

1 (n+1) n n

```
for ( i =1; i < n; i= i+2 )
{
   printf ("%d", i);
}
```

1 (n+1) n n

$f(n) = 1 + n+1 + n + n$

$= 3n + 2$

$\Rightarrow f(n) = O(n)$

$n = 3$

i = 0 ✓ 0 < 3 ✓

i = 1 1 < 3 ✓

i = 2 2 < 3 ✓

i = 3 ✗ 3 < 3 ✗

$f(n) = O(n)$

$O(n)$

n

n+1

n−1

2n+3

```
for ( i =0; i < n; i=i+20)
{
   printf ("%d", i);
}
```

1 | n+1 | n

$f(n) = O(n)$

```
for ( i = n; i >=1; i--)
{
   printf ("%d", i);
}
```

n+1 | n | $i--$

$\Rightarrow i = i-1$

$f(n) = O(n)$

```
for ( i = n; i >=1; i=i-10)
{
   printf ("%d", i);
}
```

$f(n) = O(n)$

```
for ( i =1; i < n; i= i*2)
{
   printf ("%d", i);
}
```

① | $\log n$ | $\log n$

$f(n) = \log n + \log n + 1$

$f(n) = O(\log n)$

1	$i = i*3$
2	$\log_3 n$
2^2	
2^3	
$\vdots$	
2^k	$= n$

$2^k = n$

$K \log_2 2^1 = \log_2 n$

$K = \log_2 n$

Imp

```
for ( i = n; i >= 1; i = i /2)
{
    printf ("%d", i);
}
```

i, n

O(2)

$\Rightarrow O(1)$

(I) $\rightarrow \frac{n}{2}$

(II) $\rightarrow \frac{n}{2^2}$

(III) $\frac{n}{2^3}$

K-times

$n/2^K$

$\frac{n}{2^K} = 1$

$n = 2^K$

$K = \log n$

$\Rightarrow$

Space = O(n)

A → n

n → 1 | i = 1

```
Sum(A , n)
{
    s=0;                          ——— (1)
    for ( i=0; i < n; i++)        ——— (n+1)
    {                             ——— (n)
        s= s + A[i] ;
    }
    return(s);                    ——— (1)
}
```

$f(n) = 1 + n + 1 + n + 1$

$= 2n + 3$

$f(n) = O(n)$

A, B, C
$(n \times n)$

```
Add(A , B, n)
{
    for ( i =0; i <= n; i++)  ⟶ (n+1)
    {
        for ( j=0; j < n; j++)
        {
            c[i][j]= A[i][j] +B[i][j];
        }
    }
}
```

$n \times (n+1)$
$n \times n$

$f(n) = O(n^2)$

Space Complexity

$f(n) = O(n^2)$

C ⟶ n^2
A ⟶ n^2
B ⟶ n^2
n ⟶ 1
i ⟶ 1

$3n^2 + 2$

$f(n) = (n+1) + n^2 + n + n^2$

$= 2n^2 + 2n + 1$

$f(n) = O(n^2)$

Space Complexity

A
B
C
$n^2 + O(1)$

$\Rightarrow O(n^2)$

```
Add(A , B, n)
{
    for ( i =0; i < n; i++)
    {
        for ( j=0; j < n; j++)
        {
            c[i][j]=0;
            for ( k=0; k < n; k++)
            {
                c[i][j]= A[i][j] +B[i][j];
            }
        }
    }
}
```

$(n+1)$

$n \times (n+1)$

$n \times n$

$n \times n \times (n+1)$

$n \times n \times n$

$f(n) = n+1 + n^2 + n + n^2 + n^3 + n^2 + n^3$

$f(n) = 2n^3 + 3n^2 + 2n + 1$

Time Complexity $= O(n^3)$

```
for ( i =0; i*i< n; i++)
{
    printf ("%d", i);
}
```

$i \times i = n$

$i^2 = n$

$i = \sqrt{n}$

$f(n) = O(\sqrt{n})$

```
for ( i =0; i < n; i++)
{
        for ( j=1; j < n; j=j*2)
        {
                printf("%d%d",i,j);
        }
}
```

$n \times \log n$

$n \times (\log n - 1)$

$f(n) = (n+1) + n\log n + n\log n$

$= n\log n + n + 1$

$f(n) = O(n\log n)$

AKTU

B.Tech 5th Sem

GATEWAY CLASSES

CS IT & CS Allied

DAA : Design & Analysis Of Algorithm

UNIT-1: Introduction

Lecture-3

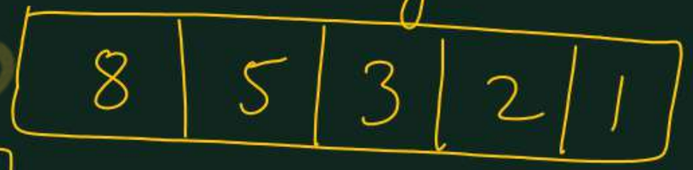
Today's Target

- Asymptotic Notations
- Introduction to Recurrence Relations:
  Substitution method

By Dr. Nidhi Parashar Ma'am
- M.Tech Gold Medalist
- Net Qualified

Complexity Analysis

Array



1) Best case →

2) Worst case →

3) Average case →

Linear Search

$A = \{ 8, 3, 2, 5, 6 \}$

$x = 8$

$x = 4$ ⟶ n

Best case — $x = 8$ — $f(n) = O(1)$

Worst case — $x = 4$ — $f(n) = O(n)$

Average case — $f(n) = O(n)$

$A = \{ \_\_ \; \_\_ \; \_\_ \; \_\_ \; \ldots \; n \}$

$$f(n) = \frac{1 + 2 + 3 \ldots n}{n} = \frac{n(n+1)}{2 \times n} = O(n)$$

Asymptotic Notations

- These notations are mathematical tools to represent the complexities.
- **Asymptotic notations** are used to write fastest and slowest possible running time for an algorithm. These are also referred to as 'best case' and 'worst case' scenarios respectively.
- In asymptotic notations, we derive the complexity concerning the size of the input.

There are mainly three asymptotic notations:

1. Big-Oh Notation (O-notation)
2. Omega Notation (Ω-notation)
3. Theta Notation (Θ-notation)

4. Small-oh notation (o)
5. Small omega (ω)

Big-Oh Notation (O-notation)

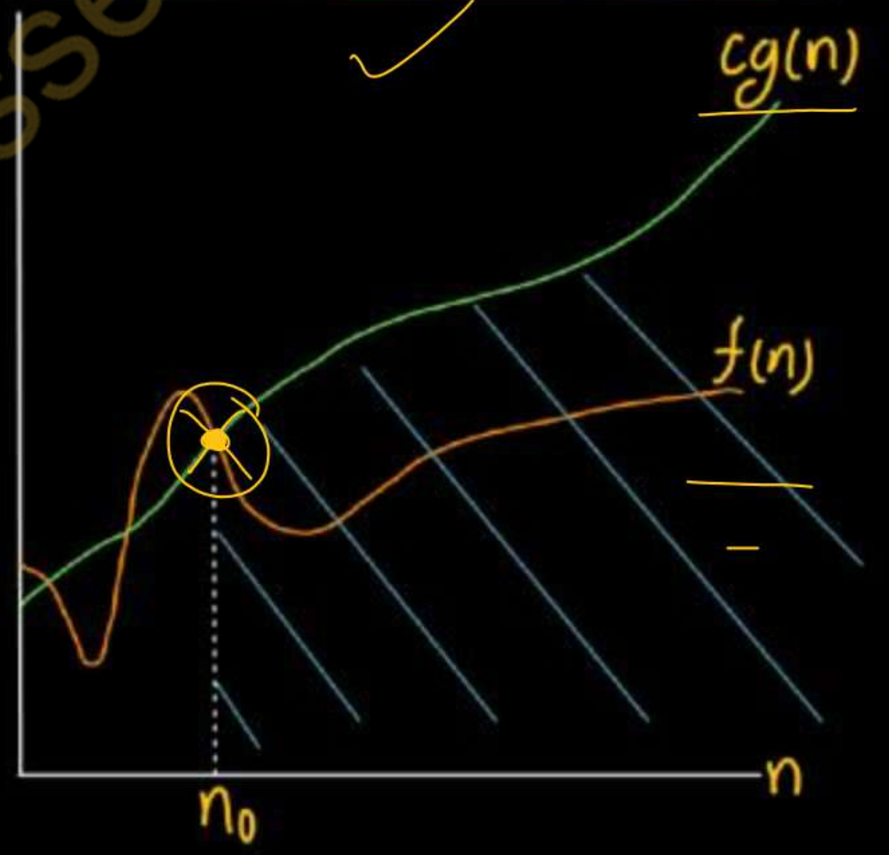
- Upper bound of the running time of an algorithm.
- It gives the worst-case complexity of an algorithm.
- Maximum time required by an algorithm

tight upper bound

If f(n) describes the running time of an algorithm, f(n) is O(g(n)) if there exist a positive constants c and n0 such that, $0 \leq f(n) \leq c*g(n)$ for all $n \geq n0$

$f(n) = O(g(n))$

$\leq c * g(n)$



$f(n) = 3n + 2$ and $g(n) = n$

Prove that $f(n) = O(g(n))$ ⟶ $f(n) \leq C * g(n)$

$f(n) = 3n + 2$

n	$f(n)$
$n=1$	5
$n=2$	8
$n=3$	11
$n=4$	14
$n=5$	17

$g(n) = n$ $\quad c = 4$

C, n_0

$1 \times 4 = 4$

$2 \times 4 = 8$

$3 \times 4 = 12$

$4 \times 4 = 16$

$5 \times 4 = 20$

Let take $c = 4$

$n_0 \geq 2$

$3n + 2 \leq 4n$

$3n^2+2 \in 4n$

$\in 5n$

$\in 6n$

$\in 6n^2$

$\in 6n^2 \log n$

$\in 2^n$

$f(n) = n^2$ and $g(n) = 2^n$

Prove that $f(n) = O(g(n))$

$\Rightarrow f(n) \leq c * g(n)$

$n^2 \leq c * 2^n$

	$f(n) = n^2$	$g(n) = 2^n$
$n=1$	1	2
$n=2$	4	4
$n=3$	9	8
$n=4$	16	16
$n=5$	25	32
$n=6$	36	64
$n=7$		
$n=8$		

$n_0 = 4$

$c = 1$

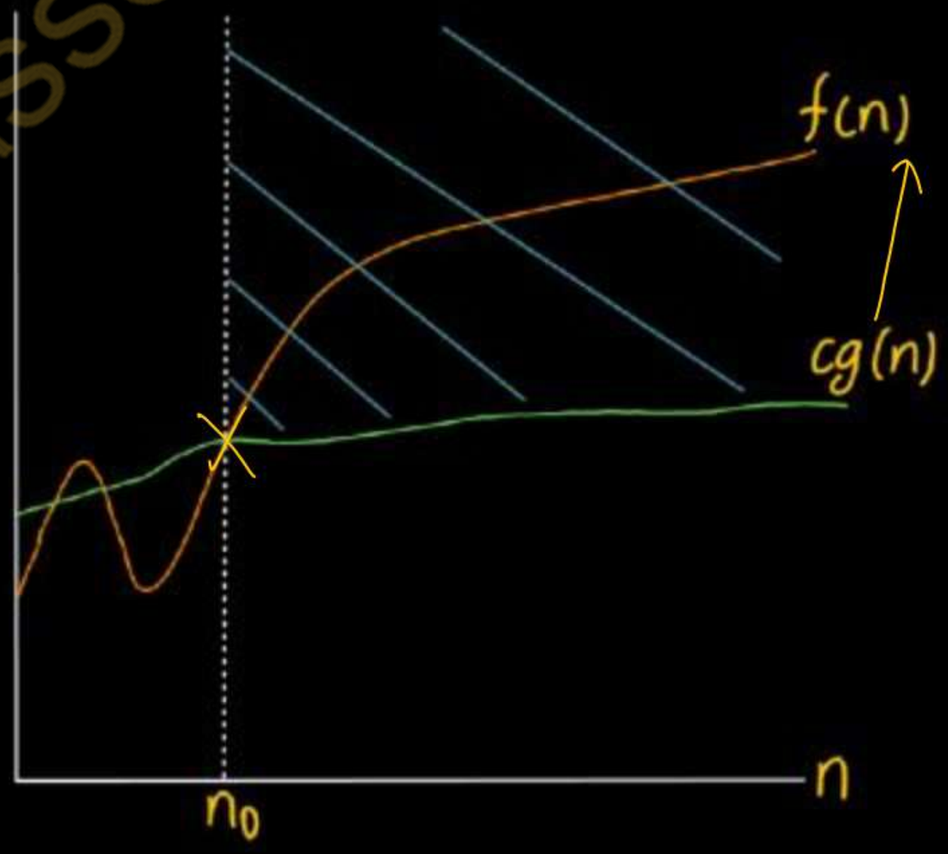
$n^2 \leq 2^n$ for $n \geq 4$

Big-Omega Notation (Ω-Notation)

- Lower bound of the running time of an algorithm.
- It gives the best-case complexity of an algorithm.
- Minimum time required by an algorithm

tight lower Bound

If f(n) describes the running time of an algorithm, f(n) is O(g(n)) if there exist a positive constants c and n0 such that, $c*g(n) \leq f(n)$ for all $n \geq n0$



$f(n)= n$ and $g(n)= 3n+2$

Prove that $f(n) = \Omega(g(n))$

$f(n) = n \quad g(n) = 3n+2$

$f(n) = \Omega(g(n))$ for all $n \geq 1$ and

$n \geq C * (3n+2)$ $\quad C = \frac{1}{10}$

Lets take $C = \frac{1}{10}$

$n \geq \frac{1}{10} * (3n+2)$

$10n \geq 3n+2$

gives the loose
↑ Upper Bound

④ Small-oh (o)

$f(n) = o(g(n))$

$f(n) < c * g(n)$

⑤ Small-omega (ω)

↓ loose lower bound

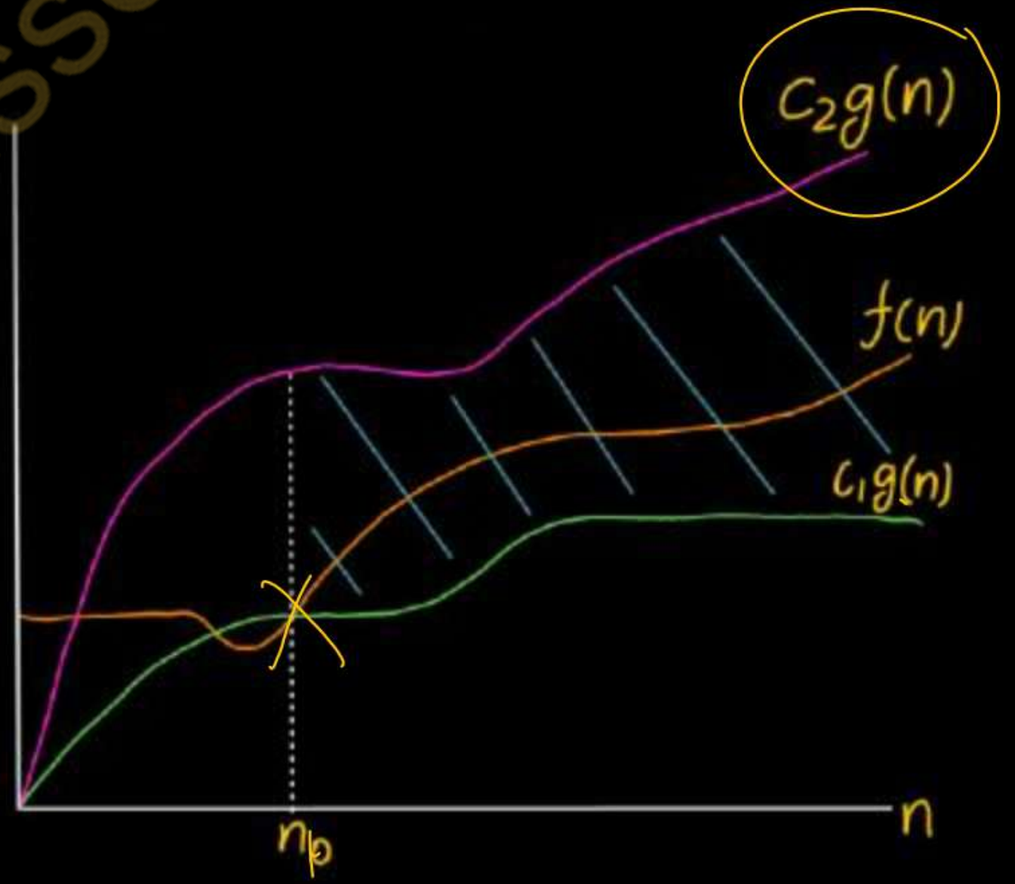
$f(n) = \omega(g(n))$

$f(n) > c * g(n)$

Theta Notation (Θ-Notation)

- Both Lower bound and Upper bound (Exact Bound) of the running time of an algorithm.
- It gives the average-case complexity of an algorithm.
- Average time required by an algorithm

If f(n) describes the running time of an algorithm, f(n) is Θ(g(n)) if there exist a positive constants c and n0 such that, $c1*g(n) \leq f(n) \leq c2*g(n)$ for all $n \geq n0$



$f(n) = 3n+2$ and $g(n) = n$

Prove that $f(n) = \Theta(g(n))$

$f(n) = 3n+2$

$1 * n \leq 3n+2 \leq 5n$

$c_1 = 1$

$c_2 = 5$

for all $n \geq 1$

AKTU

B.Tech 5th Sem

GATEWAY CLASSES

CS IT & CS Allied

DAA : Design & Analysis Of Algorithm

UNIT-1: Introduction

Lecture-4

Today's Target

- Introduction to Recurrence Relations:
- Substitution method

By Dr. Nidhi Parashar Ma'am

- M.Tech Gold Medalist
- Net Qualified

Recurrence Relations

- A recurrence is an equation that describes a function in terms of its value on smaller inputs.
- Recurrences are generally used to model the time complexity of recursive algorithms.

Example 1:
recursive definition for the factorial function: $n!=(n-1)!* n$ for $n>1$; $1!=0$ and $0!=1$

n-2 (n-1) (n)

1, 1, 2 3 5 8 . . . . .

Example 2:
Fibonacci Sequence: $Fib(n)=Fib(n-1)+Fib(n-2)$ for $n>2$; $Fib(1)=Fib(2)=1$.

Recurrence Relations

- Decreasing Function:

$$T(n) = a \times T(n-b) + f(n)$$

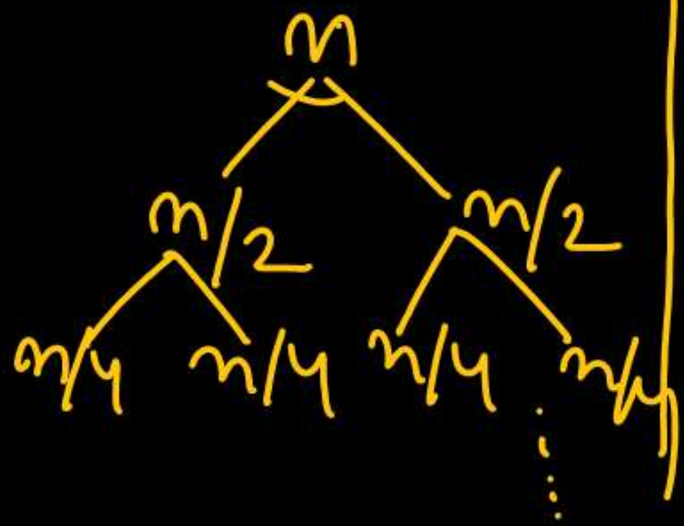
- $T(n)$: cost of solving the main problem
- a: No. of subproblems
- $T(n-b)$: cost of solving smaller subproblem
- $f(n)$: additional cost for reducing the problem size

- Dividing Functions

$$T(n) = aT(n/b) + D(n) + C(n)$$

- $T(n)$: Total cost
- a: No. of subproblems
- $T(n/b)$: cost of solving smaller subproblems
- $D(n)$: cost of divide step
- $C(n)$: cost of combining step
- $D(n) + C(n)$: $f(n)$

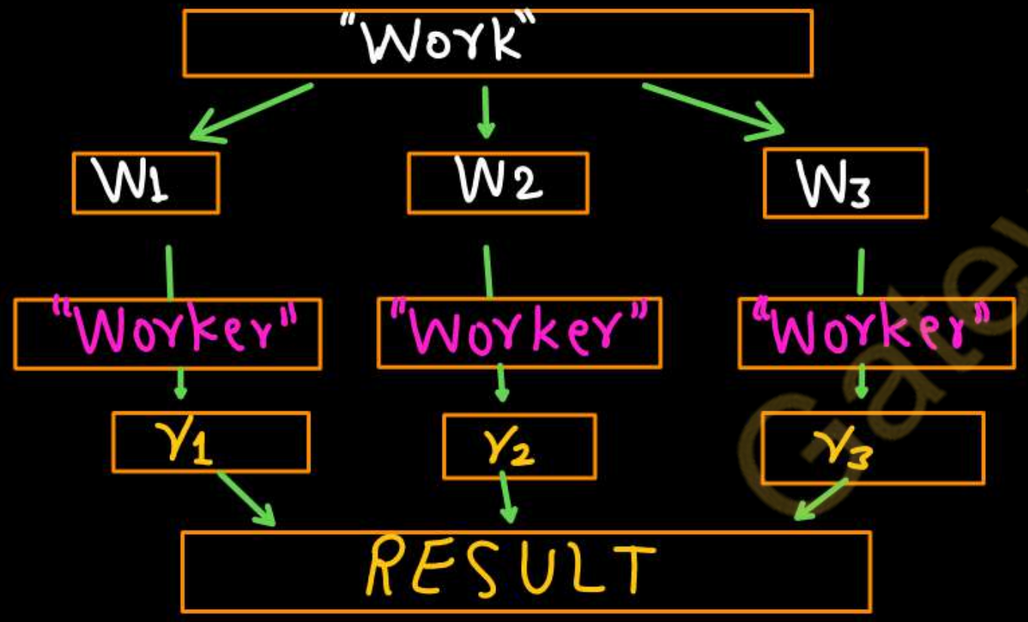
$a = 2$, $b = 2$

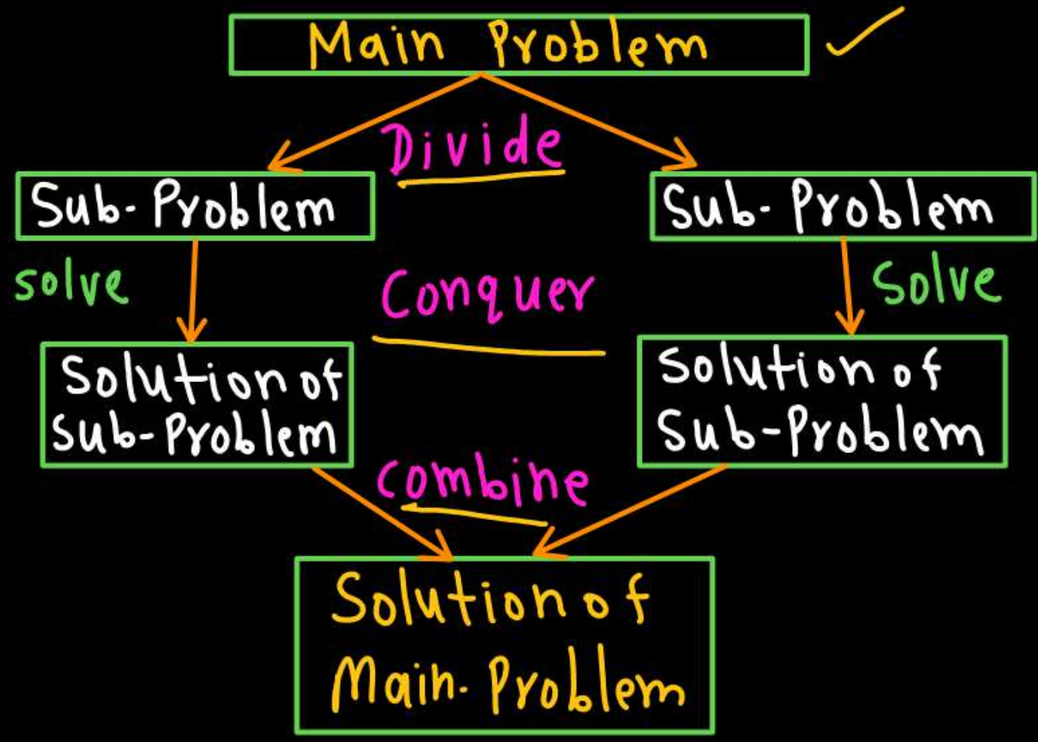


Divide and Conquer approach

we solve the problem recursively, applying three basic steps at each level of the recursion:

- Divide the problem into a number of subproblems that are smaller instances of the same problem
- Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough (bottom out)
- Combine the solutions to the subproblems into the solution for the original problem.

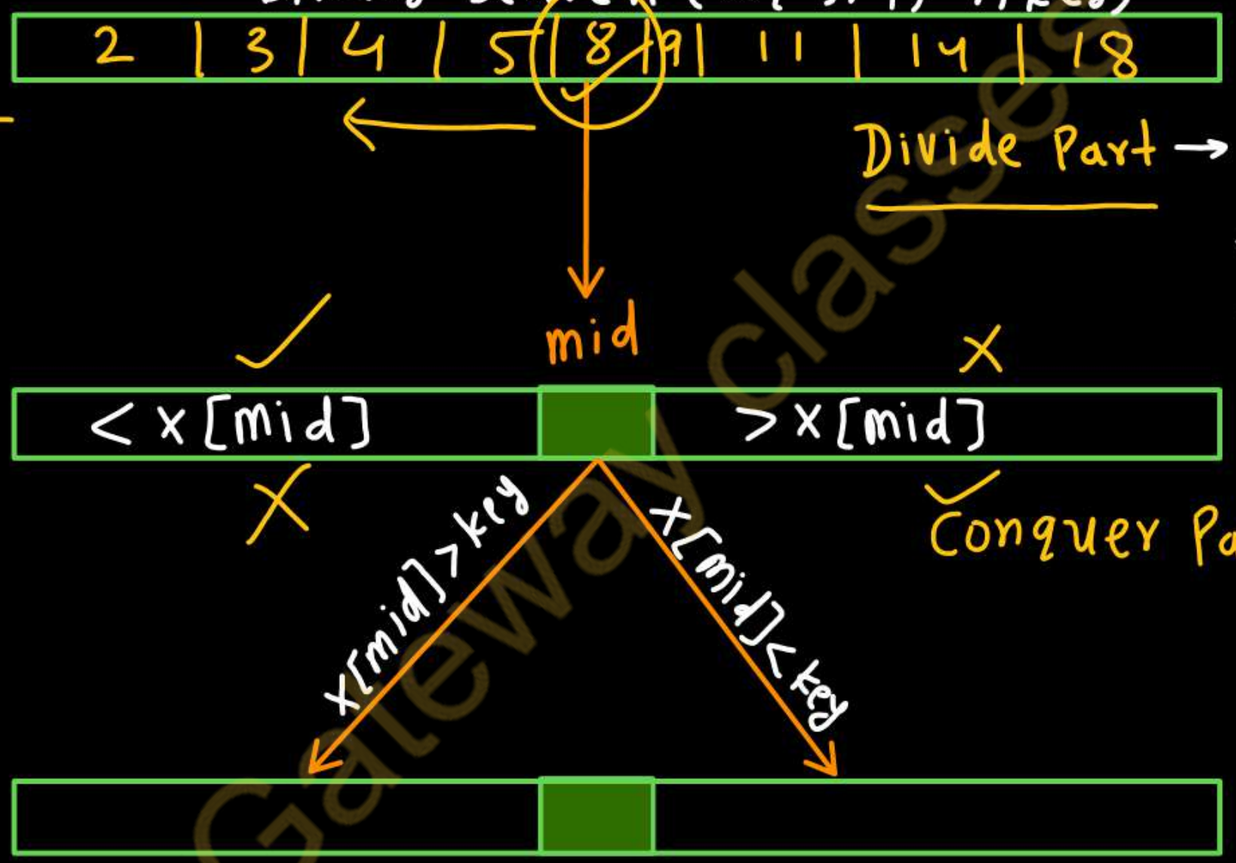




Divide and Conquer approach

Binary Search

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



already sorted array

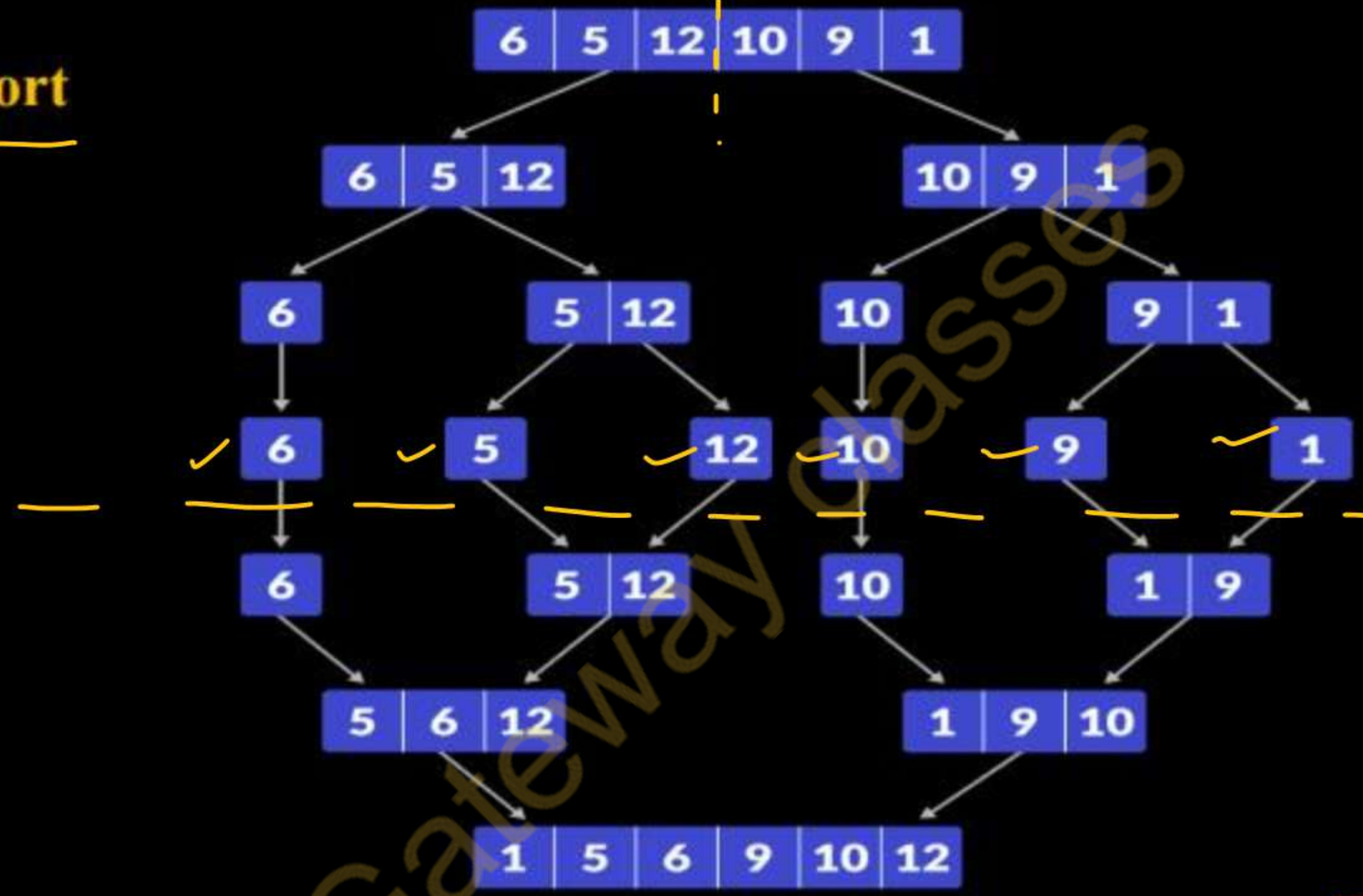
4

Divide Part → Dividing the array into two equal halves using mid index

Conquer Part → Based on the comparision with X[mid] and key, we recursively search the key in only one half

Divide and Conquer approach

Merge Sort



$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1, \end{cases}$$

Recurrence Relations

- Recurrence relations are often used to model the cost of recursive functions. For example, the number of multiplications required by a recursive version of the factorial function for an input of size n will be 0, when n=1 and n=0.
- It can be represented using the following recurrence relation.

$$T(n)= T(n-1)+1 \text{ for } n>1; \quad T(0)=T(1)=0$$

Methods for solving Recurrences:

- Substitution Method
- Recursion tree method
- Master's Method

Substitution Method

General steps :

- Substitute the input size to obtain a sequence of terms.
- Identify a pattern in the sequence of terms and simplify the recurrence relation to obtain a closed-form expression for the number of operations performed by the algorithm.
- Determine the order of growth.

$T(n) = T(n-1) + n$ for $n>0$ and $T(n)=1$ for $n=0$

$T(n) = T(n-1) + n$ ——— (1)

$T(n-1) = T(n-2) + (n-1)$

$T(n) = T(n-2) + (n-1) + n$ ——— (2)

$T(n) = T(n-3) + (n-2) + (n-1) + n$ ——— (3)

(K times)

$T(n) = T(n-k) + (n-(k-1)) + (n-(k-2)) \ldots\ldots (n-1) + n$

$T(n) = T(n-1) + n$

$T(n-1) = T(n-2) + n-1$

$T(n) = T(n-2) + (n-1) + n$

$T(n-2) = T(n-3) + (n-2)$

$$T(n) = T(n-K) + (n-(K-1)) + (n-(K-2)) + \ldots\ldots n$$

$T(n) = 1$ for $n = 0$

put $(n-K) = 0$

$K = n$

$$T(n) = T(0) + (n-n+1) + (n-n+2) \ldots\ldots n$$

$$T(n) = 1 + [1+2+3+\ldots\ldots n]$$

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

$$T(n) = O(n^2)$$

Substitution Method

$T(n) = 2T(n-1) + 1$ for $n>0$ and $T(n)=1$ for $n=0$

$T(n) = 2T(n-1) + 1$ ——— ①

$T(n-1) = 2T(n-2) + 1$

$T(n) = 2[2T(n-2) + 1] + 1$

$= 2^2T(n-2) + 2 + 1$ ——— ②

$T(n) = 2^3 T(n-3) + 2^2 + 2 + 1$ ——— ③

⋮ K times

$T(n) = 2^k T(n-k) + 2^{k-1} + 2^{k-2} + \ldots\ldots + 2 + 1$

$T(n) = 2T(n-1) + 1$ ——— ①

$T(n-1) = 2T(n-2) + 1$

$T(n) = 2[2T(n-2)+1] + 1$

$T(n) = 2^2 T(n-2) + 2 + 1$ ——— ②

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

$T(n) = 2^2[2T(n-3)+1] + 2 + 1$

$= 2^3 T(n-3) + 2^2 + 2 + 1$

$$T(n) = 2^K T(n-K) + 2^{K-1} + 2^{K-2} + \cdots + 2 + 1$$

put $n-K=0$

$$K = n$$

$$T(n) = 2^n T(0) + 2^{n-1} + 2^{n-2} + \cdots + 2 + 1$$

$$2^n + 2^n - 1$$

$$T(n) = O(2^n)$$

G.P. Series

$1 + ar + ar^2 + ar^3 \cdots ar^{n-1}$

$$S = a \times \left[\frac{r^n - 1}{r-1}\right]$$

$1 + 2 + \cdots + 2^{n-2} + 2^{n-1}$

$a = 1, \; r = 2$

$$1 \times \left[\frac{2^n - 1}{2-1}\right]$$

Substitution Method

$T(n) = 2T(n/2) + n$ for $n>1$ and $T(n)=1$ for $n=1$

$T(n) = 2T(n/2) + n$ ——— (1)

$T(n/2) = 2T(n/2^2) + n/2$

$T(n) = 2[2T(n/2^2) + n/2] + n$

$T(n) = 2^2T(n/2^2) + n + n$ ——— (2)

$T(n) = 2^3T(n/2^3) + n+n+n$ ——— (3)

k times

$T(n) = 2^kT(n/2^k) + kn$

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

$T(n/2) = 2T(n/2^2) + n/2$

$T(n) = 2\left[2T(n/2^2) + n/2\right] + n$

$T(n) = 2^2T(n/2^2) + n + n$

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

$T(n) = 2^2\left[2T(n/2^3) + n/2^2\right] + n + n$

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

$T(n) = 1$ $T(n) = 2^K T(n/2^K) + Kn$
$n = 1$

Given $T(1) = 1$

put $n/2^K = 1$

$2^K = n$

$K = \log_2 n$

$T(n) = n\,T(1) + \log_2 n \times n$

$= n + n \log_2 n$

$T(n) = n \log_2 n$

AKTU

B.Tech 5th Sem

GATEWAY CLASSES

CS IT & CS Allied

DAA : Design & Analysis Of Algorithm

UNIT-1: Introduction

Lecture-5

Today's Target

Master's Method For Solving Recurrence Relations

By Dr. Nidhi Parashar Ma'am

- M.Tech Gold Medalist
- Net Qualified

Master's Method

It can only be applied on decreasing and dividing recurring functions.

Master's Theorem for Dividing Functions

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

$f(n) = n^k \log^p n$

- Here, a >= 1 and b > 1,
- n – size of the problem
- a – number of sub-problems
- n/b– size of the sub problems assuming that all sub-problems are of same size.
- f(n) – cost of work done outside the recursion

$f(n) = \Theta(n^k \log^p n)$,where k >= 0 and p is a real number;

If $a > b^k$, then $T(n) = \Theta(n^{\log_b a})$

If $a = b^k$

- If $p > -1$, then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$
- If $p = -1$, then $T(n) = \Theta(n^{\log_b a} \log \log n)$
- If $p < -1$, then $T(n) = \Theta(n^{\log_b a})$

If $a < b^k$,

- If $p >= 0$, then $T(n) = \Theta(n^k \log^p n)$.
- If $p < 0$, then $T(n) = \Theta(n^k)$

[$\log_b a = \log a / \log b$.]

Master's Method

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

$a = 4$, $b = 2$, $K = 1$, $P = 0$

$f(n) = n^k \log^p n$

$f(n) = n^1 \log^{p=0} n$

a	b^k
4	$2^1 = 2$

$a > b^k$ ⟹ case - I

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

$$\Theta(n^{\log_2 4}) = \Theta(n^2)$$

(I) $f(n) = \Theta(n^k \log^p n)$, where $k >= 0$ and p is a real number;

If $a > b^k$, then $T(n) = \Theta(n^{\log_b a})$

(II) If $a = b^k$

- If $p > -1$, then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$
- If $p = -1$, then $T(n) = \Theta(n^{\log_b a} \log \log n)$
- If $p < -1$, then $T(n) = \Theta(n^{\log_b a})$

(III) If $a < b^k$,

- If $p >= 0$, then $T(n) = \Theta(n^k \log^p n)$.
- If $p < 0$, then $T(n) = \Theta(n^k)$

[$\log_b a = \log a / \log b$.]

Master's Method

$T(n) = 2T(n/2) + n \rightarrow n^1 \log^0 n$

$a = 2$

$b = 2$

$p = 0$

$k = 1$

$\frac{a}{2} \quad \frac{b^k}{2^1 = 2}$

$2 = 2 \Rightarrow$ Case – II

$T(n) = \Theta(n^{\log_b a} \cdot \log^{p+1} n)$

$= \Theta(n^{\log_2 2} \log^1 n)$

$\boxed{T(n) = \Theta(n \log n)}$

$f(n) = \Theta(n^k \log^p n)$,where $k >= 0$ and p is a real number;

If $a > b^k$, then $T(n) = \Theta(n^{\log_b a})$

If $a = b^k$

- If $p > -1$, then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$
- If $p = -1$, then $T(n) = \Theta(n^{\log_b a} \log \log n)$
- If $p < -1$, then $T(n) = \Theta(n^{\log_b a})$

If $a < b^k$,

- If $p >= 0$, then $T(n) = \Theta(n^k \log^p n)$.
- If $p < 0$, then $T(n) = \Theta(n^k)$

[$\log_b a = \log a / \log b$.]

Master's Method

$T(n) = 2T(n/2) + n/\log n$

$a = 2$, $b = 2$, $K = 1$, $P = -1$

$f(n) = n/\log n$

$n^1 \cdot \log^{-1} n$

$\frac{a}{2} \quad \frac{b^K}{2^1 = 2}$

$a = b^K \Rightarrow$ case-II

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

$= \Theta(n^{\log_2 2} \log\log n)$

$\boxed{T(n) = \Theta(n \log\log n)}$

$f(n) = \Theta(n^k \log^p n)$, where $k >= 0$ and p is a real number;

If $a > b^k$, then $T(n) = \Theta(n^{\log_b a})$

If $a = b^k$

- If $p > -1$, then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$
- If $p = -1$, then $T(n) = \Theta(n^{\log_b a} \log\log n)$
- If $p < -1$, then $T(n) = \Theta(n^{\log_b a})$

If $a < b^k$,

- If $p >= 0$, then $T(n) = \Theta(n^k \log^p n)$.
- If $p < 0$, then $T(n) = \Theta(n^k)$

[$\log_b a = \log a / \log b$.]

Master's Method

$a = 1$, b

$T(n) = T(n/2) + n^2$

$f(n) = n^2 \log^0 n$

$a = 1$
$b = 2$
$k = 2$
$p = 0$

a	b^k
1	$2^2 = 4$

$a < b^k \Rightarrow$ case – III

$T(n) = \Theta(n^k \log^p n)$

$= \Theta(n^2 \log^0 n)$

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

$f(n) = \Theta(n^k \log^p n)$,where $k >= 0$ and p is a real number;

(I) If $a > b^k$, then $T(n) = \Theta(n^{\log_b a})$

(II) If $a = b^k$
- If $p > -1$, then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$
- If $p = -1$, then $T(n) = \Theta(n^{\log_b a} \log \log n)$
- If $p < -1$, then $T(n) = \Theta(n^{\log_b a})$

(III) If $a < b^k$,
- If $p >= 0$, then $T(n) = \Theta(n^k \log^p n)$.
- If $p < 0$, then $T(n) = \Theta(n^k)$

[$\log_b a = \log a / \log b$.]

Master's Method

$T(n) = T(n/2) + n^2 \log n$

$a = 1$, $b = 2$, $k = 2$, $p = 1$

$k = 2$, $p = 1$

a	b^k
1	$2^2 = 4$

$a < b^k \Rightarrow$ case – III

$T(n) = \Theta(n^k \log^p n)$

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

$f(n) = \Theta(n^k \log^p n)$, where $k >= 0$ and p is a real number;

If $a > b^k$, then $T(n) = \Theta(n^{\log_b a})$

If $a = b^k$

- If $p > -1$, then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$
- If $p = -1$, then $T(n) = \Theta(n^{\log_b a} \log \log n)$
- If $p < -1$, then $T(n) = \Theta(n^{\log_b a})$

If $a < b^k$,

- If $p >= 0$, then $T(n) = \Theta(n^k \log^p n)$.
- If $p < 0$, then $T(n) = \Theta(n^k)$

[$\log_b a = \log a / \log b$.]

Master's Method

Master's Theorem for Decreasing Functions

$T(n) = a\ T(n-b) + f(n)$
where, $a >= 1$ and $b > 1$, and
$f(n)$ is asymptotically positive

- Here, $a >= 1$ and $b > 1$,
- n – size of the problem
- a – number of sub-problems
- n-b– size of the sub problems assuming that all sub-problems are of same size.
- f(n) – cost of work done outside the recursion

$f(n) = \Theta(n^k)$,where $k >= 0$

- **If $a = 1$, $T(n) = O(n^{k+1})$**
- **if $a > 1$, $T(n) = O(a^{n/b} * n^k)$**
- **if $a < 1$, $T(n) = O(n^k)$**

Master's Method

$T(n) = T(n-1) + 1$ ($a = 1$, $b = 1$)

$f(n) = 1 \Rightarrow n^0$

$a = 1$, $K = 0$ $\Rightarrow$ case-I

$T(n) = O(n)$

$T(n) = T(n-1) + \log n$ ($a = 1$)

$T(n) = O(n \times \log n)$

$T(n) = T(n-1) + n$ ($a = 1$)

$a = 1$, $K = 1$

$T(n) = O(n^2)$

$= O(n^2)$

$f(n) = O(n^K)$

(I) If $a = 1$, $T(n) = O(n^{k+1})$
$= O(n \times f(n))$

(II) if $a > 1$, $T(n) = O(a^{n/b} * n^k)$

(III) if $a < 1$, $T(n) = O(n^k)$

Master's Method

- If $a = 1$, $T(n) = O(n^{k+1})$
- if $a > 1$, $T(n) = O(a^{n/b} * n^k)$
- if $a < 1$, $T(n) = O(n^k)$

$T(n) = 2T(n-1) + 1$

$f(n) = 1 = n^0$ (k = 0)

$a = 2$
$b = 1$
$k = 0$

$T(n) = O(a^{n/b} \times n^k)$

$T(n) = O(2^{n/1} \times n^0)$

$= O(2^n)$

$T(n) = 3T(n-1) + 1$

$a = 3$
$b = 1$
$k = 0$

$T(n) = O(a^{n/b} \times n^k)$

$O(3^{n/1} \times 1)$

$T(n) = O(3^n)$

$T(n) = 2T(n-1) + n$

$a = 2$
$b = 1$
$k = 1$

$T(n) = O(2^n \times n^1)$

$T(n) = O(n \cdot 2^n)$

Master's Method

Test For masters theorem application

- $T(n) = 2^n T(n/2) + n$ ✗
- $T(n) = 64T(n/2) - n^2$ ✗
- $T(n) = T(n/2) + \sin n$ ✗
- $T(n) = 2T(n/2) + 1/n$ ✗ → $f(n) = n^{-1}$
- $T(n) = 3.2T(n/2) + n$ ✗
- $T(n) = T(\sqrt{n}) + 1$ for $n>2$ and 1 for $n=2$ ✓

$f(n) = \Theta(n^k \log^p n)$, where $k >= 0$ and p is a real number;

If $a > b^k$, then $T(n) = \Theta(n^{\log_b a})$

If $a = b^k$

- If $p > -1$, then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$
- If $p = -1$, then $T(n) = \Theta(n^{\log_b a} \log \log n)$
- If $p < -1$, then $T(n) = \Theta(n^{\log_b a})$

If $a < b^k$,

- If $p >= 0$, then $T(n) = \Theta(n^k \log^p n)$.
- If $p < 0$, then $T(n) = \Theta(n^k)$

Master's Method

Test For masters theorem application

$T(n) = T(\sqrt{n}) + 1$ for $n>2$ and
1 for $n=2$

$\sqrt{n} = n^{1/2}$

Let $n = 2^m$

$T(2^m) = T(2^{m/2}) + 1$

$S(m) = S(m/2) + 1$ → $n^0 \log^0 n$

$T(2^m) \Rightarrow S(m)$

Now apply masters method.

$a=1, b=2, k=0, p=0$

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

$a = b^k \Rightarrow$ case-II

Case-II will be applied

$S(m) = O(\log m)$

As $m=\log n$

$=O(\log \log n)$

$O(\log \log n)$

$f(n) = \Theta(n^k \log^p n)$, where $k >= 0$ and p is a real number;

If $a > b^k$, then $T(n) = \Theta(n^{\log_b a})$

If $a = b^k$

- If $p > -1$, then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$
- If $p = -1$, then $T(n) = \Theta(n^{\log_b a} \log \log n)$
- If $p < -1$, then $T(n) = \Theta(n^{\log_b a})$

If $a < b^k$,

- If $p >= 0$, then $T(n) = \Theta(n^k \log^p n)$.
- If $p < 0$, then $T(n) = \Theta(n^k)$

Master's Method

Test For masters theorem application

$T(n) = 2T(\sqrt{n}) + \log n$

Let $n = 2^m$

$T(2^m) = 2T(2^{m/2}) + m$

$S(m) = 2S(m/2) + m$

$\log_2^2 m$

$m \log_2^2 1$

$\Rightarrow m$

Now apply masters method.

a=2, b=2, k=1, p=0

Case-II will be applied

$O(m^{\log_2^2} \log^{0+1} m)$

$\Rightarrow O(m \log m)$

S(m) = O(m log m))

As m=log n

=O(logn log log n)

$O(\log n \log \log n)$

$f(n) = \Theta(n^k \log^p n)$,where k >= 0 and p is a real number;

If $a > b^k$, then $T(n) = \Theta(n^{\log_b a})$

If $a = b^k$

- If $p > -1$, then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$
- If $p = -1$, then $T(n) = \Theta(n^{\log_b a} \log \log n)$
- If $p < -1$, then $T(n) = \Theta(n^{\log_b a})$

If $a < b^k$,

- If $p >= 0$, then $T(n) = \Theta(n^k \log^p n)$.
- If $p < 0$, then $T(n) = \Theta(n^k)$

AKTU

B.Tech 5th Sem

GW GATEWAY CLASSES

CS IT & CS Allied

DAA : Design & Analysis Of Algorithm

UNIT-1: Introduction

Lecture-6

Today's Target

- Recursion Tree method Solving Recurrence Relations
- Mixed Problems
- AKTY PYQs

By Dr. Nidhi Parashar Ma'am
- M.Tech Gold Medalist
- Net Qualified

Recursion Tree Method

- A recursion tree is a tree where each node represents the cost of a certain recursive sub-problem.
- We sum up the values in each node to get the cost of the entire algorithm..

Steps to Solve Recurrence Relations Using Recursion Tree Method-

Step-01: Draw a recursion tree based on the given recurrence relation.

Step-02: Determine-

- Cost of each level
- Total number of levels in the recursion tree

Step-03: Add cost of all the levels of the recursion tree and simplify the expression so obtained in terms of asymptotic notation

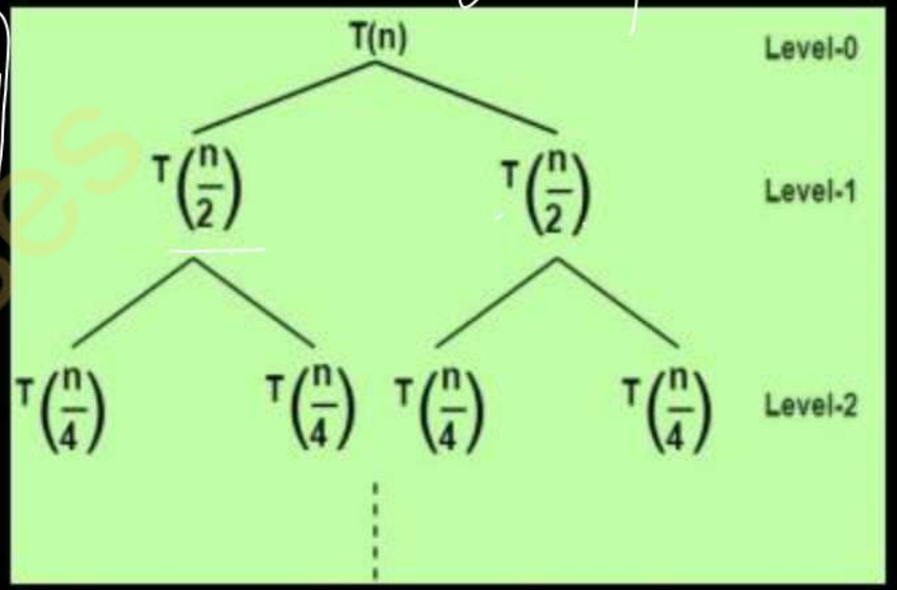
Recursion Tree Method

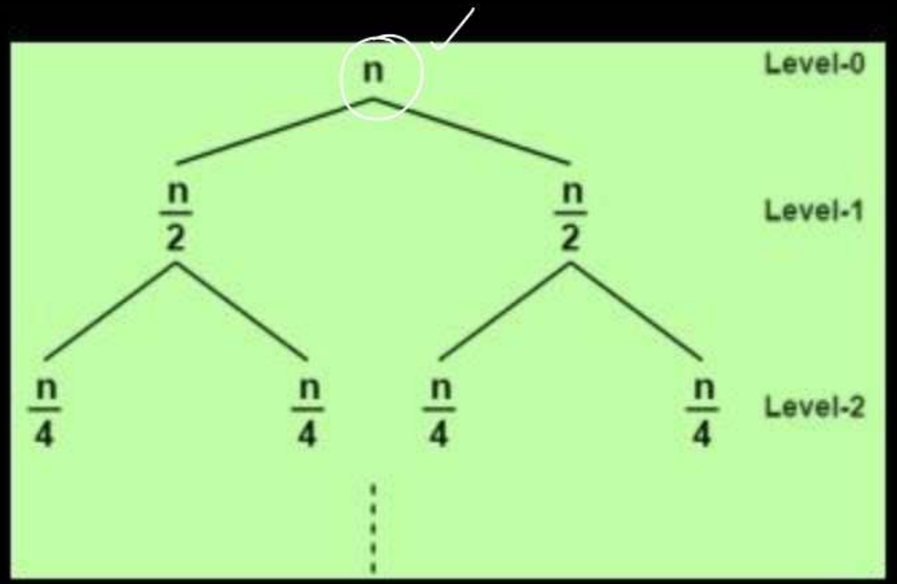
size of subproblem

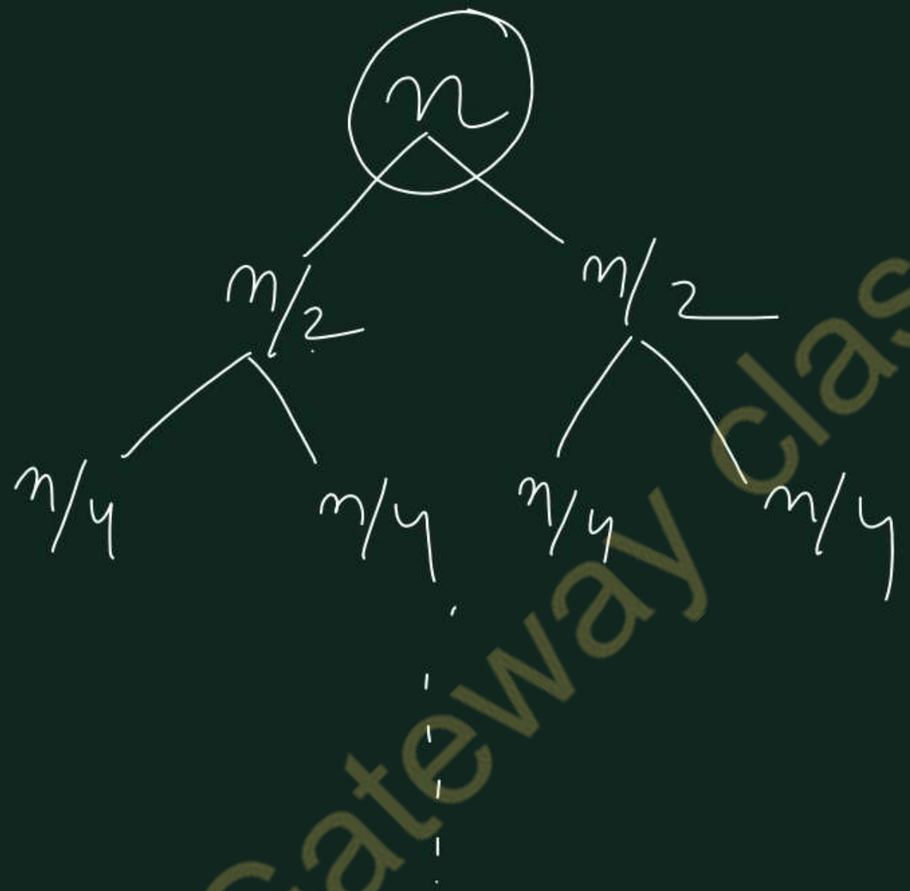
$$T(n) = 2T(n/2) + n \quad \text{for } n > 1$$
$$= 1 \quad \text{for } n = 1$$

The given recurrence relation shows-

- A problem of size n will get divided into 2 sub-problems of size n/2.
- Then, each sub-problem of size n/2 will get divided into 2 sub-problems of size n/4 and so on.
- At the bottom most layer, the size of sub-problems will reduce to 1.
- The cost of dividing a problem of size n into two subproblems of size n/2 and then combining their solution is n.



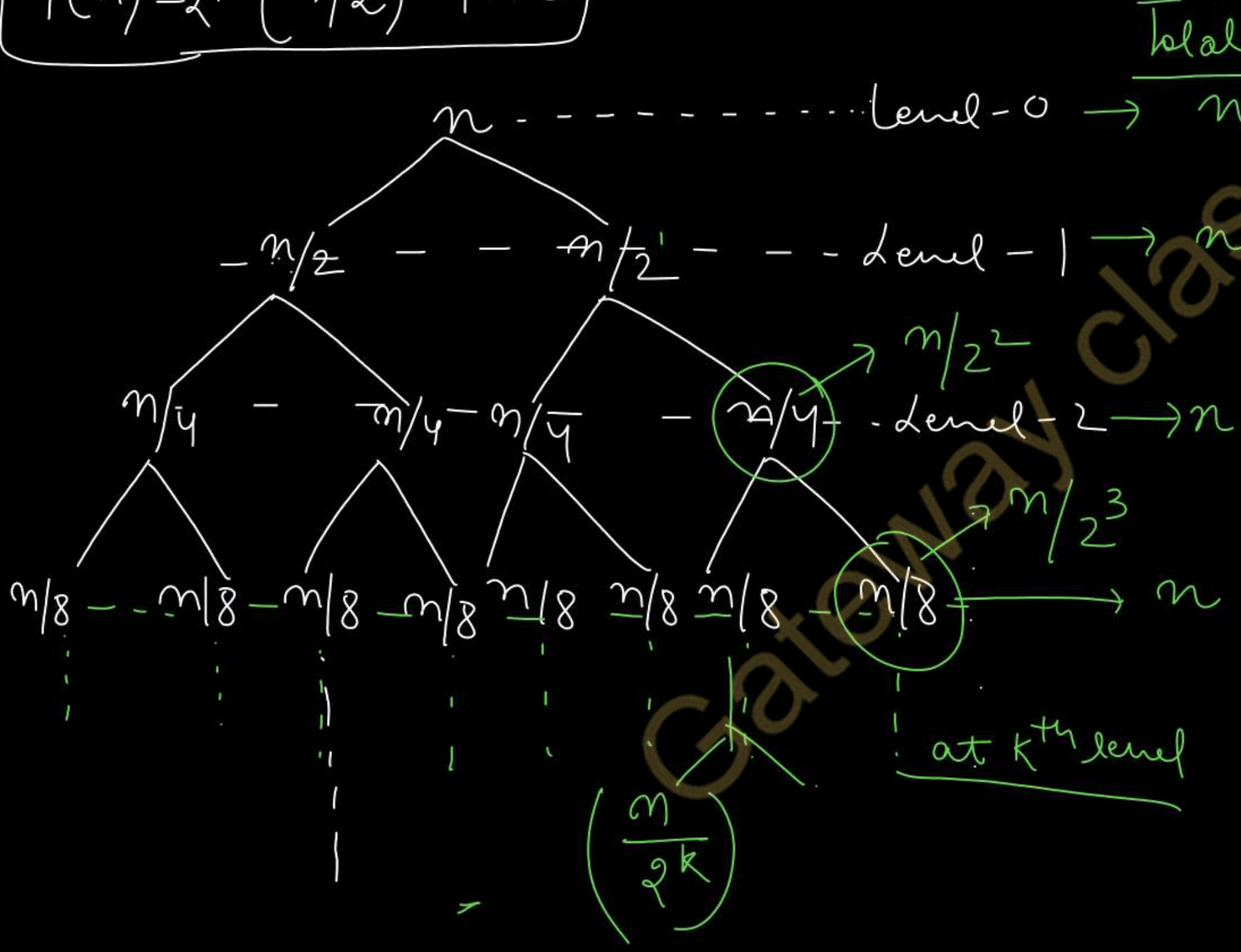




Q.1 Solve the recurrence $T(n) = 2T(n/2) + n$ using recursion tree method.

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

↳ Root of Tree



Let $\frac{n}{2^k} = 1$ for base condition

$n = 2^k$

$K = \log_2 n$

Total No. of levels of the tree $= (K+1)$

Total cost $= n \times (K+1)$

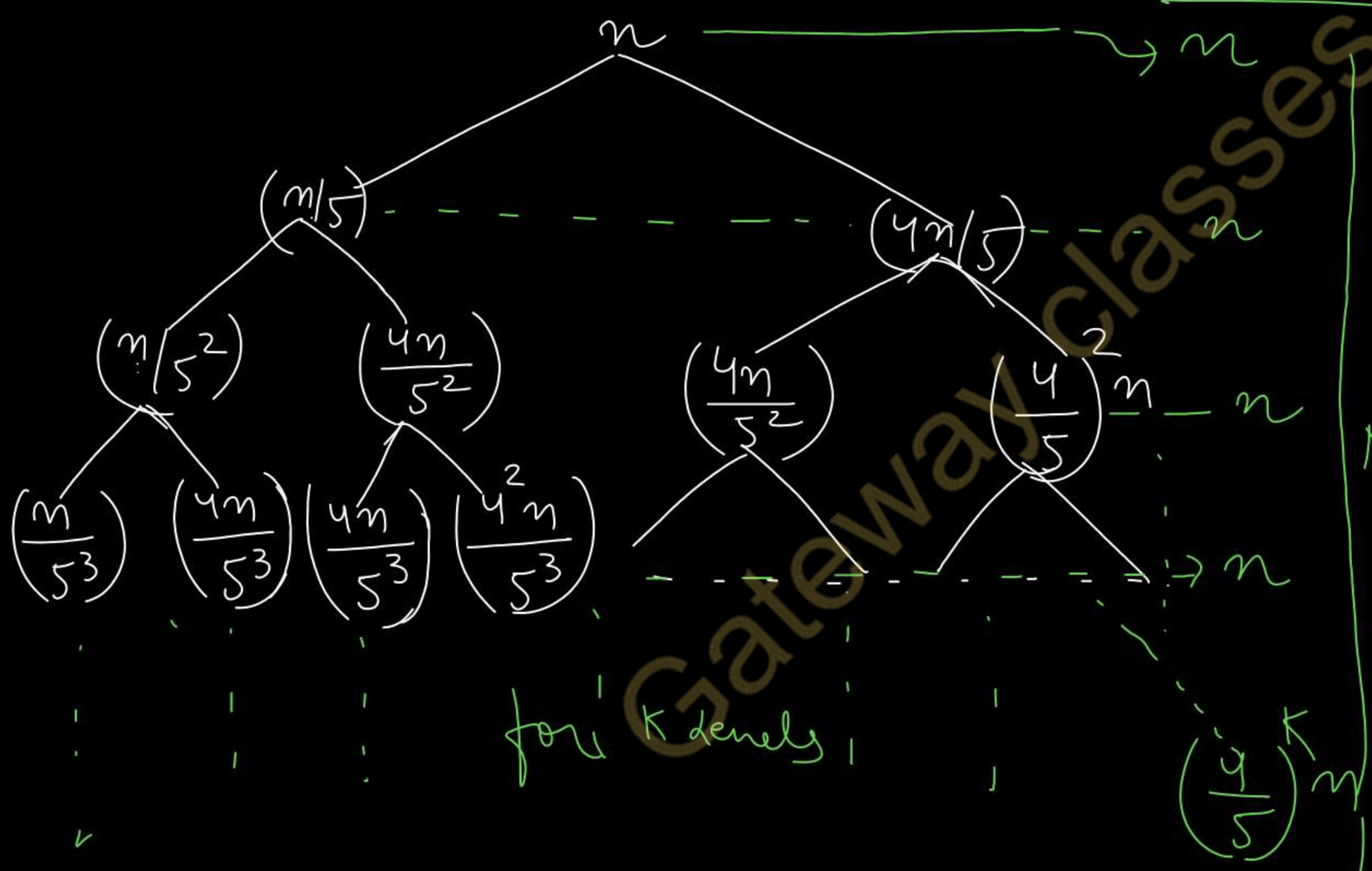
$= n \times (\log_2 n + 1)$

$T(n) = n \log_2 n + n$

$T(n) = O(n \log n)$

Q.2 Solve the recurrence $T(n) = T(n/5) + T(4n/5) + n$ using recursion tree method.

$T(n) = T(n/5) + T(4n/5) + n$



Total cost

$n + n + n + \dots + \left(\frac{4}{5}\right)^K n$

$\frac{n}{(5/4)^K} = 1$

$n = \left(\frac{5}{4}\right)^K$

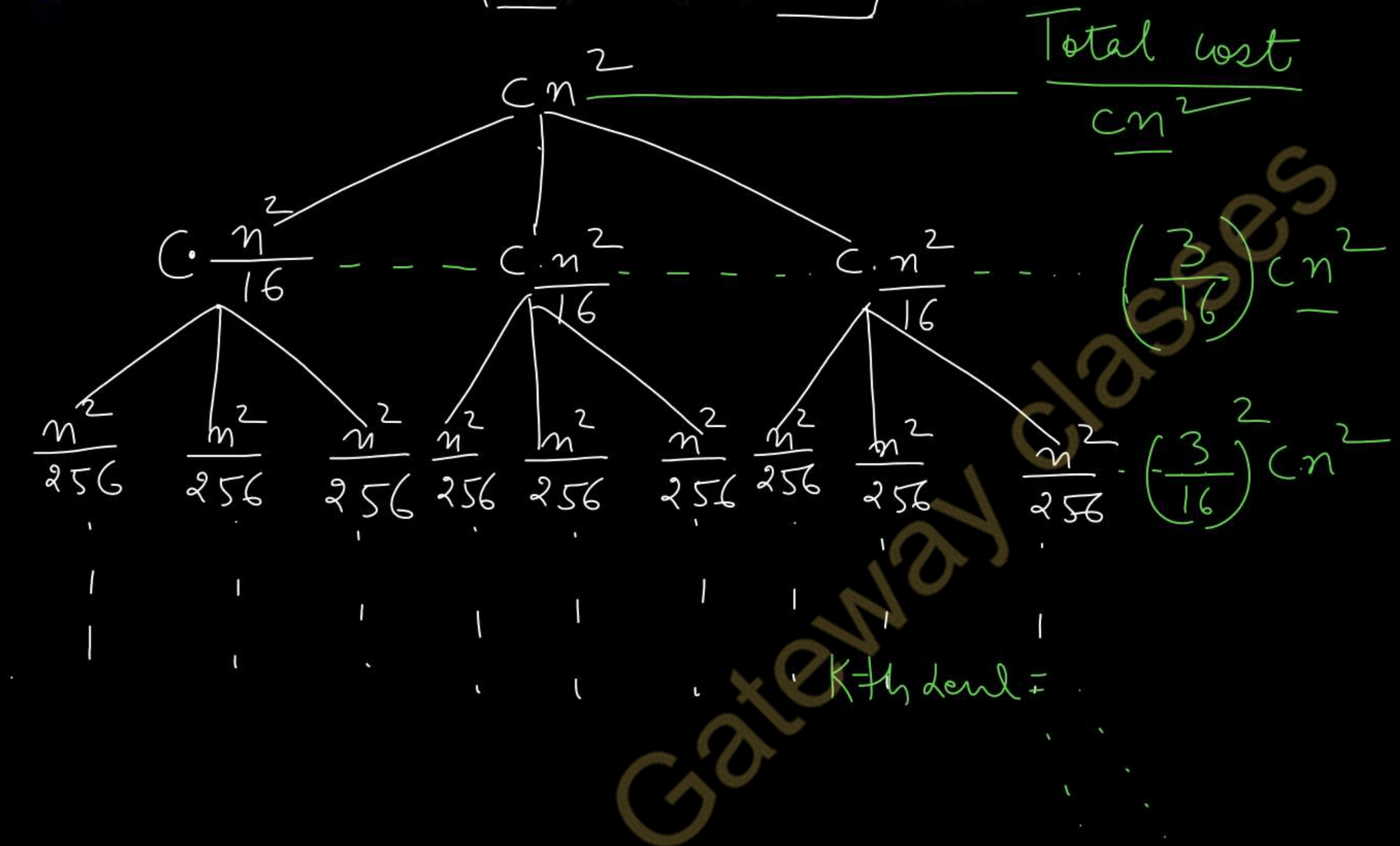
$K = \log_{(5/4)} n$

No. of levels $= (K+1)$

$T(n) = \left(\log_{5/4} n + 1\right) \times n$

$T(n) = n \log_{5/4} n$

Q.3 Solve the recurrence $T(n) = 3T(n/4) + cn^2$ using recursion tree method . (AKTU 2021-22)



Total cost $= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots\cdots$

$= cn^2 \left[1 + \left(\frac{3}{16}\right) + \left(\frac{3}{16}\right)^2 + \cdots\cdots \right]$

$T(n) = cn^2 \left[\frac{1}{1 - \frac{3}{16}} \right]$ $\quad 1 + \cdots\cdots$

$= cn^2 \left(\frac{16}{13}\right)$

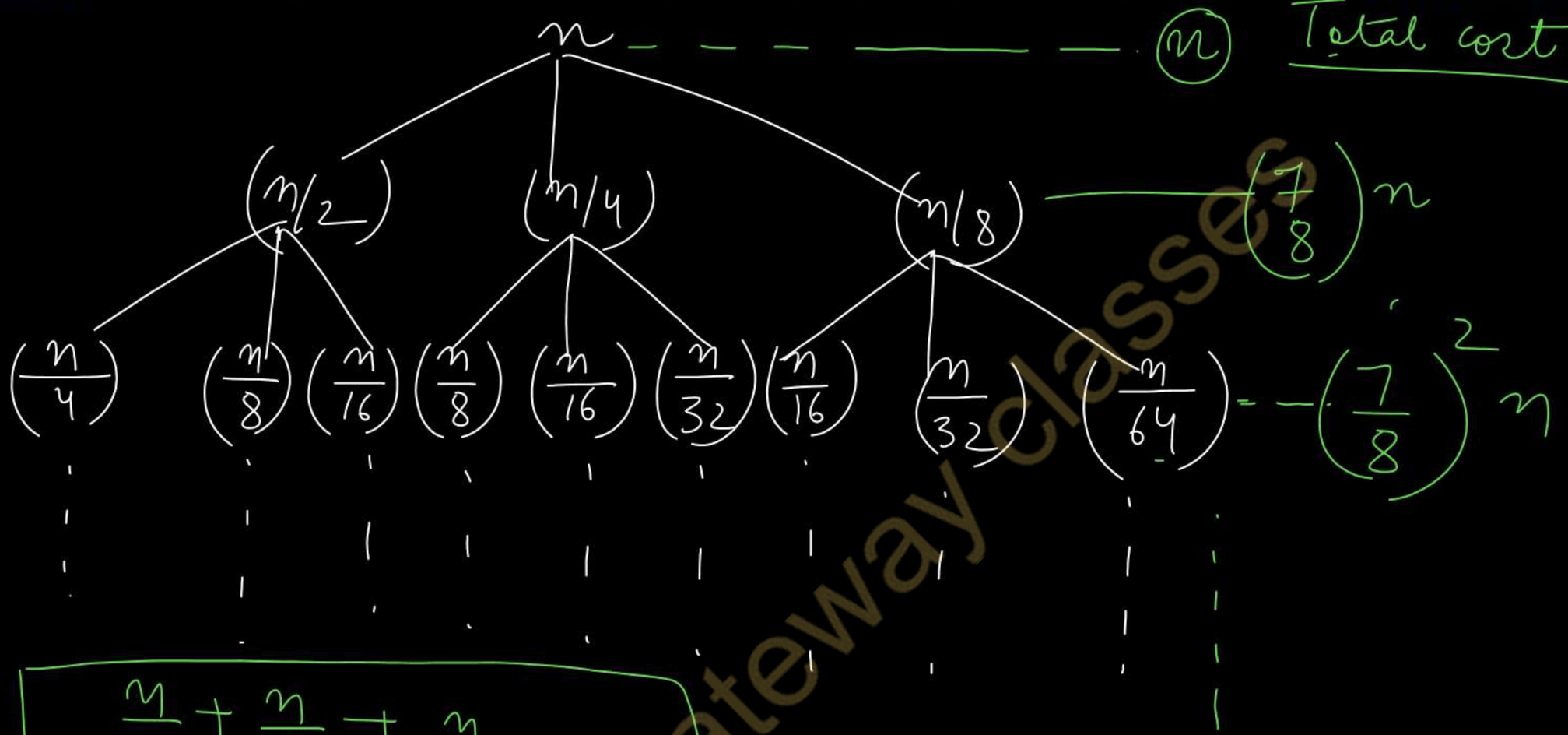
$T(n) = O(cn^2)$

GP $\Rightarrow 1 + ar + ar^2 + \cdots\cdots \infty$

$sum = \frac{a}{1 - r}$ $\quad r < 1$

$r = \frac{3}{16}$

Q.4 Solve using recursion tree: $T(n) = T(n/2) + T(n/4) + T(n/8) + n$. (AKTU 2019-20)



Total cost = Sum of costs at all levels.

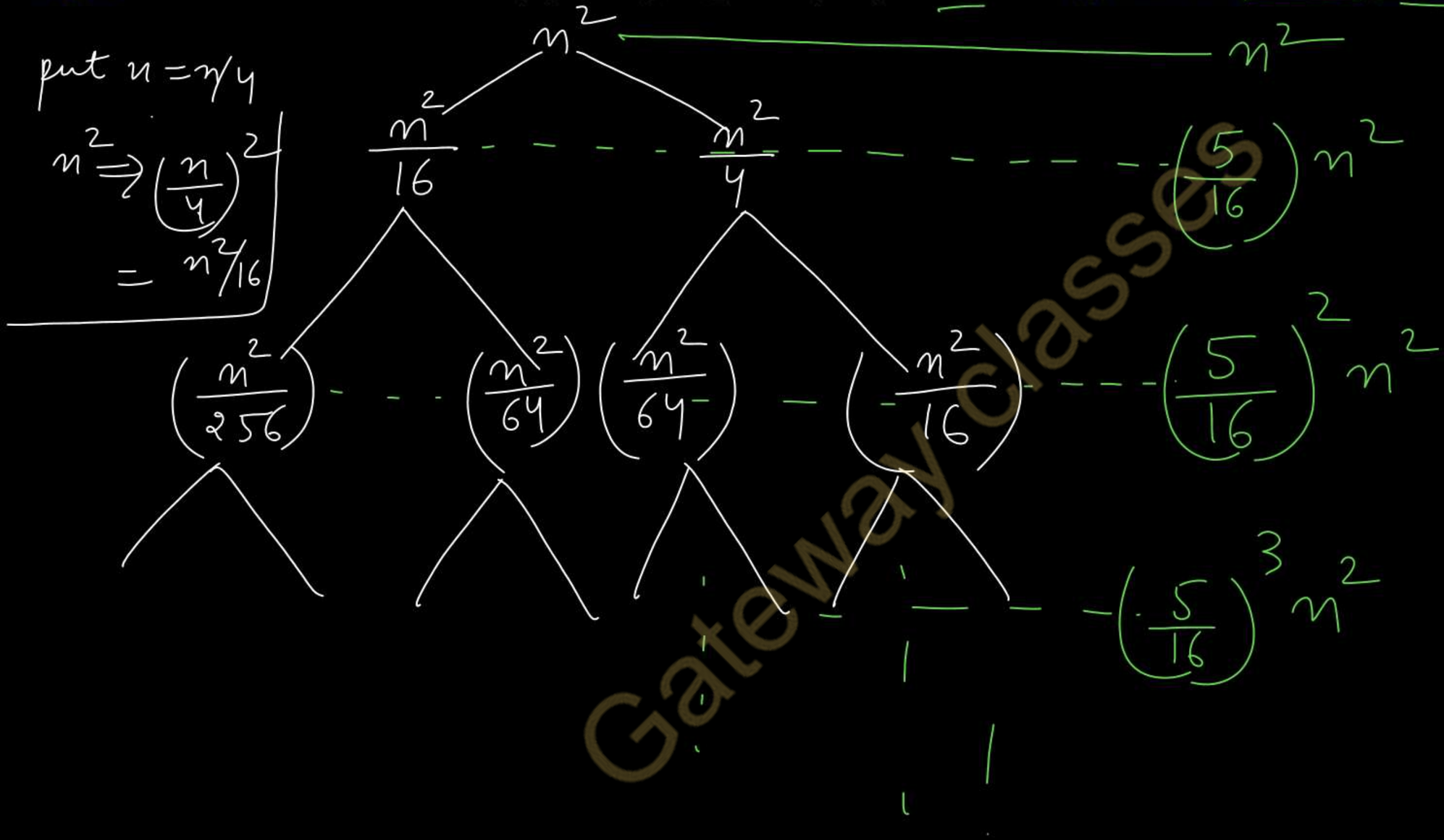
$$T(n) = n + \left(\frac{7}{8}\right)n + \left(\frac{7}{8}\right)^2 n + \left(\frac{7}{8}\right)^3 n - - - -$$

$$= n\left[1 + \frac{7}{8} + \left(\frac{7}{8}\right)^2 + \left(\frac{7}{8}\right)^3 + \text{—} - -\right]$$

$$\approx 1$$

$$T(n) = O\left(n \cdot \left[\frac{1}{1 - \frac{7}{8}}\right]\right) \Longrightarrow \boxed{T(n) = O(n)}$$

Q.5 Solve the recurrence $T(n) = T(n/4) + T(n/2) + n^2$. (AKTU 2022-23, 2020-21)



$$T(n) = n^2 + \left(\frac{5}{16}\right) n^2 + \left(\frac{5}{16}\right)^2 n^2 + \cdots$$

$$= n^2 \left[1 + \frac{5}{16} + \left(\frac{5}{16}\right)^2 + \cdots \right]$$

$$\leq 1$$

$$= n^2 \times \left[\frac{1}{1 - \frac{5}{16}} \right]$$

$$\boxed{T(n) = O(n^2)}$$

Q.6 Solve recurrence $T(n) = T(n-1) + \log n$ using substitution (also called iterative substitution/ backward substitution) method.

$n \geq 0$ $\quad n = 0 \quad T(n) = 1$

$$T(n) = T(n-1) + \log n \quad \text{——} (1)$$

$$T(n-1) = T(n-2) + \log(n-1)$$

$$\therefore T(n) = [T(n-2) + \log(n-1)] + \log n$$

$$(II)\ T(n) = T(n-2) + \log(n-1) + \log n \quad \text{——} (2)$$

$$(III)\ T(n) = T(n-3) + \log(n-2) + \log(n-1) + \log n \quad \text{——} (3)$$

K-times

$$T(n) = T(n-K) + \log(n-(K-1)) + \log(n-(K-2)) + \cdots \log n$$

$$T(n) = T(n-k) + \log(n-(k-1)) + \log(n-(k-2)) + \cdots \log n$$

Put $(n-k) = 0$

$$\boxed{k = n}$$

$$T(n) = T(n-n) + \log 1 + \log 2 + \log 3 \cdots \log n$$

$$= \log(1 \times 2 \times 3 \times 4 \cdots n)$$

$$T(n) = \log(n!)$$

$$T(n) \leq \log(n^n) \Longrightarrow \boxed{T(n) = O(n \log n)}$$

upper bound of $n!$

$$n! = n \times (n-1) \times (n-2) \times \cdots 1$$

for calculating the upper bound

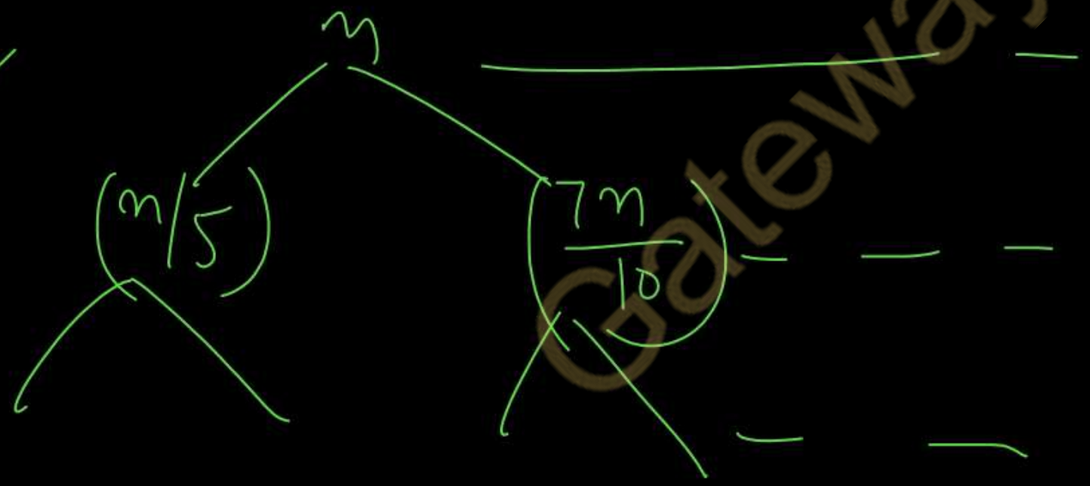
$$n! \leq n \times n \times n \times n \cdots$$

$$2^n < n! < n^n$$

$$n! \leq n^n$$

Q.7 Solve recurrence $T(n) = T(n/4) + T(3n/4) + n^2$.

Q.8 Solve recurrence $T(n) = T(n/5) + T(7n/10) + n$.



...AKTU PYQs

1. Discuss the basic steps in the complete development of an algorithm. (AKTU 2022-23,2017-18)
2. What do you mean by algorithm? Explain different characteristics of algorithms. (AKTU 2023-24)
3. How do you compare the performance of various algorithms? (AKTU 2019-20)
4. Explain how algorithms are analyzed? (AKTU 2021-22)
5. What is asymptotic notation? Explain Omega notation. (AKTU 2020-21)
6. Show that the equation is correct: $10n^2+9=O(n^2)$ (AKTU 2023-24)
7. What is recurrence relation? How it is solved using master's theorem? (AKTU 2020-21)
8. Rank following by growth rate: n, $2^{\lg\sqrt{n}}$, log n, log (log n), $\log^2 n$, $(\lg n)^{\lg n}$, 4, $(3/2)^n$, n! (AKTU 2018-19)
9. Rank following by growth rate: $n^{2.5}$, $\sqrt{2^n}$, n+10, 10n, 100n, $n^2\log n$ (AKTU 2019-20)

$f(n) = O(g(n))$

$f(n) \leq c \cdot g(n)$ for all $n \geq n_0$

AKTU PYQs

10. Solve the recurrence $T(n) = T(n-1) + n^4$ (AKTU 2022-23, 2020-21))

11. Solve the recurrence $T(n) = 2T(n/2) + n$ and $T(1)=1$ (AKTU 2021-22)

12. Solve the recurrence $T(n) = 4T(n/2) + n^2$ and $T(1)=1$ (AKTU 2020-21)

13. Solve the recurrence $T(n) = 2T(n/2) + n^2 + 2n+1$ and $T(1)=1$ (AKTU 2019-20)

14. Use a recursion tree to give an asymptotically tight solution to the following recurrence

$T(n) = T(\alpha n) + T((1 - \alpha)n) + cn$, where α is a constant in the range $0 < \alpha < 1$ and $c > 0$ is also a constant.

(AKTU 2018-19)

15. The recurrence $T(n) = 7T(n/3) + n^2$ describes the running time of an algorithm A. Another competing algorithm B has a running time of $S(n) = a\,S(n/9) + n^2$. What is the smallest value of 'a' such that A is asymptotically faster than B.? (AKTU 2018-19)

AKTU

B.Tech 5th Sem

GATEWAY CLASSES

CS IT & CS Allied

DAA : Design & Analysis Of Algorithm

UNIT-1: Introduction

Lecture-7

Today's Target

- Algorithm Development steps
- Algorithm Design Approaches
- AKTY PYQs

By Dr. Nidhi Parashar Ma'am

- M.Tech Gold Medalist
- Net Qualified

Basic steps in development of algorithm AKTU,2021-22,2022-23

Consider the example to add three numbers and print the sum.

Step-3. Analyze algo

Step 1: Fulfilling the pre-requisites

- The problem to be solved : Add 3 numbers and print their sum.
- The constraints to be considered : The numbers must contain only digits and no other characters.
- The input to be taken : The three numbers to be added.
- The output expected : The sum of the three numbers taken as the input i.e. a single integer value.
- The solution in the given constraints: The solution consists of adding the 3 numbers. It can be done with the help of the '+' operator, or bit-wise, or any other method.

Step 2: Designing the algorithm

Step 3: Testing the algorithm by implementing it.

Types of Computational problems

1. Optimization problems (Minimization / Maximization)
   - This type of problem asks…
   - What is the optimal solution to this problem x?
   - Example: finding minimum spanning tree

Minimizing the objective function $f(n)$

Maximize the objective function

2. Decision Problems:
   - A problem with yes/no answer
   - Example: Does a graph G has a minimum spanning tree having cost <= x

MST with cost $\leq x$ $\Rightarrow$ YES / NO

Algorithm Design Approaches AKTU 2019-20,2021-22,2022-23

There are various approaches to design an algorithm

1. Greedy approach
2. Divide and Conquer
3. Dynamic Programming
4. Backtracking
5. Branch and Bound

Algorithm Design Approaches AKTU 2019-20,2021-22,2022-23

1 Greedy Algorithm

- Used for solving optimization problems
- Selects the best option available at the moment without worrying that it will bring the overall optimal result.
- Never reverses the earlier decision even if the choice is wrong. It works in a top-down approach.
- May not produce the best result for all the problems.

Example:

- Fractional Knapsack problem
- Prim's and Kruskal's algos to find minimum spanning tree

Pseudo-Code for Greedy Algorithm (Imp)

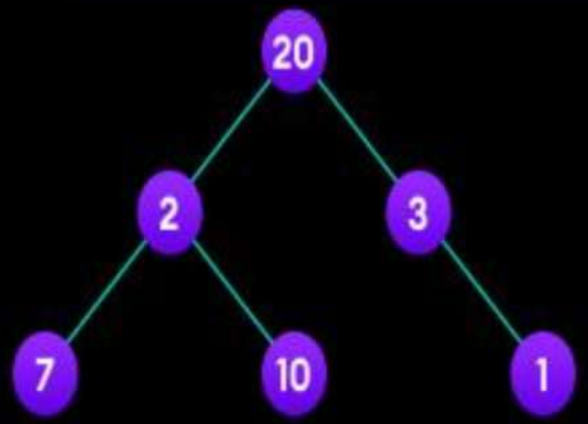
```
Algorithm Greedy (a, n) //a[1:n] contains the n inputs.
{
    solution:=∅; //initialize the solution. set.
    for i= 1 to n do
    {
        X= Select (a);
        if Feasible ( solution, X) then
            solution: = Union(solution, X);
    }
    return solution;
}
```

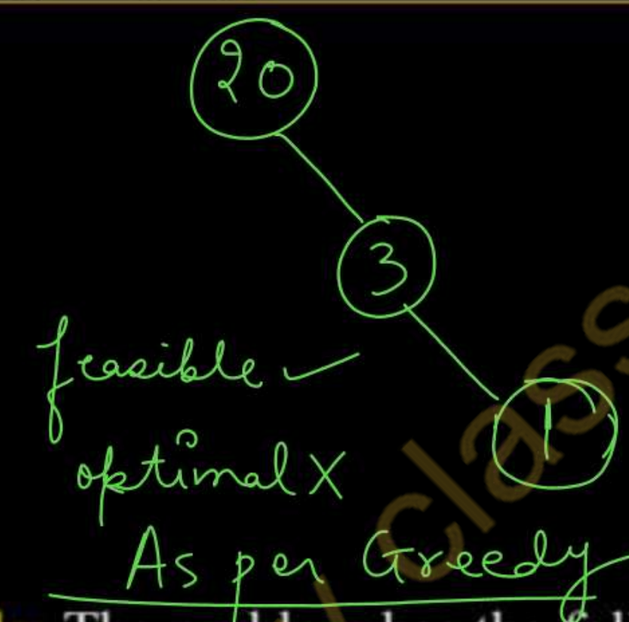
X = Selected choice.
a → No. of available choices

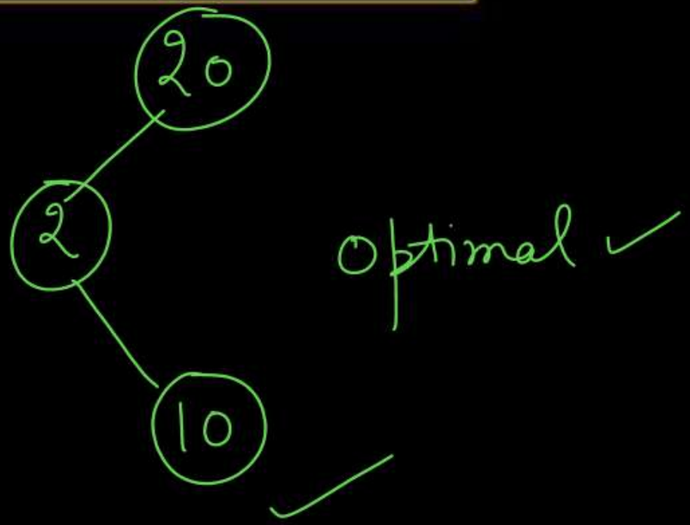
Solution = $\{x_1, x_2, x_3 \ldots\}$

- The function Select selects an input from 'a', removes it and assigns its value to 'x'.
- Feasible is a Boolean valued function, which determines if 'x' can be included into the solution vector.

Problem Statement : Apply greedy approach to this tree to find the longest route





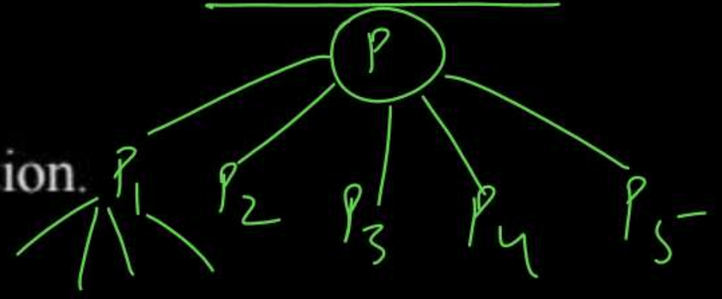


When Greedy approach is applicable: The problem has the following properties:

- Greedy Choice Property: Optimal solution can be found by choosing the best choice at each step without reconsidering the previous steps once chosen.
- Optimal Substructure: Optimal overall solution corresponds to the optimal solution to its subproblems.

Divide and Conquer:

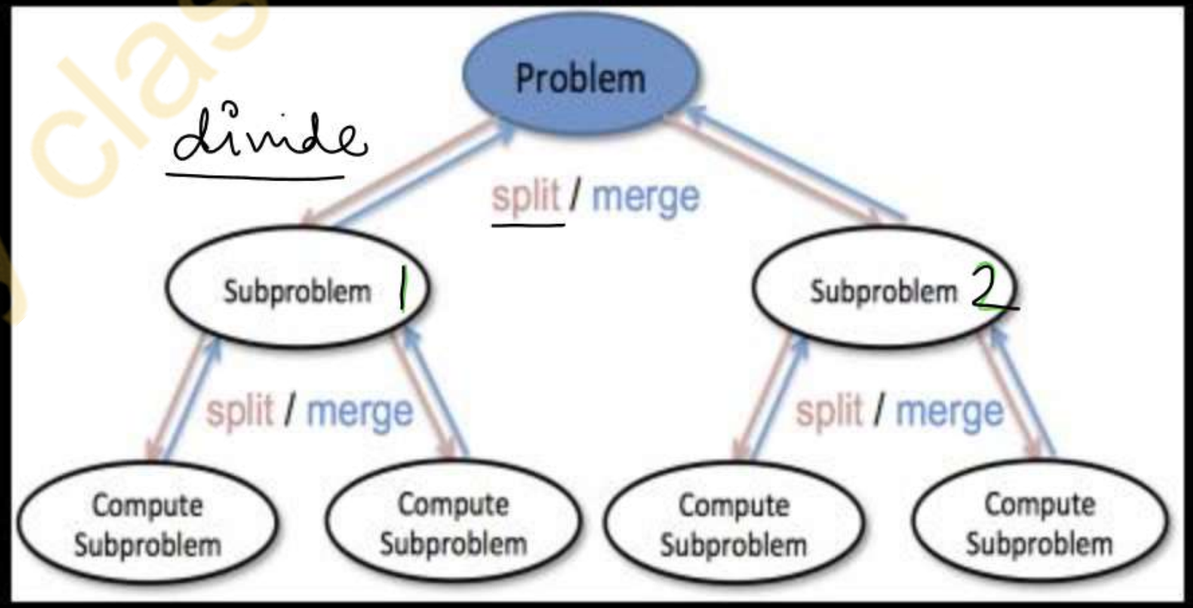
- Recursively divide the problem into smaller subproblems until they become simple enough to be solved directly.
- Solutions to the subproblems are combined to produce the overall solution.



Pseudocode:

```
DC(p)
{     if(small(p))
          S(p);
      else
          Divide P into p1, p2, ….pk
          Apply DC(p1),.. DC(pk)
          Combine((DC(p1)),.. (DC(pk)));
}
```

- for base case
- used for termination of recursion calls



- Example: Merge Sort, Quick Sort

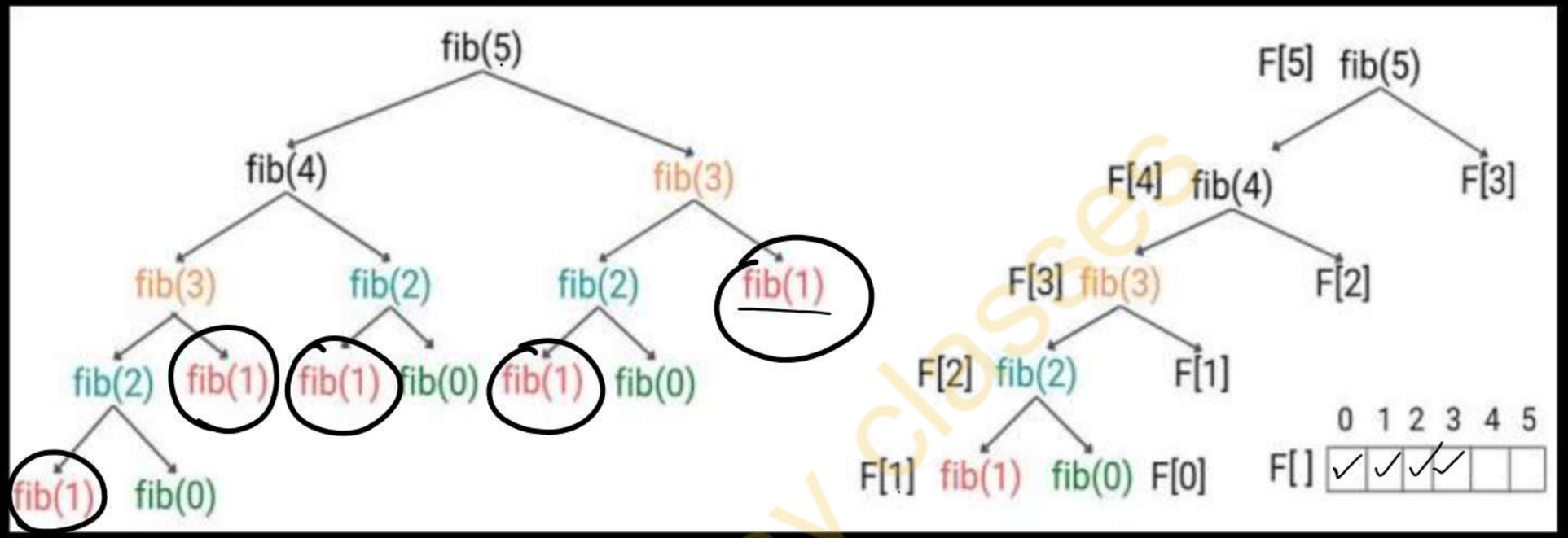
WHY Dynamic Programming?

- Divide and conquer is efficient when subproblems are independent. In case of repeated subproblems it does much more computation than expected leading to exponential time complexity. Such problems can be effectively solved using dynamic programming.

Dynamic Programming(AKTU 2019-20)

- Used for solving optimization problems.
- Divide the main problem into smaller subproblems(Overlapping).
- Solve each subproblem and store the solution in a table or array.
- Use stored solutions to build up the solution to the main problem.
- Avoids redundancy by storing solutions and ensures that each subproblem is solved only once, reducing computation time.
- Example: Fibonacci Series, 0-1 knapsack problem, sum of subset problem

Problem Statement : Calculating Nth Fibonacci term



Recursive Solution

Dynamic programming

The following problems can be solved using Dynamic programming:

- Optimal substructure: An optimal solution to the problem contains the optimal solutions to all subproblems.
- Overlapping subproblem: A recursive solution contains a small number of distinct subproblems repeated many times.

Comparison among Greedy, Divide and conquer and dynamic programming

	Greedy Algorithm	Divide and conquer	Dynamic Programming
1.	Follows Top-down approach	Follows Top-down approach	Follows bottom-up approach
2.	Solve optimization problem	Solve decision problem	Solve optimization problem
3.	The optimal solution is generated without revisiting previously generated solutions	Solution of subproblem is computed recursively more than once.	The solution of subproblems is computed once and stored in a table for later use.
4.	May or may not generate an optimal solution.	Does not aim for the optimal solution	Always generates optimal solution.

	Greedy Algorithm	Divide and conquer	Dynamic Programming
5.	Iterative in nature.	Recursive in nature.	May or may not use recursion(optional)
6.	Efficient and fast than divide and conquer.	less efficient and slower.	more efficient but slower than greedy.
7.	extra memory is not required.	some memory is required. (stack)	more memory is required to store subproblems for later use.
8.	Examples: Fractional Knapsack problem, Activity selection problem, Job sequencing problem.	Examples: Merge sort, Quick sort, Strassen's matrix multiplication.	Examples: 0/1 Knapsack, All pair shortest path, Matrix-chain multiplication.

Backtracking(AKTU 2023-24):

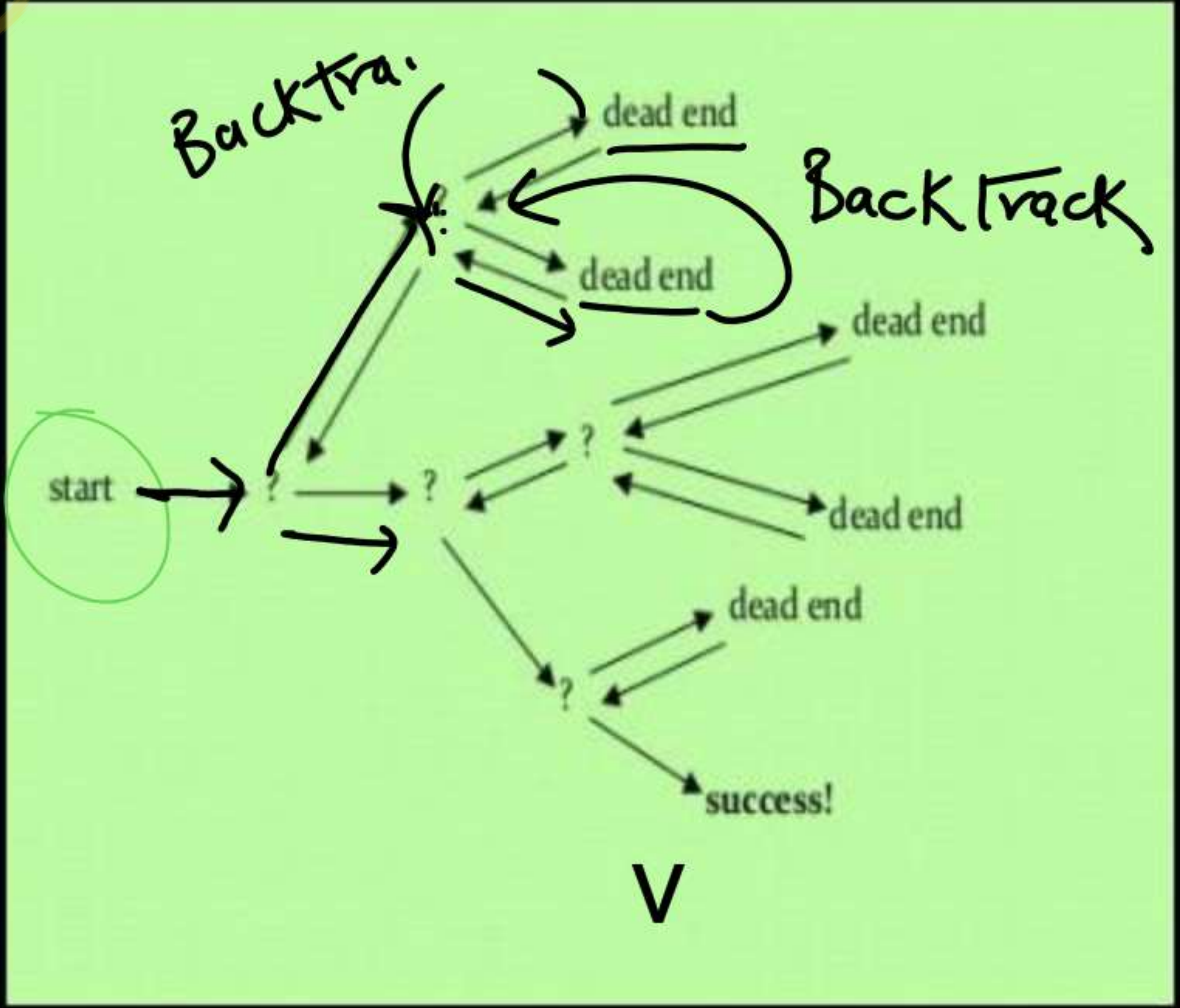
- Tries each possibility until they find the right one.
- It is a depth-first search of the set of possible solution.
- During the search, if an alternative doesn't work, then backtrack to the choice point, the place which presented different alternatives, and tries the next alternative.

- Example: N-Queens's Problem

Pseudo-code for backtracking problems:

explore(choices):

    if there are no more choices to make: stop.
    else:
        • Make a single choice C from the set of choices.
          ----Remove C from the set of choices.
        • explore the remaining choices.
        • Un-make choice C.
          ----Backtrack!



Why Backtracking:

Problem: Imagine you have 3 closed boxes, among which 2 are empty and 1 has a gold coin. Your task is to get the gold coin.

Why dynamic programming fails: Each and every box is independent of each other and opening/closing state of one box can not determine the transition for other boxes. Hence DP fails.

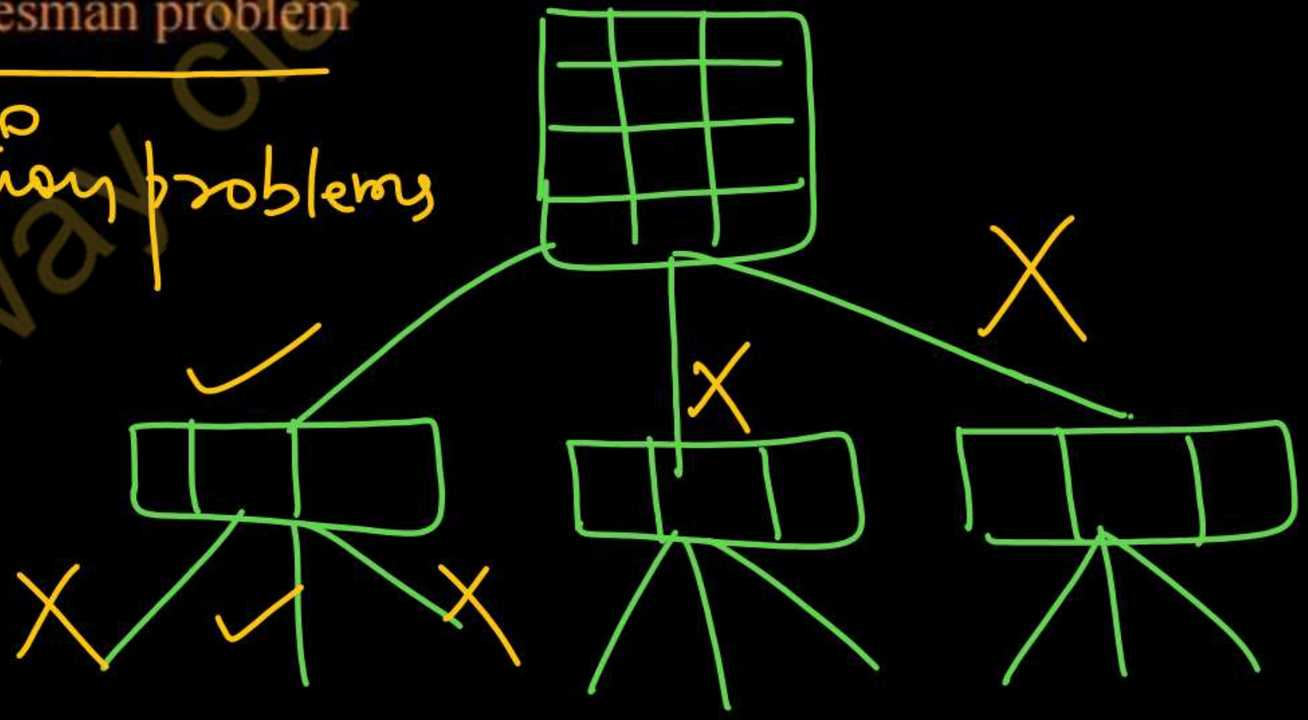
Why greedy fails: Each and every box has equal probability of having a gold coin i.e 1/3 hence there is no criteria to make a greedy choice.

Why Backtracking works: Simply brute forcing each and every choice, hence we can one by one choose every box to find the gold coin, If a box is found empty we can close it back which acts as a Backtracking step.

Branch and Bound:

- Branching: Original problem is partitioned into smaller subproblems(branches).
- Bounding: Compare the computed optimal solution with the best known solution.
- Elimination: Identify nodes which do not lead to the best solution and eliminate them.
- Selection: Exploration strategy used such as breadth first search, depth first search etc..
- Example: Job sequencing, Travelling salesman problem

used to solve optimization problems



Advantages of Branch and Bound Algorithm

- It can reduce the time complexity by avoiding unnecessary exploration of the state space tree.
- It has different search techniques(DFS,BFS, least cost search) that can be used for different types of problems.

Disadvantages of Branch and Bound Algorithm

- In the worst case scenario, it may search for all the combinations to produce solutions.
- It can be time consuming if the state space tree is too large.

Difference between Backtracking and Branch and Bound.(AKTU 2020-21, 2022-23)

Parameter	Backtracking	Branch and Bound
Approach	Write definitions	Write definitions
Traversal	traverses the state space tree by DFS (Depth First Search) manner.	traverse the tree by DFS or BFS (breadth first search).
Function	Involves feasibility function.	Involves a bounding function.
Problems	used for solving Decision Problem.	used for solving Optimization Problem.
Applications	Useful in solving N-Queen Problem, Sum of subset, Hamilton cycle problem, graph coloring problem	Useful in solving Knapsack Problem, Travelling Salesman Problem.
Next move	Next move from current state can lead to bad choice.	Next move is always towards better solution.
Solution	On successful search of solution in state space tree, search stops.	Entire state space tree is search in order to find optimal solution.

AKTU PYQs

1. Discuss the basic steps in the complete development of an algorithm. (AKTU 2022-23)
2. Explain Backtracking. (AKTU 2023-24)
3. Differentiate Backtracking and Branch and Bound Techniques .(AKTU 2020-21, 2022-23)
4. Discuss backtracking problem solving approach. (AKTU 2022-23)
5. Explain "greedy algorithm" and their characteristics.(AKTU 2019-20, 2021-22, 2022-23)
6. Explain Branch and bound in brief. (AKTU 2021-22)
7. What is dynamic programming? How it is different from recursion? Explain with example. (AKTU 2019-20)
8. Compare various algorithm paradigms such as divide and conquer, dynamic programming and greedy approach. (AKTU 2019-20)

AKTU

B.Tech 5th Sem

GW GATEWAY CLASSES

CS IT & CS Allied

DAA : Design & Analysis Of Algorithm

UNIT-1: Introduction

Lecture-8

Today's Target

- Introduction to sorting
- Insertion Sort with complexity analysis
- AKTY PYQs

By Dr. Nidhi Parashar Ma'am
- M.Tech Gold Medalist
- Net Qualified

SORTING

In-place sorting:

Do not require any extra space for sorting the array.

Example: Bubble sort, Selection Sort, Insertion Sort, Heapsort, Quick sort

Not-in-place Sorting :

The program requires space which more than or equal to the elements being sorted.

Example: Merge-sort

Internal sorting:

Sorting data that can fit entirely in the main memory of a computer system. Commonly used when sorting relatively small datasets.

External sorting:

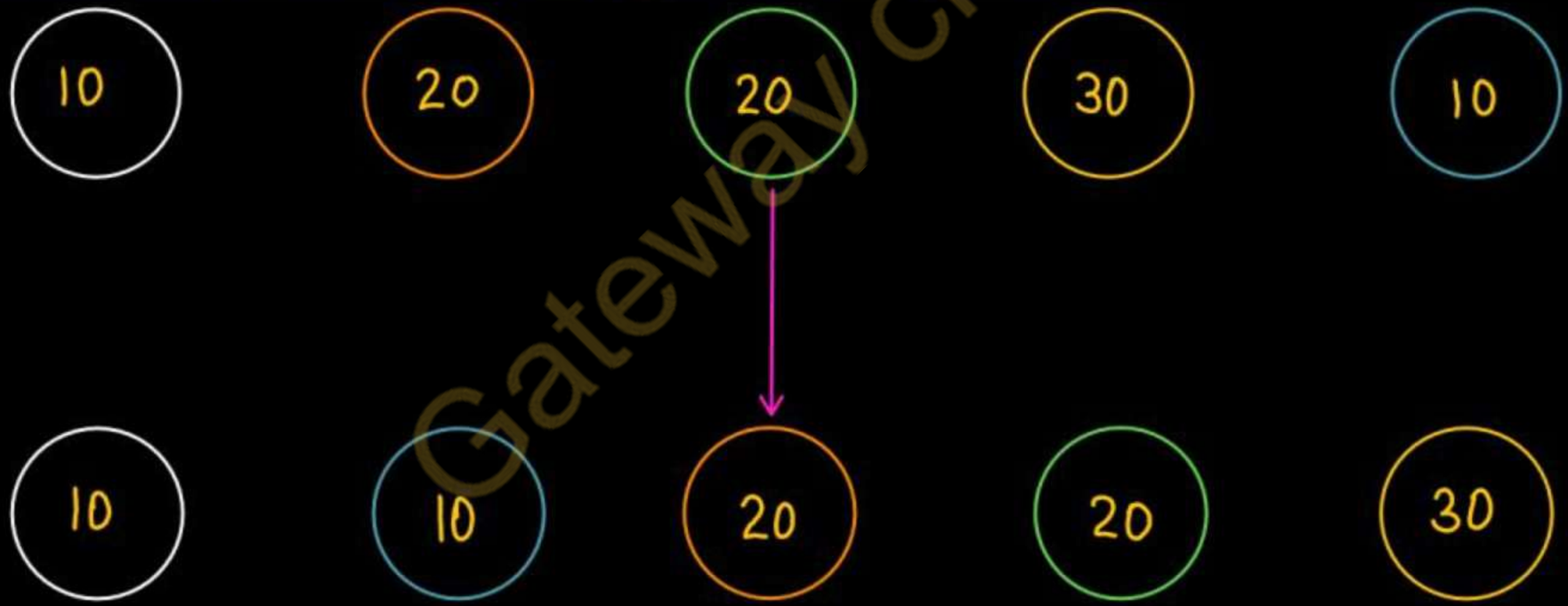
Sorting data that is too large to fit in the main memory of a computer system and must be stored on hard drives. Commonly useful in sorting significantly large datasets that exceed the available memory.

What do you understand by stable and unstable sorting? AKTU 2021-22,2023-24

Stable sorting:

- A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array means initial order of equal items is preserved.
- Stability means that equivalent elements retain their relative positions, after sorting.

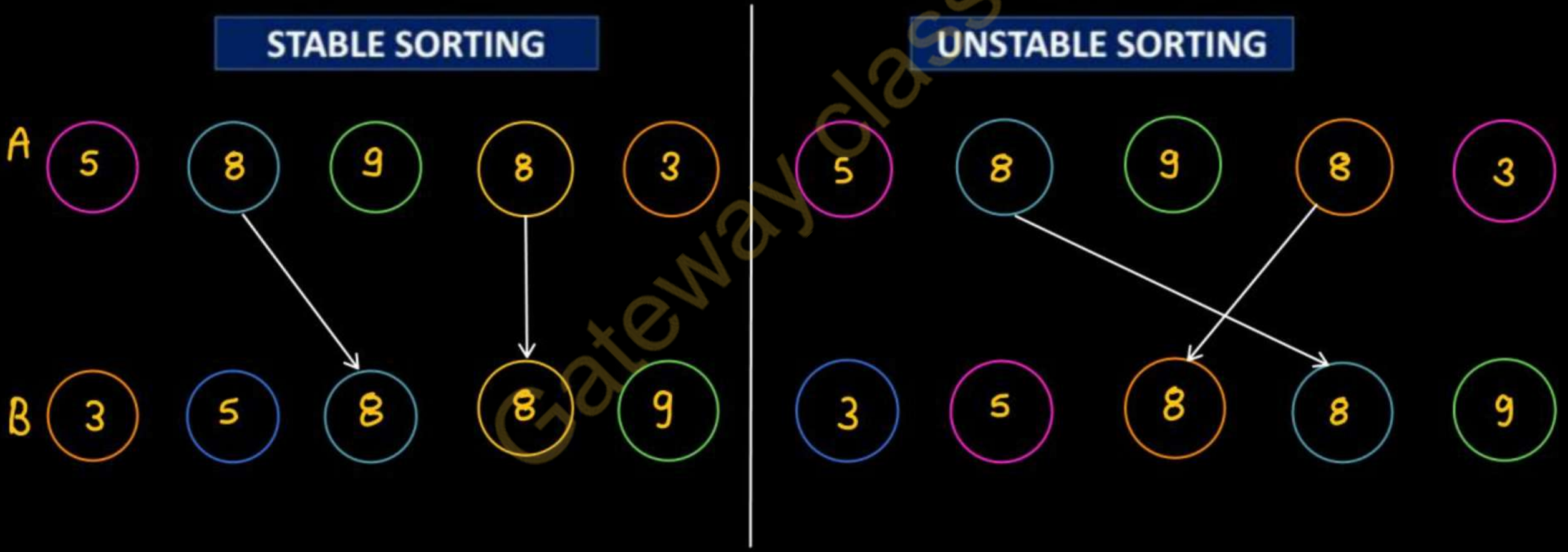
Example: bubble sort, insertion sort, merge sort, counting sort etc.



Unstable sorting:

- In an unstable sorting algorithm the ordering of the same values is not necessarily preserved after sorting.

Example: selection sort, shell sort, Quicksort, Heapsort.



Insertion Sort (AKTU 2020-21, 2019-20, 2022-23)

The simple steps of insertion sort -

Step 1 – If it is the first element, it is already sorted.

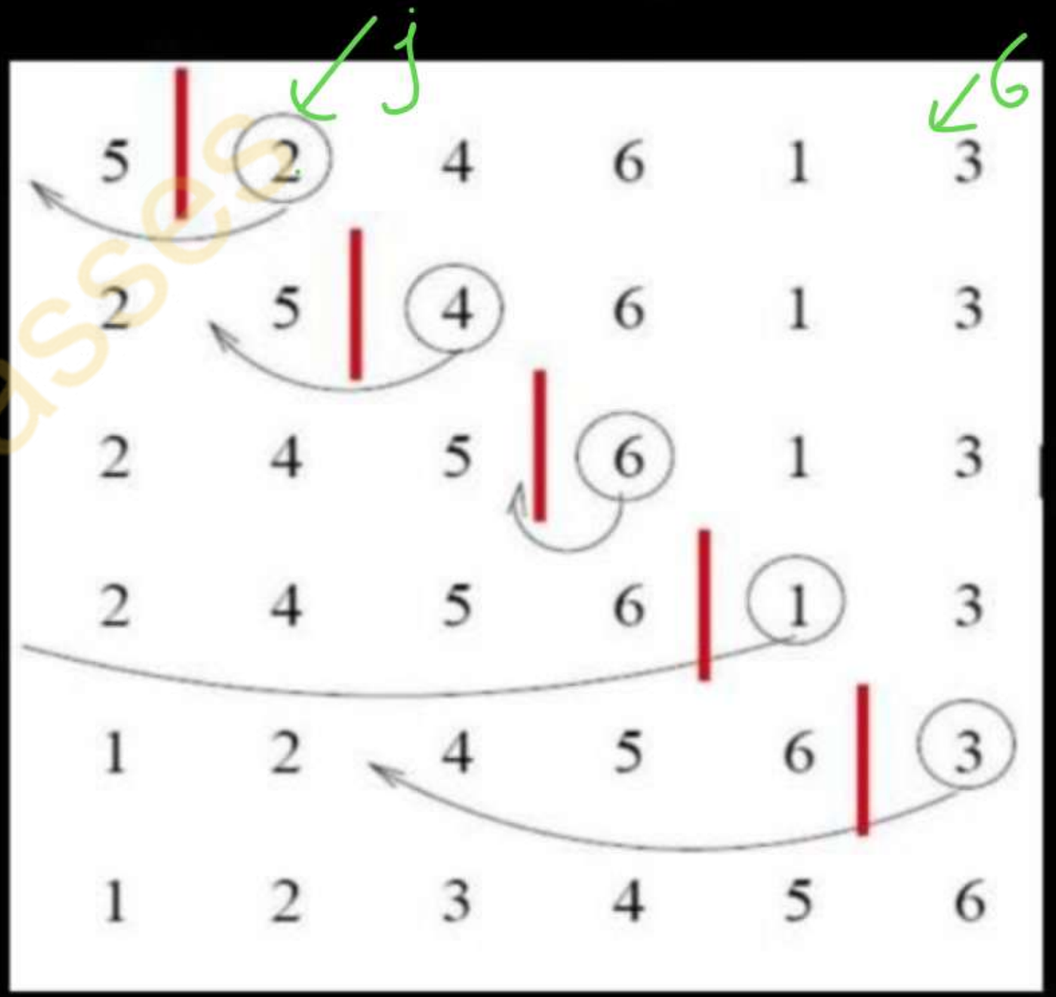
Step 2 – Pick next element

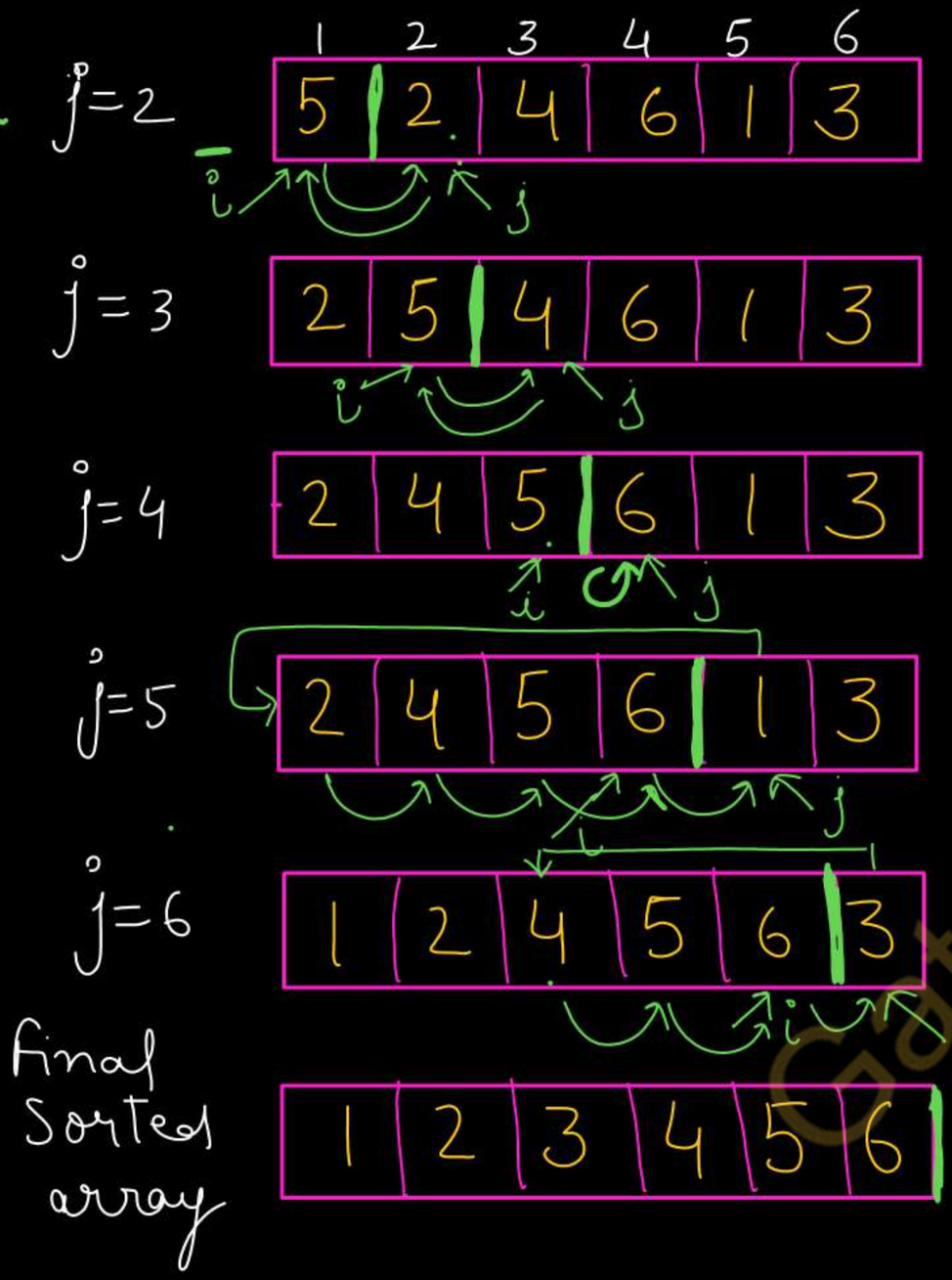
Step 3 – Compare with all elements in the sorted sub-list

Step 4 – Shift all the elements in the sorted sub-list that is greater than the value to be sorted

Step 5 – Insert the value

Step 6 – Repeat until list is sorted





Key = 2
A[i] = 5
A[i] > key

Key = 4
A[i] = 5
A[i] > key

Key = 6
A[i] = 5
A[i] < key

Key = 1
A[i] = 6
A[i] > key

Key = 3
A[i] = 6
A[i] > key

```
Insertion-Sort(A)
for j = 2 to A. length
{
    key = A[j]
    i = j – 1
    while i > 0 and A[i] > key
    {   A[i + 1] = A[i]
        i = i -1 }
    A[i + 1] = key
}
```

Points to Remember about Insertion sort

- stable sort ✓
- In place ✓

Insertion-Sort Time Complexity Analysis

Insertion-Sort(A) n is	cost	No. of times
(1) for j = 2 to A. length	c_1	n
(2) key = A[j]	c_2	$(n-1)$
(3) i = j – 1	c_3	$(n-1)$
(4) while i > 0 and A[i] > key	c_4	$\sum_{j=2}^{n}(t_j)$
(5) A[i + 1] = A[i]	c_5	$\sum_{j=2}^{n}(t_j-1)$
(6) i = i -1	c_6	$\sum_{j=2}^{n}(t_j-1)$
(7) A[i + 1] = key	c_7	$(n-1)$

t_j

Note:
t_j is number of times while loop test is executed for a value of j. It depends on input.

$$T(n) = c_1 \times n + c_2(n-1) + c_3 \times (n-1) + c_4 \times \sum_{j=2}^{n} t_j + c_5 \times \sum_{j=2}^{n}(t_j-1) + c_6 \times \sum_{j=2}^{n}(t_j-1) + c_7 \times (n-1)$$

Best Case Time Complexity

$$T(n) = c_1 \times n + c_2 \times (n-1) + c_3 \times (n-1) + c_4 \times \sum_{j=2}^{n} (t_j) + c_5 \times \sum_{j=2}^{n} (t_j - 1) + c_6 \times \sum_{j=2}^{n} (t_j - 1) + c_7 \times (n-1)$$

For Best case → Array is already sorted $[t_j = 1]$

$$T(n) = c_1 \times n + c_2 \times n - c_2 + c_3 \times n - c_3 + c_4 \times (n-1) + c_5 \times 0 + c_6 \times 0 + c_7 \times (n-1)$$

$$T(n) = (c_1 + c_2 + c_3 + c_4 + c_7)\, n - (c_2 + c_3 + c_4 + c_7)$$

$$T(n) = O(n)$$

$$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} (1) = n-1$$

$$\sum_{j=2}^{n} (t_j - 1) = \sum_{j=2}^{n} (1-1) \Rightarrow 0$$

Worst Case Time Complexity

$$T(n) = c_1 \times n + c_2 \times (n-1) + c_3 \times (n-1) + c_4 \times \sum_{j=2}^{n} (t_j) + c_5 \times \sum_{j=2}^{n} (t_j - 1) + c_6 \times \sum_{j=2}^{n} (t_j - 1) + c_7 \times (n-1)$$

worst case ⇒ Array is in Reverse order

$t_j = (j-1) + 1$ → for checking termination condition

for $(j-1)$ comparisons

$\boxed{t_j = j}$

$$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} (j)$$

we know that $\sum_{j=1}^{n} j = \frac{n(n+1)}{2}$ so

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

$$\sum_{j=2}^{n} t_j - 1 = \sum_{j=2}^{n} (j-1)$$

$$\frac{n(n-1)}{2}$$ ✓

$$T(n) = c_1 x(n) + c_2 x(n-1) + c_3 x(n-1) + c_4 x \left(\frac{n(n+1)}{2} - 1\right) + c_5 x \frac{n(n-1)}{2}$$

$$+ c_6 x \frac{n(n-1)}{2} + c_7 x((n-1)$$

$$\boxed{T(n) = \Theta(n^2)}$$

Average case $\hookrightarrow t_j \simeq j/2$

$$\boxed{T(n) = O(n^2)}$$

AKTU PYQs

1. Discuss the basic steps in the complete development of an algorithm. (AKTU 2022-23)
2. What do you mean by stable and unstable sorting? (AKTU 2019-20)
3. What is stable sorting algorithm? Which sorting algos we have seen are stable or unstable. Give name with explanation. (AKTU 2023-24)

4) Explain Insertion sort in detail with derivation of its time complexities in all cases.

AKTU

B.Tech 5th Sem

GATEWAY CLASSES

CS IT & CS Allied

DAA : Design & Analysis Of Algorithm

UNIT-1: Introduction

Lecture-9

Today's Target

- Merge Sort
- AKTY PYQs

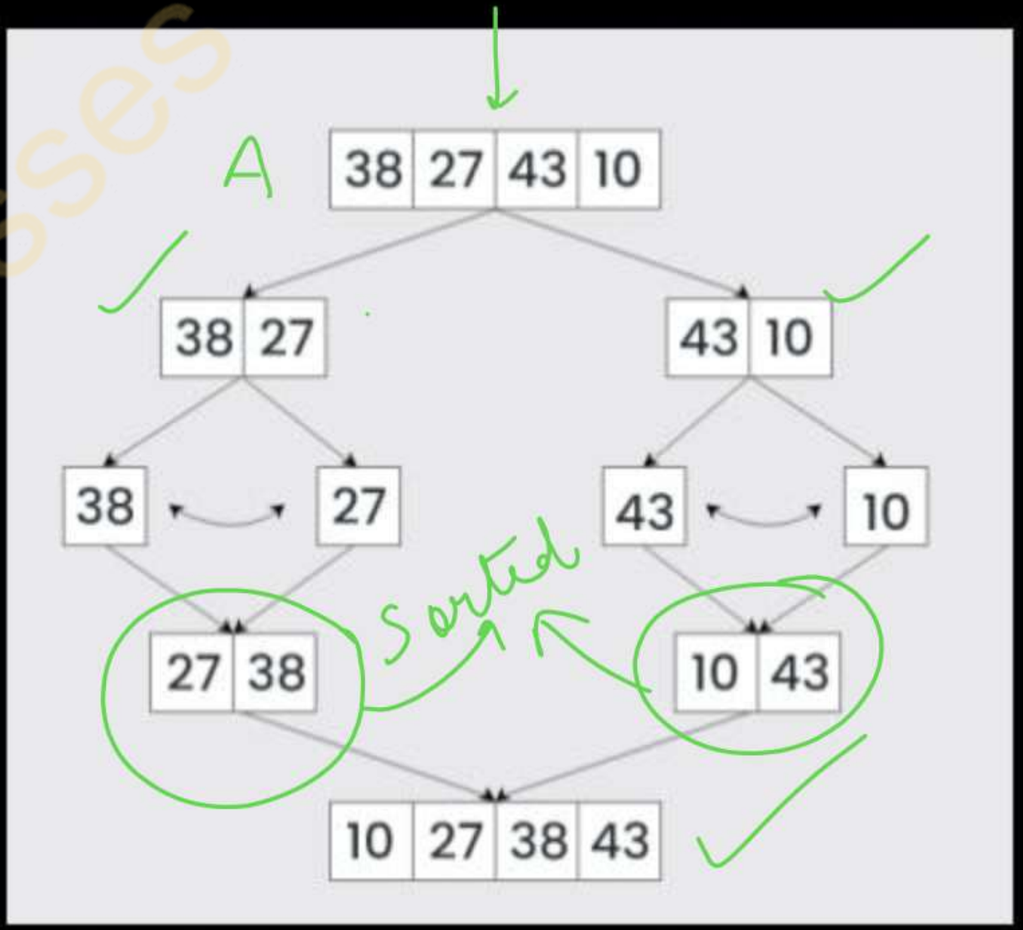
By Dr. Nidhi Parashar Ma'am

- M.Tech Gold Medalist
- Net Qualified

Merge Sort AKTU 2021-22, 2022-23,2023-24

How merge sort works:

- Divide: Divide the list or array recursively into two halves until it can no more be divided.
- Conquer: Each subarray is sorted individually using the merge sort algorithm.
- Merge: The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.



Merge Sort AKTU 2021-22, 2022-23,2023-24

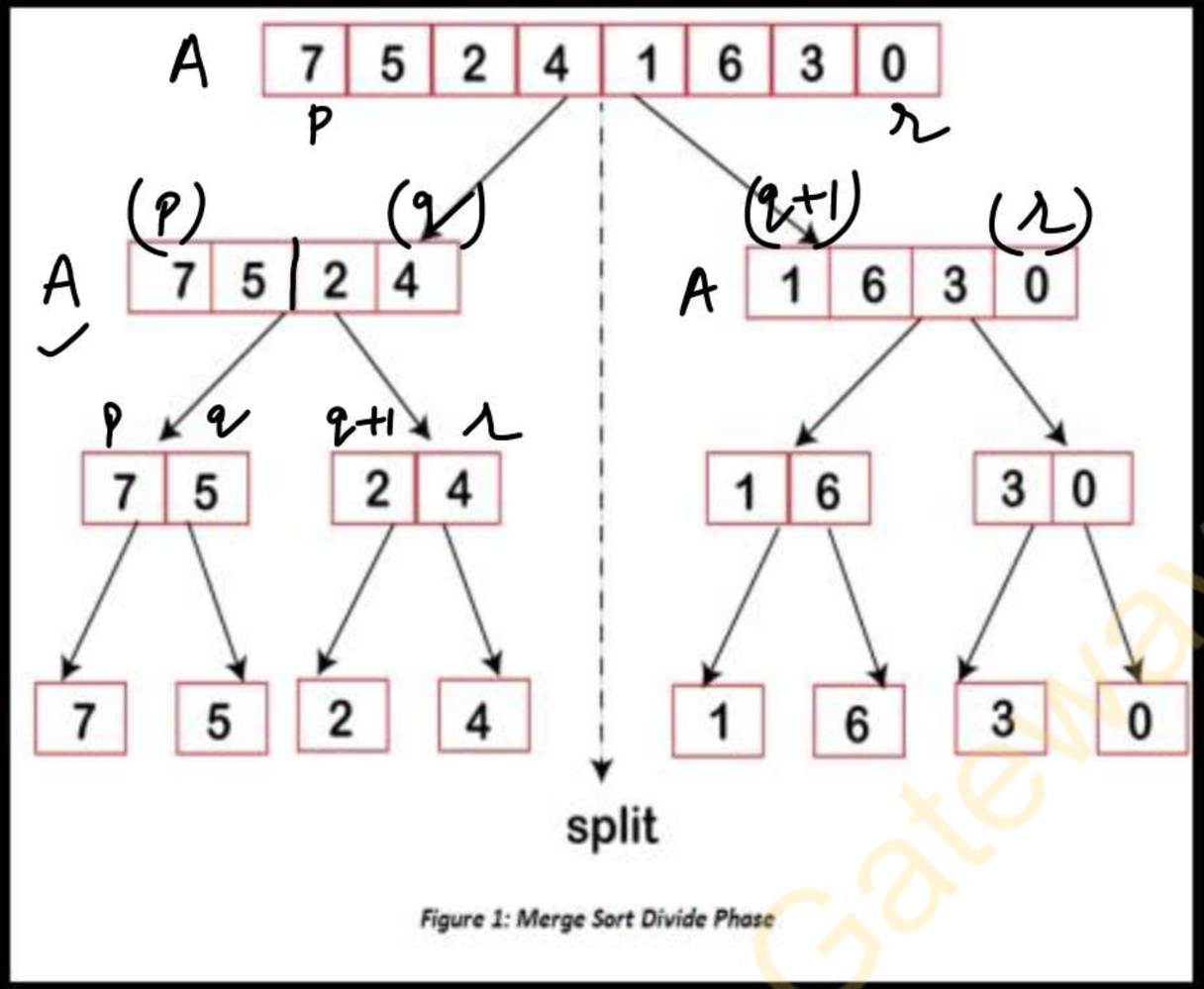


Figure 1: Merge Sort Divide Phase

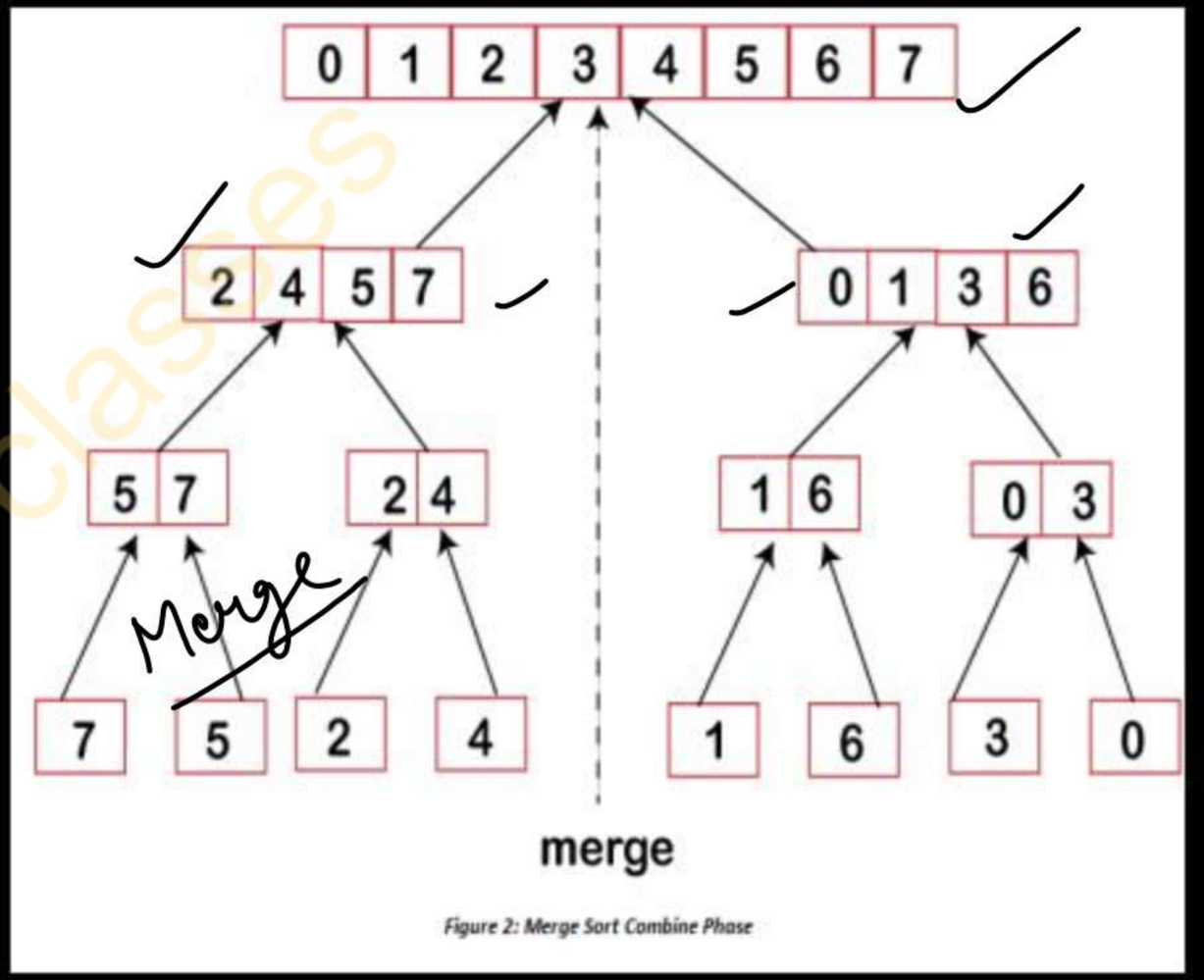
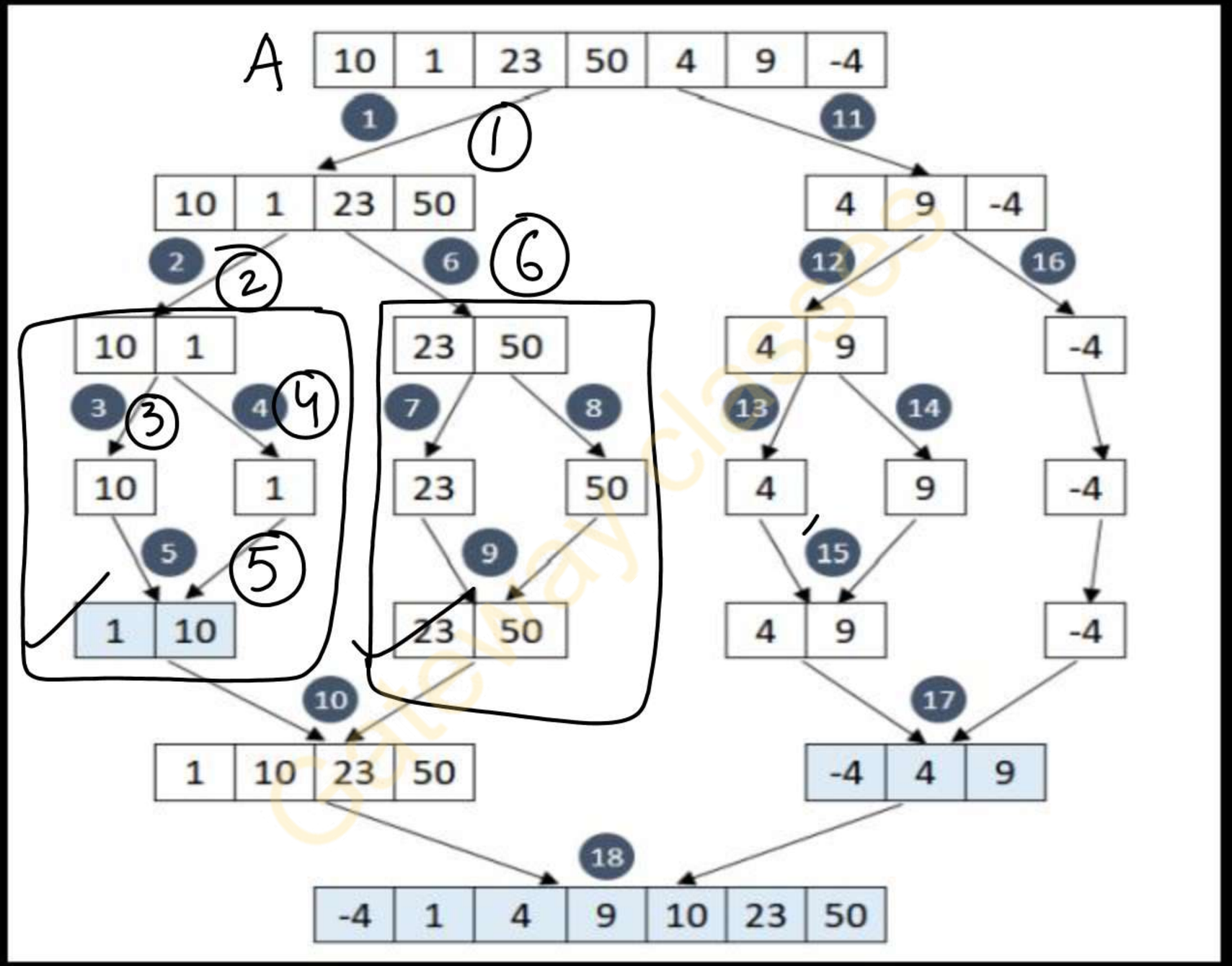
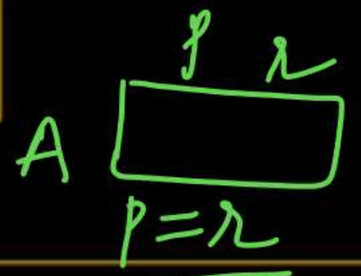


Figure 2: Merge Sort Combine Phase


A
10 1 23 50 4 9 -4
1
11
10 1 23 50
4 9 -4
2
6
12
16
10 1
23 50
4 9
-4
3
4
7
8
13
14
10
1
23
50
4
9
-4
5
9
15
1 10
23 50
4 9
-4
10
17
1 10 23 50
-4 4 9
18
-4 1 4 9 10 23 50

Merge Sort AKTU 2021-22, 2022-23,2023-24



Array **A[p.. r]** starts at index **p** and ends at index **r**.

Divide: **q** as central point between **p** and **r**, we divide array into two arrays A[p..q] and A[q+1, r].

Conquer:

Individually sort subarrays **A[p..q]** and **A[q+1, r]** recursively.

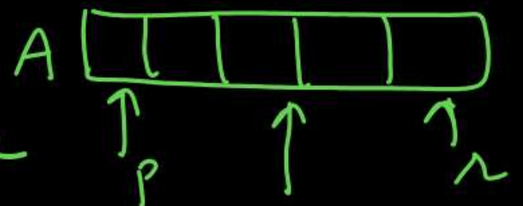
Combine: **A[p..q]** and **A[q+1, r]** are merged to form a new sorted array **[p..r]**.

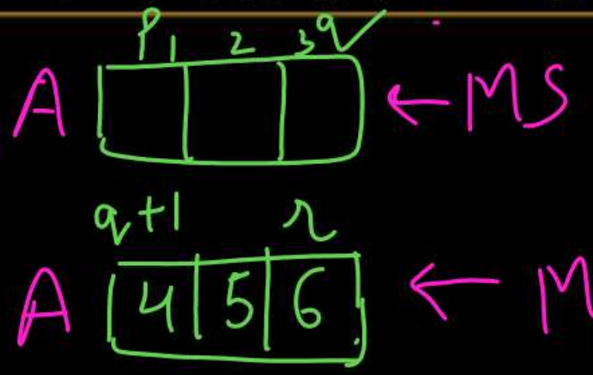
$q = \frac{1+6}{2} = \left\lfloor \frac{7}{2} \right\rfloor$

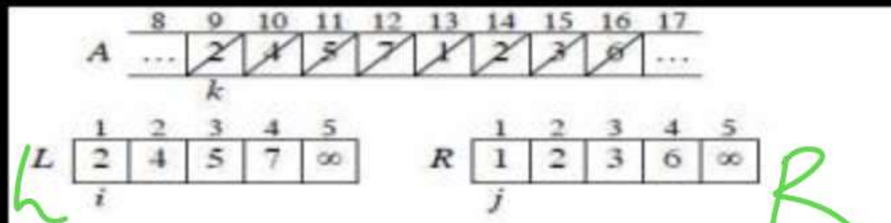
$q = 3$

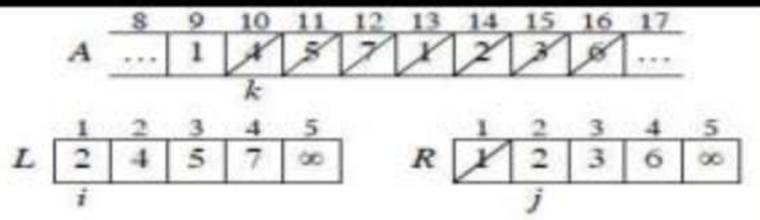
MERGE SORT(A,p,r)

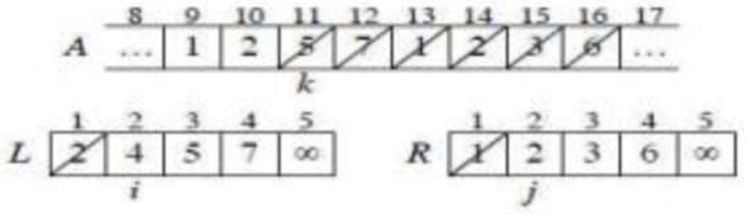
1. If p<r // Base case
2. Then q ← floor[(p+ r)/2]
3. MERGE-SORT (A, p, q) //Sort first half
4. MERGE-SORT (A, q+1,r) //Sort second half
5. MERGE (A, p, q, r) //Merge two sorted halves

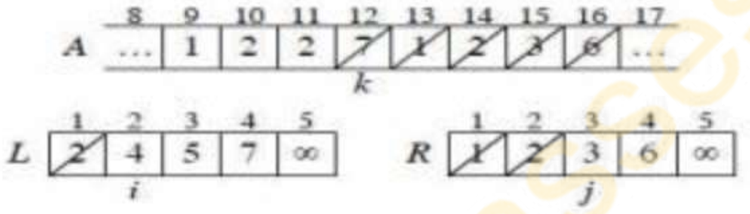


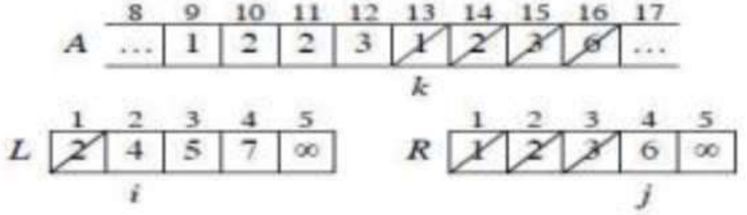


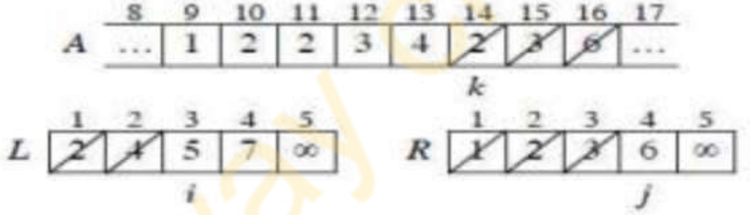


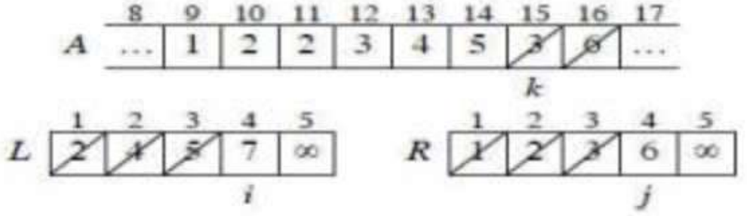


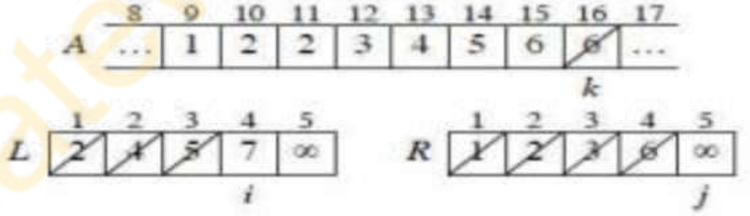


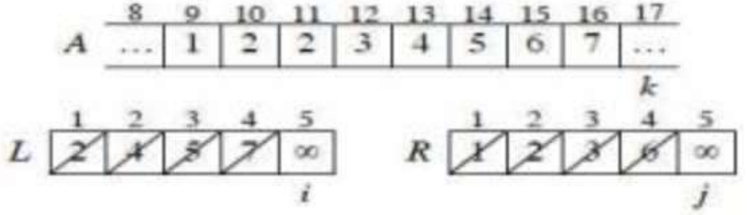






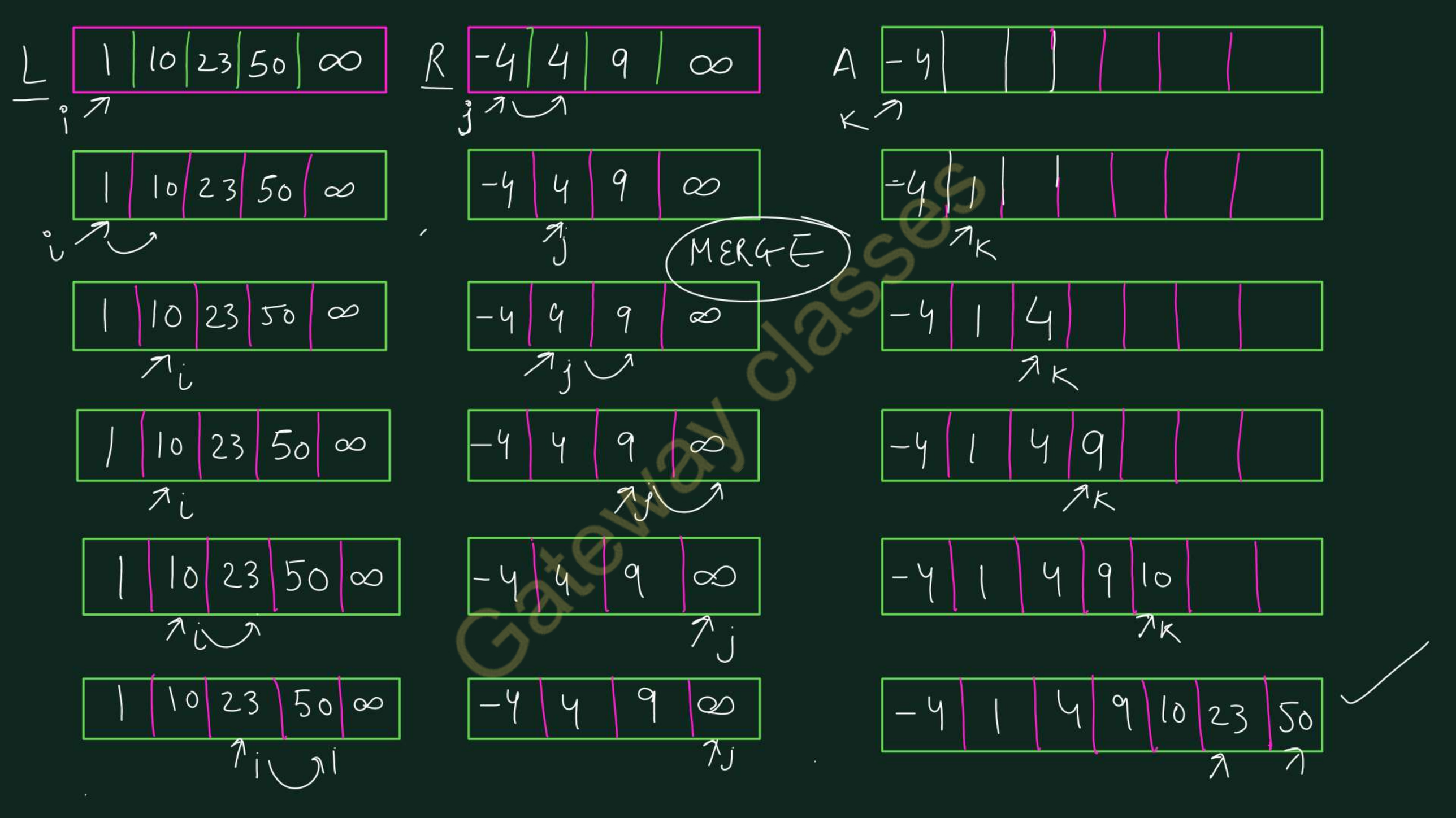







```
MERGE (A, p, q, r)

n1 = q-p+1                                   — O(1)
n2= r-q                                      — O(1)
create arrays L[1.....n1 + 1] and
                    R [ 1.....n2 +1 ]        — O(1)
for i ← 1 to n1
        do L[i] ← A [ p+ i-1]                ] O(n)
for j ← 1 to n2
        do R[j] ← A[ q + j]                  ] O(n)
L [n1+ 1] ← ∞                                —
R[n2+ 1] ← ∞                                 —
i ← 1                                        —
j ← 1                                        —
For k ← p to r                               O(n)
        do if L [i] ≤ R[j]
for Merge open  then A[k] ← L[ i]
                     i ← i +1
T(n)= O(n)           else A[k] ← R[j]
                     j ← j+1
```


L
1 10 23 50 ∞
i
R
-4 4 9 ∞
j
A
-4
K
MERGE
Gateway classes

Merge Sort Time Complexity Analysis

Let T (n) be the total time taken by the Merge Sort algorithm.

•Sorting two halves will take at the most 2T(n/2) time. Merge procedure takes O(n) time.

```
MERGE SORT (A, p, r)        ——— T(n)
1. If p<r                   ——— constant
2. Then q → ( p+ r)/2       ——— constant
3. MERGE-SORT (A, p, q)     ——— T(n1) = T(n/2)
4. MERGE-SORT ( A, q+1,r)   ——— T(n2) = T(n/2)
5. MERGE ( A, p, q, r)      ——— O(n)
```

$$T(n) = T(n_1) + T(n_2) + O(n)$$

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

Time complexity:

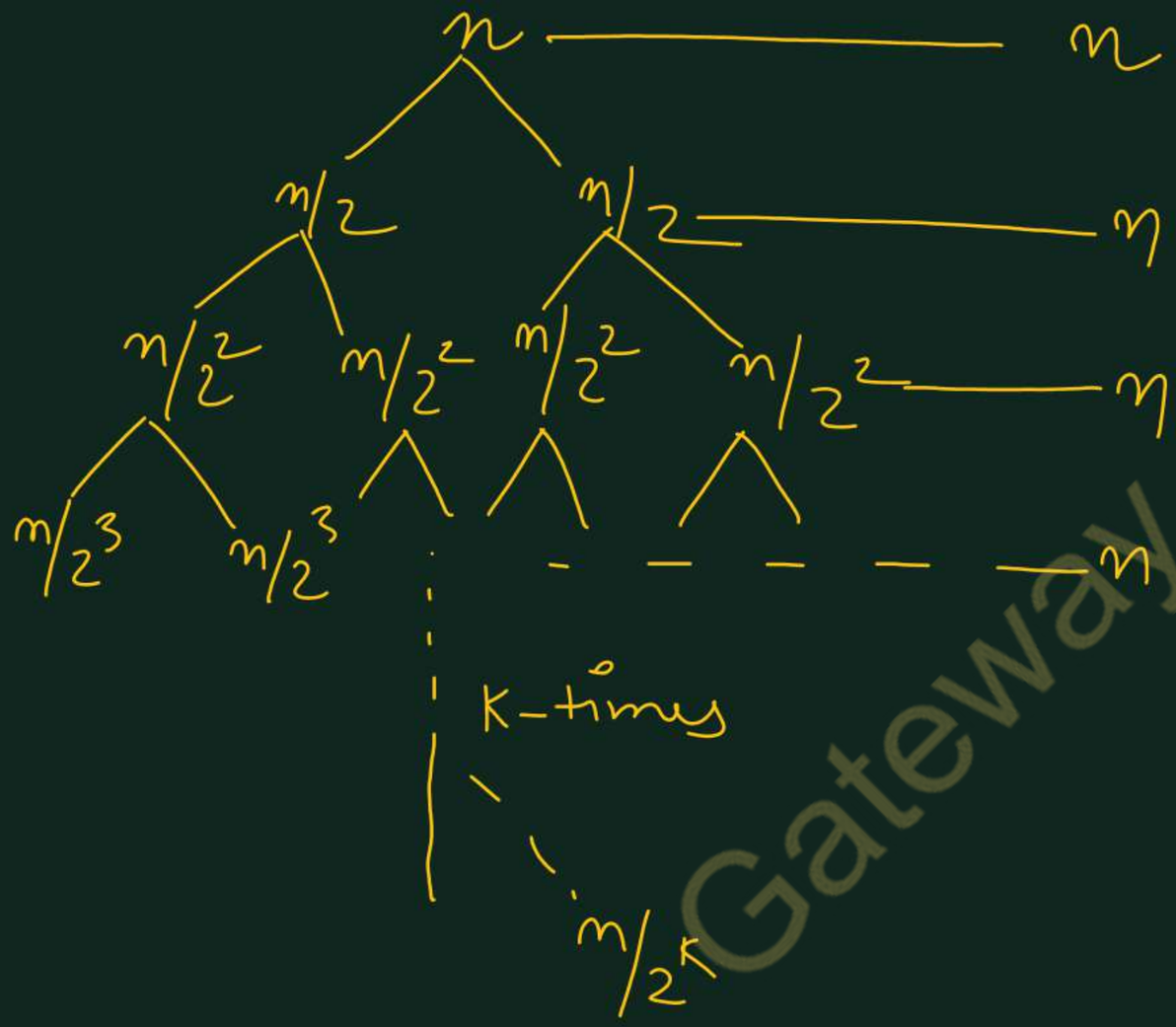
Best case: O(n log n)

Worst case: O(n log n)

Average case: O(n log n)

Space complexity: O(n)

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



put $\frac{n}{2^K} = 1$

$\log n = K \log 2$

$K = \log_2 n$

No. of levels $= K+1$

cost at each level $= n$

$T(n) = n \times (K+1)$

$= n \times (\log_2 n + 1)$

$T(n) = n \log_2 n$

Advantages of Merge Sort:

- Stable
- Guaranteed worst-case performance even on large datasets.
- Simple to implement.
- suitable for parallel processing. as we independently merge subarrays.

Disadvantages of Merge Sort:

- Space complexity: Merge sort requires additional memory to store the merged sub-arrays during the sorting process.
- Not in-place as it requires additional memory to store the sorted data.
- Slower than Quick Sort because quick sort works in-place.

AKTU PYQs

1. Write algorithm for merge sort and prove its worst case time complexity. (AKTU 2023-24)
2. Explain Merge sort algorithm and sort the following sequence {23, 11, 5, 15, 68,31, 4,17} using merge sort. (AKTU 2021-22, 2022-23)

AKTU

B.Tech 5th Sem

GATEWAY CLASSES

CS IT & CS Allied

DAA : Design & Analysis Of Algorithm

UNIT-1: Introduction

Lecture-10

Today's Target

- Quick Sort
- AKTY PYQs

By Dr. Nidhi Parashar Ma'am

- M.Tech Gold Medalist
- Net Qualified

Quick Sort AKTU 2021-22, 2022-23,2023-24

To sort Array **A[p.. r]**

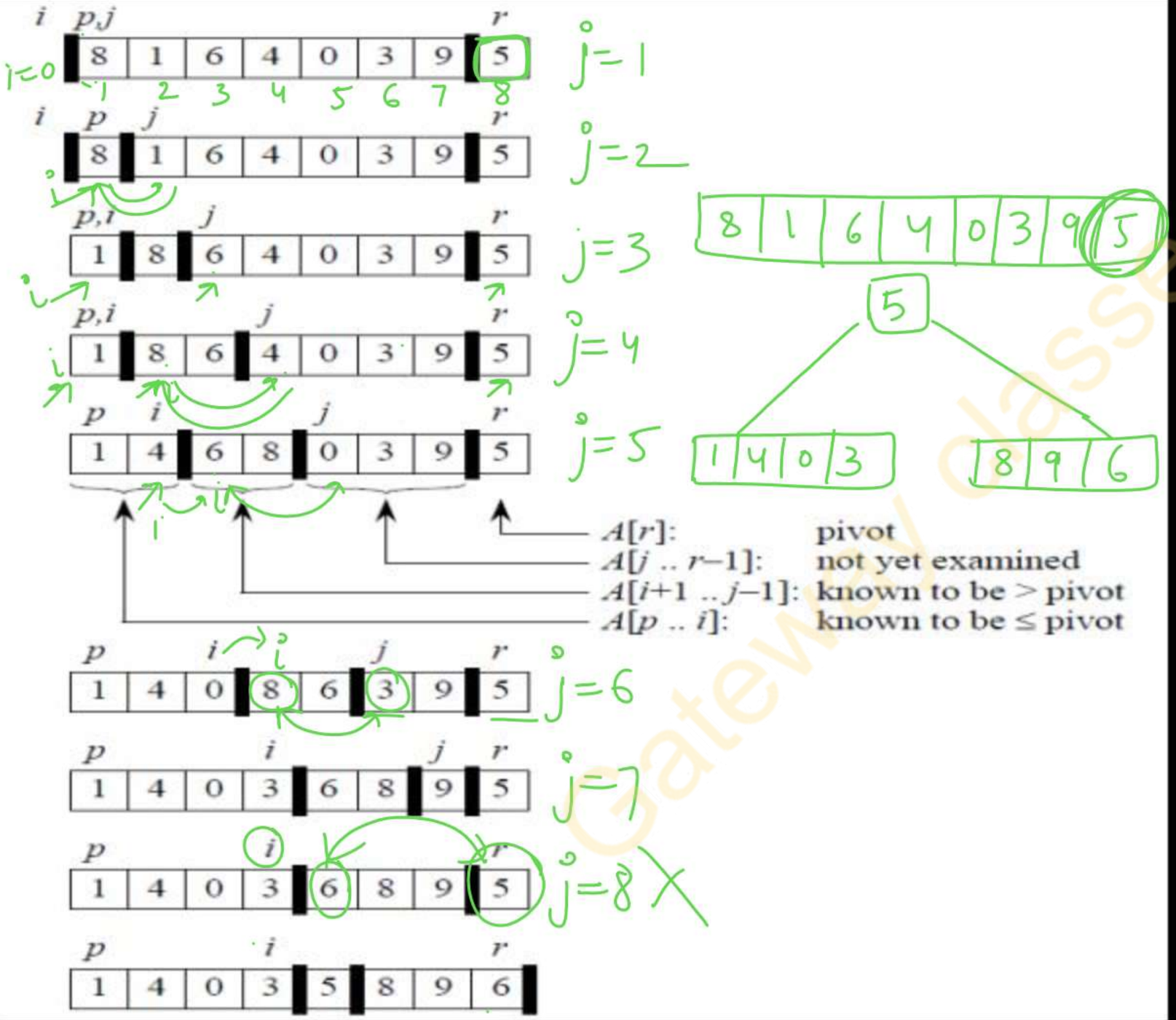
Divide: Partition **A[p.. r]**, into two (possibly empty) subarrays **A[p.. q-1]** and **A[q+1.. r]**, such that

- Each element in the first subarray **A[p.. q-1]** is <= **A[q] ,and**
- **A[q] is** <= each element in the second subarray **A[q+1.. r]**

Conquer: Sort two subarrays by recursive calls to QUICKSORT

Combine: No work is needed to combine the subarrays, because they are sorted in place.

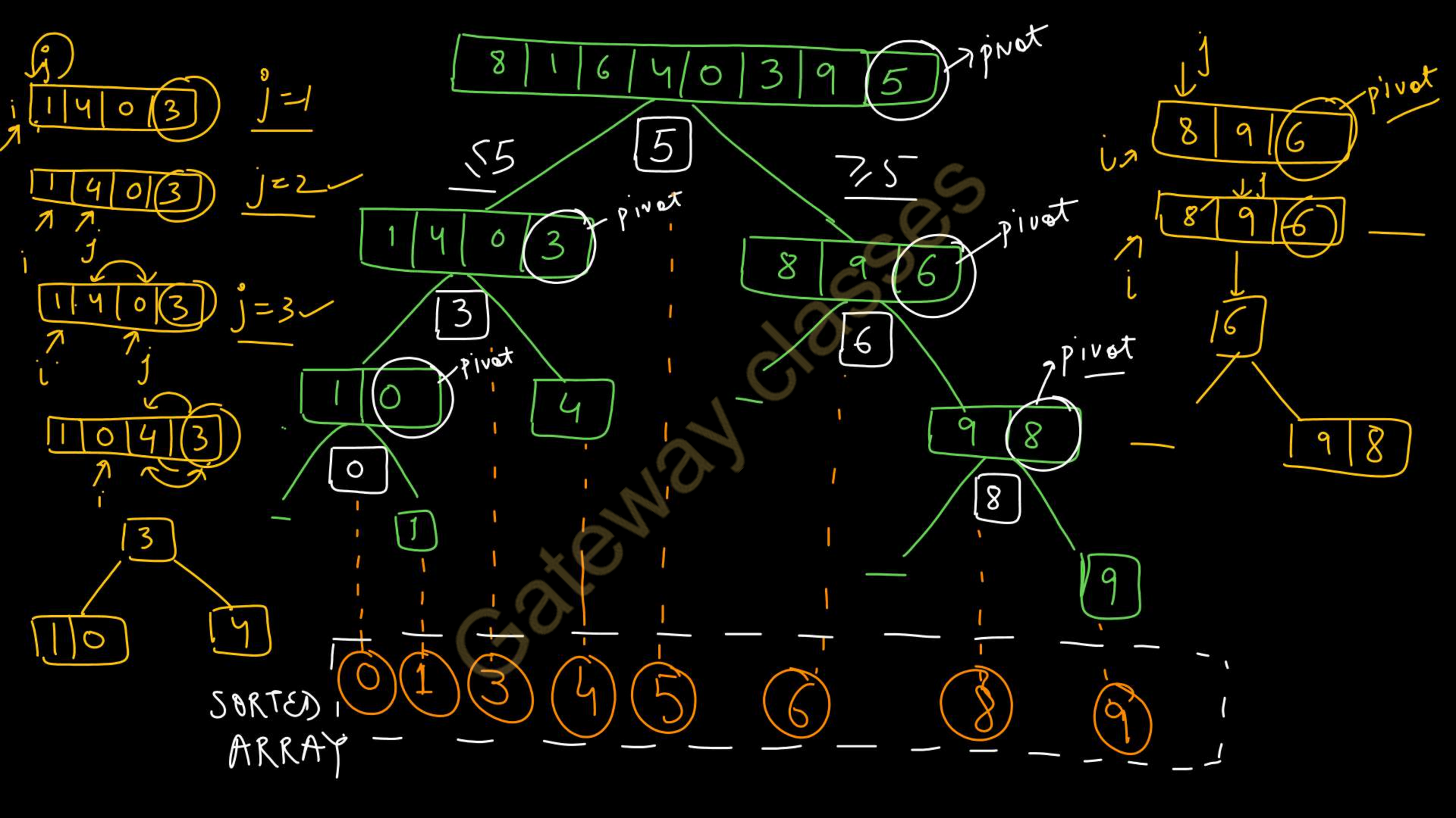
```
QUICK SORT(A,p,r)
1. If p<r
2. Then q ← PARTITION(A,p,r )
3. QUICK SORT (A, p, q-1)
4. QUICK SORT ( A, q+1,r)
```



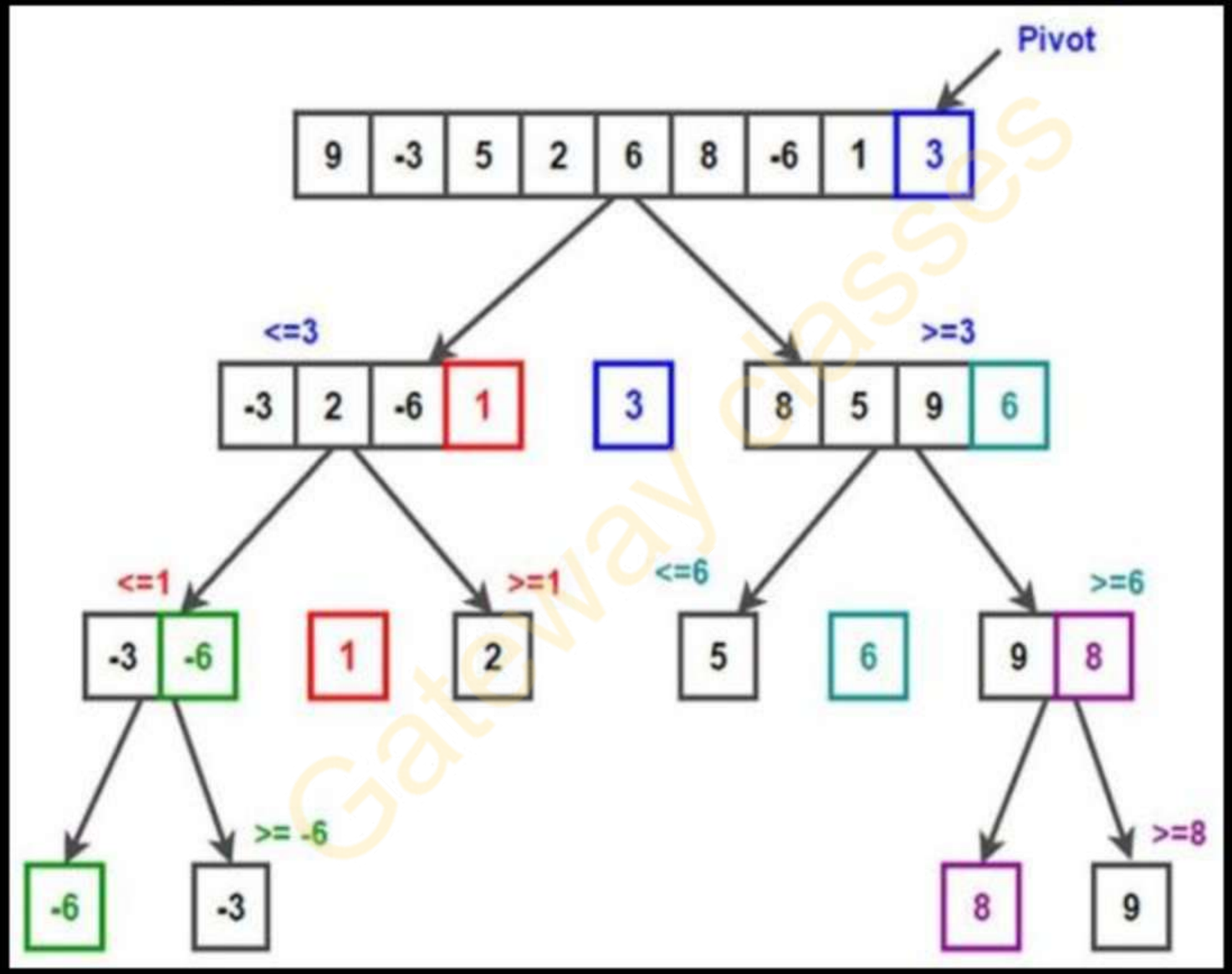

```
PARTITION(A, p, r)
{
x=A[r]
i=p-1
for j=p to r-1
{ if A[j] <=x
   {
        i=i+1
        Exchange A[i] with A[j]
   }
  }
Exchange A[i+1] with A[r]
return (i+1)
}
```

pivot element

(1 — 7)


8 1 6 4 0 3 9 5
pivot
5
<5
>5
1 4 0 3
pivot
3
1 0
pivot
0
1
4
8 9 6
pivot
6
9 8
pivot
8
9
1 4 0 3
j=1
j=2
j=3
1 0 4 3
3
1 0
4
8 9 6
pivot
8 9 6
6
9 8
SORTED ARRAY
0 1 3 4 5 6 8 9
Gateway classes

Quick Sort working



Partition part Time Complexity

```
PARTITION(A, p, r)
{
x=A[r]                                  O(1)
i=p-1                                   O(1)
for j=p to r-1                          O(n)
{  if A[j] <=x
   {
        i=i+1
        Exchange A[i] with A[j]
   }
}
Exchange A[i+1] with A[r]               O(1)
return (i+1)                            O(1)
}
```

Time Complexity = O(n)

Quick Sort Time and space Complexity Analysis

```
QUICK SORT(A,p,r)                          ——— T(n)
1. If p<r                                  ——— O(1)
2. Then q ← PARTITION(A,p,r )              ——— O(n)
3.  QUICK SORT (A, p, q-1)                 ——— T(n1)
4.  QUICK SORT ( A, q+1,r)                 ——— T(n2)
```

Time complexity:

Best case: O(n log n)

Worst case: $O(n^2)$

Average case: O(n log n)

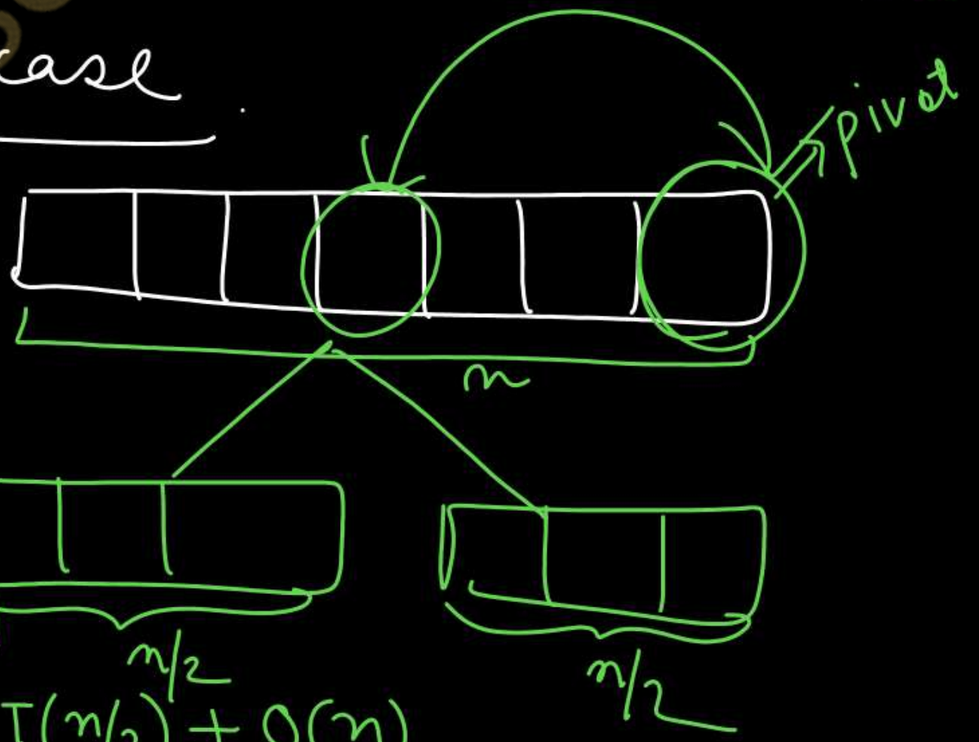
Space complexity:

O(log n)(best case)

O(n)(worst case)

$T(n) = T(n_1) + T(n_2) + O(n) + O(1)$

Best case



$T(n_1) = T(n/2)$

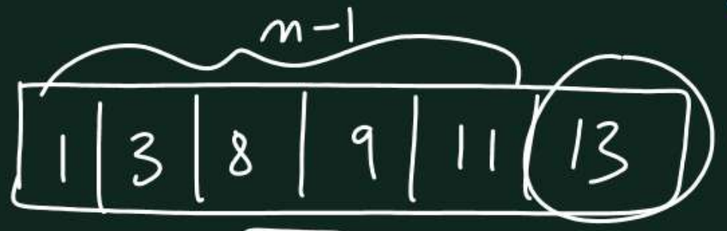
$T(n_2) = T(n/2)$

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

$T(n) = n \log_2 n$

Average case – $T(n) = O(n \log n)$

Imp) Worst case – Array is already sorted.



13

(n-1) elements

$T(n_1) = T(n-1)$

$T(n_2) = T(0)$

$T(n) = T(n_1) + T(n_2) + O(n)$

$T(n) = T(n-1) + 0 + n$

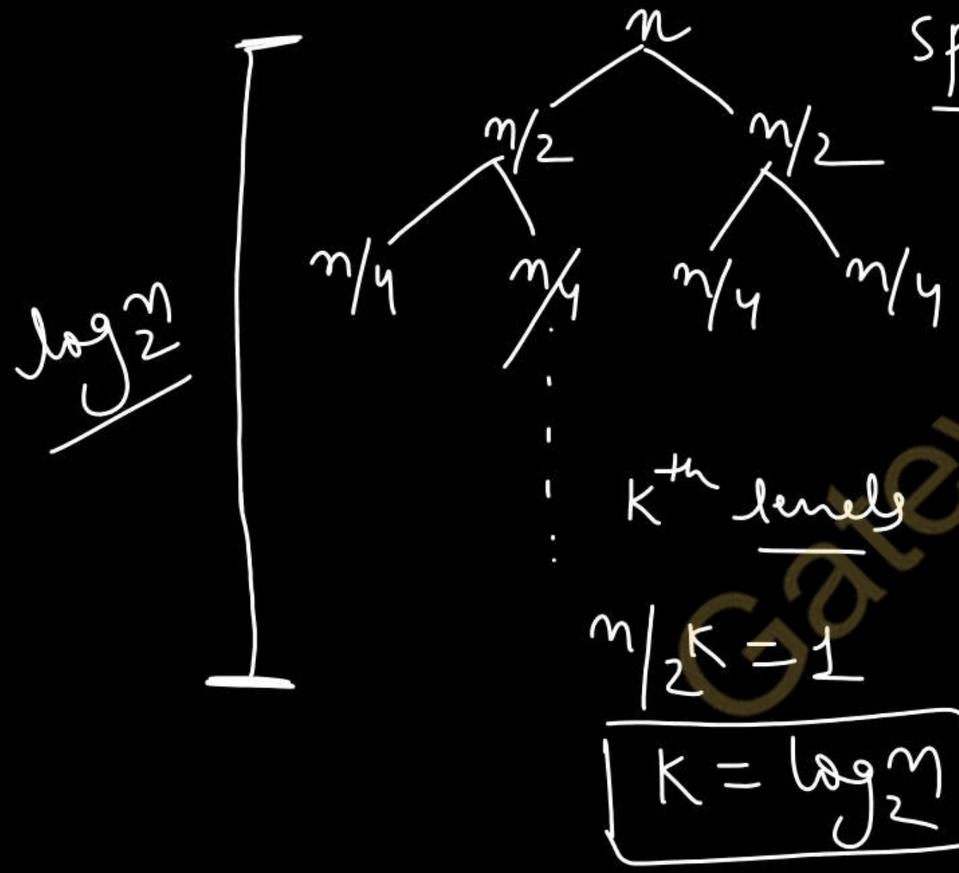
$T(n) = T(n-1) + n$

✓ $T(n) = O(n^2)$

Quick Sort Time and space Complexity Analysis

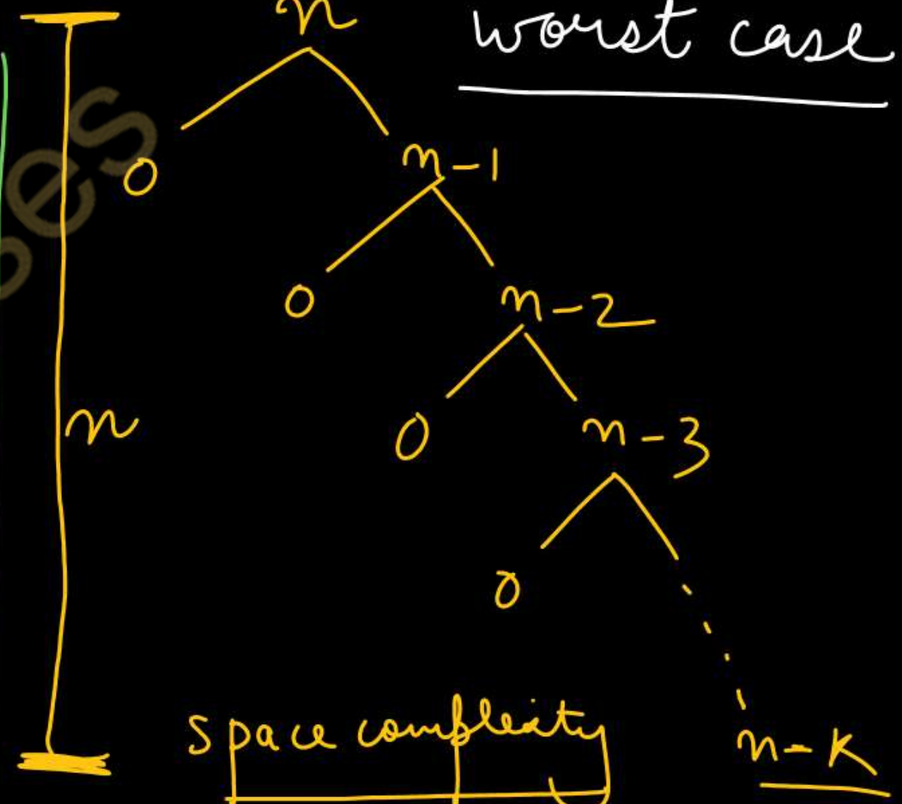
Space Complexity

Best case



Space complexity

$O(\log_2 n)$



worst case

Space complexity

$O(n)$

- **Advantages of Quick Sort:**
- Divide-and-conquer approach makes it easier to solve problems.
- Unlike merge sort, it is Cache Friendly as we work on the same array to sort and do not copy data to any auxiliary array(In-Place algo).
- Fastest general purpose algorithm for large data when stability is not required.

Disadvantages of Quick Sort:

- It has a worst-case time complexity of $O(n^2)$, which occurs when the pivot is chosen poorly.
- Not a good choice for small data sets.
- It is not a stable sort
- **In case of already sorted array, if last/first element is chosen as pivot then time complexity is $O(n^2)$.To address this randomized Quick sort is developed**

Randomized Quick Sort

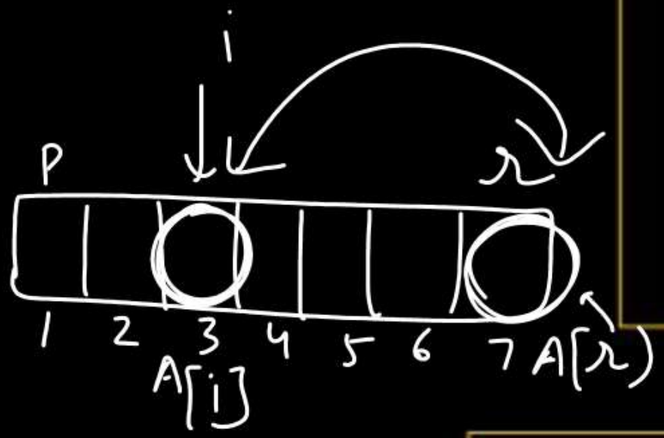
Randomized quick sort is designed to decrease worst case time complexity of quick sort.

Idea:

- Don't always use A[r] as the pivot. Instead, randomly pick an element from the subarray that is being sorted.
- Randomly selecting the pivot element will, on average will split the input array reasonably well balanced.

last / first element of array

Randomized Quick Sort




```
RANDOMIZED-PARTITION (A. p. r)
    i = RANDOM(p , r)
    exchange A[r] with A[i]
    return PARTITION(A. p. r)
```

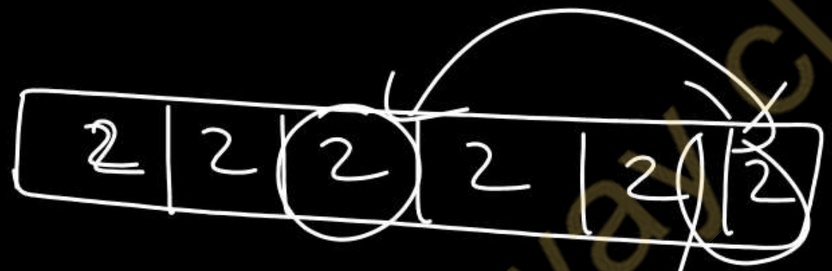
Avg Time Complexity
$O(n \log_2 n)$

```
RANDOMIZED-QUICKSORT (A. p . r)
if p<r
    q = RANDOMIZED-PARTITION(A. p. r)
    RANDOMIZED-QUICKSORT (A, p, q-1)
    RANDOMIZED-QUICKSORT (A, q + 1,r)
```

Time Complexity of Randomized Quick Sort

Worst case Time Complexity

When all elements in the array are same



$T(n) = O(n^2)$

AKTU PYQs

1. Explain and compare best and worst time complexity of Quick Sort.(AKTU 2022-23)
2. Explain Randomized algorithms. (AKTU 2022-23)
3. Among merge sort, insertion sort and quick sort which sorting algorithm is best in worst case. Sort the list E,X,A,M,P,L,E in alphabetical order .(AKTU 2019-20)

AKTU

B.Tech 5th Sem

GATEWAY CLASSES

CS IT & CS Allied

DAA : Design & Analysis Of Algorithm

UNIT-1: Introduction

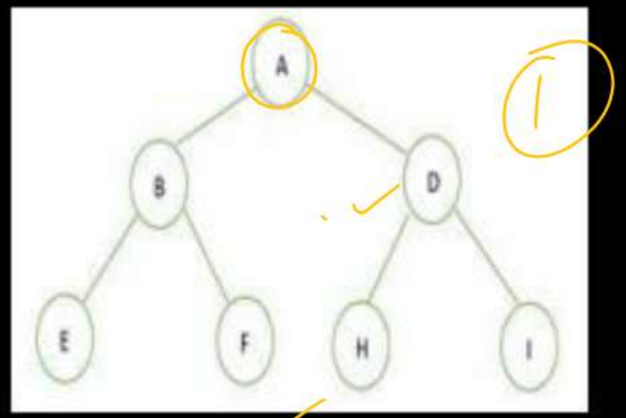
Lecture-11

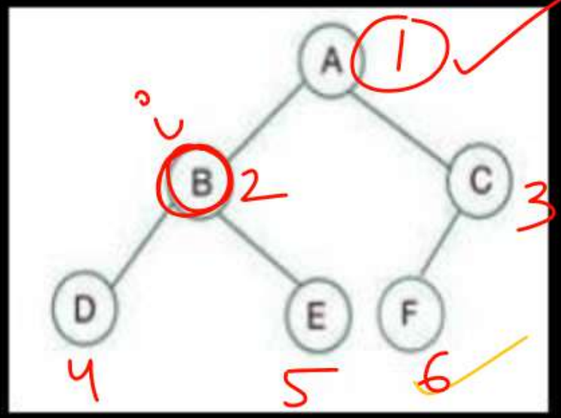
Today's Target

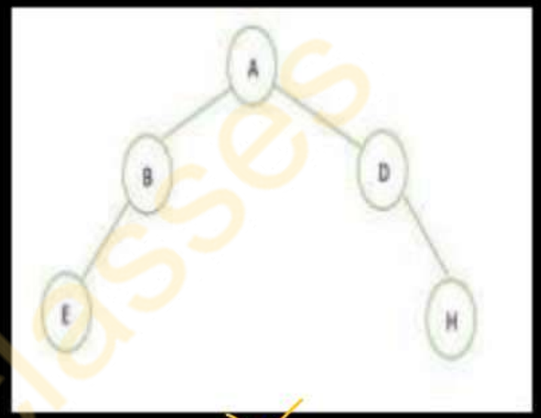
- Heap Sort
- AKTY PYQs

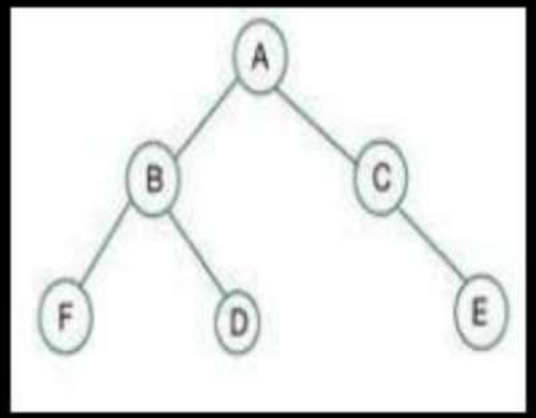
By Dr. Nidhi Parashar Ma'am
- M.Tech Gold Medalist
- Net Qualified

Complete Binary Tree: A tree in which except for the last level all the other level has maximum nodes and the nodes are lined up from left to right side.









1	2	3	4	5	6	7
A	B	D	E	F	H	I

Array

1	2	3	4	5	6
A	B	C	D	E	F

$i=2$ parent $\left\lfloor \frac{1}{2} \right\rfloor = 1$

A	B	D	E	–	–	H

A	B	C	F	D	E

Using arrays, we find parents/children as follows

(Assuming we start indexing at 1):

If parent is it index: floor(**i/2**)

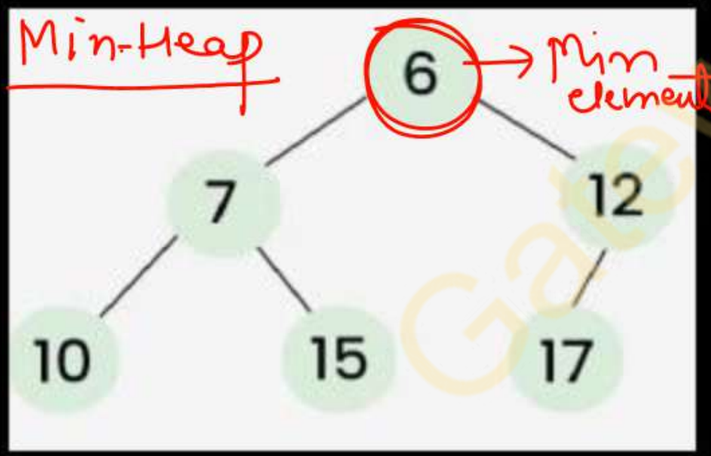
left child is at : **2*i**

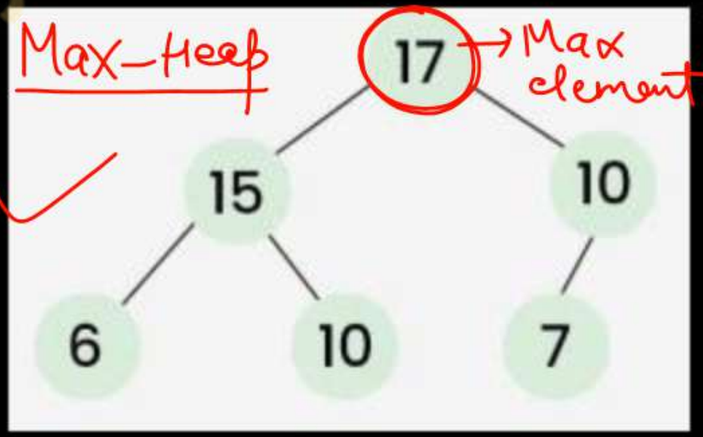
right child is at: **2*i+1**

Binary Heap: It is a **complete Binary Tree** which is used to store data efficiently to get the max or min element based on its structure.

Min Heap: Key at the root must be minimum among all keys present in Heap. for all nodes i , excluding the root, A[PARENT(i) <=A[i]

Max Heap: Key at the root must be maximum among all keys present in Heap. for all nodes i , excluding the root, A[PARENT(i) >=A[i]





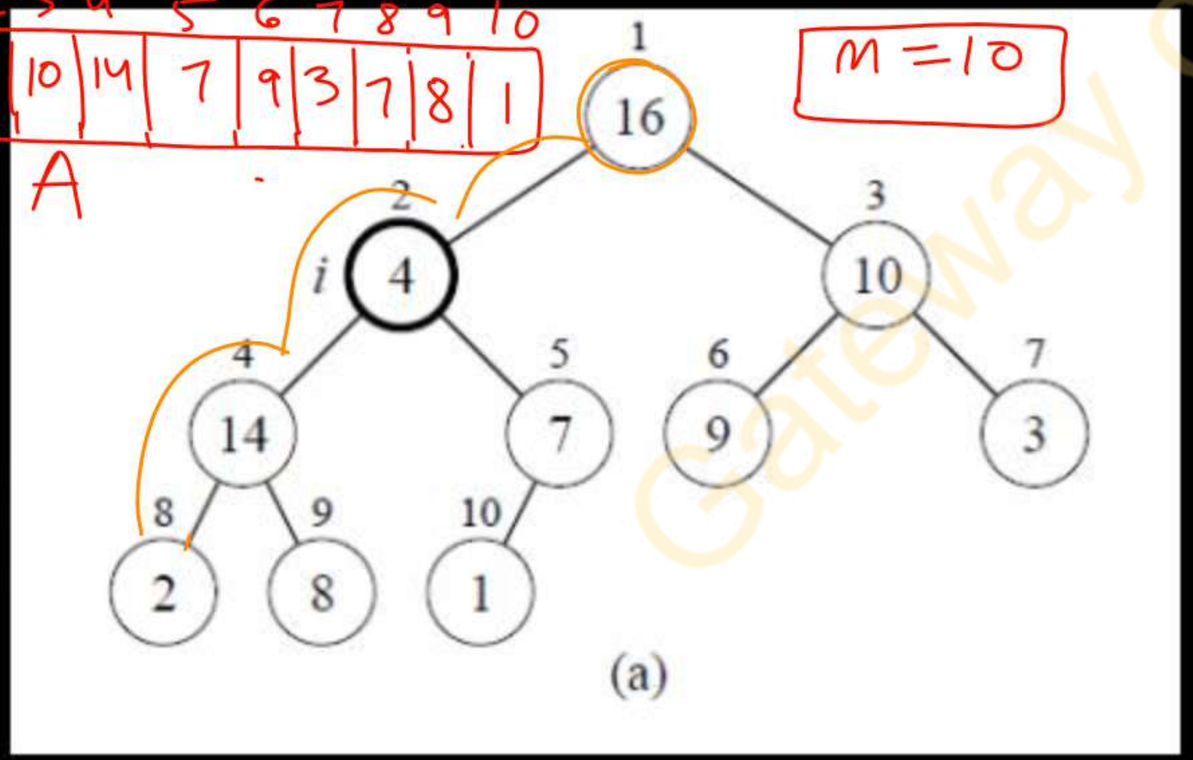
Maintaining the heap property

- MAX-HEAPIFY is used to maintain the max-heap property.
- Before MAX-HEAPIFY, A[i] may be smaller than its children.
- Assume left and right subtrees of i are max-heaps.
- After MAX-HEAPIFY, subtree rooted at i is a max-heap.

1	2	3	4	5	6	7	8	9	10
16	4	10	14	7	9	3	7	8	1

A

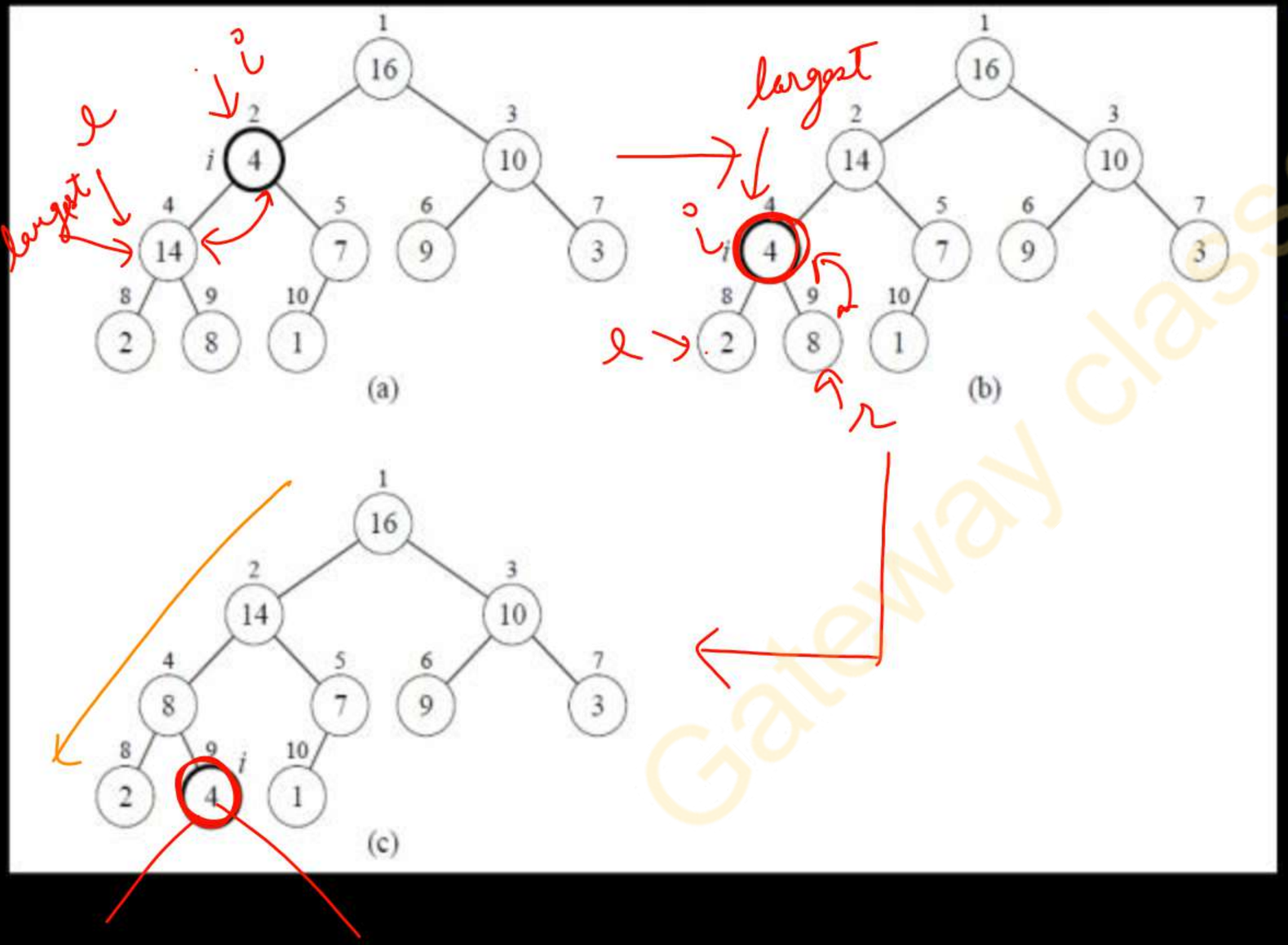
n = 10



(a)

```
MAX-HEAPIFY (A, i, n)
{ l=Left(i);
  r=Right(i);
  if(l<= n && A[l] > A[i])
        largest=l;
  else  largest=i;
  if(r<= n && A[r] > A[largest])
        largest=r;
  if(largest !=i)
  { Swap(A[i], A[largest])
  MAX-HEAPIFY(A, largest, n);
  }
}
```

Illustration of MAX-HEAPIFY



Time Complexity of MAX-HEAPIFY

Heap is almost-complete binary tree, so must process O(log n) levels, with constant work at each level (comparing 3 items and maybe swapping 2).

Time Complexity: O(log n)

Building a Max-heap from list of elements

```
BUILD-MAX-HEAP(A, n)
{
for i = floor(n/2) down to 1
        MAX-HEAPIFY(A, i, n)
}
```

i = 5, 4, 3, 2, 1

4	1	3	2	16	9	10	14	8	7

INPUT

MAX-HEAP

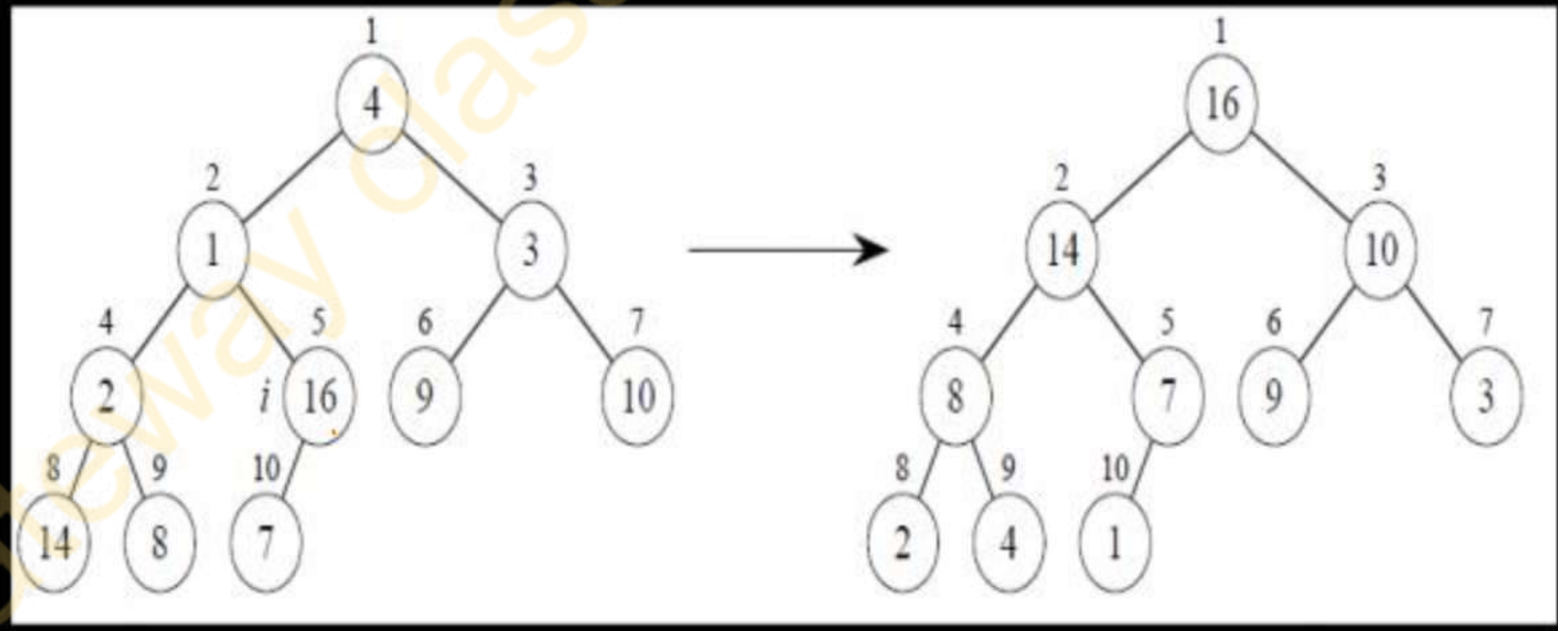
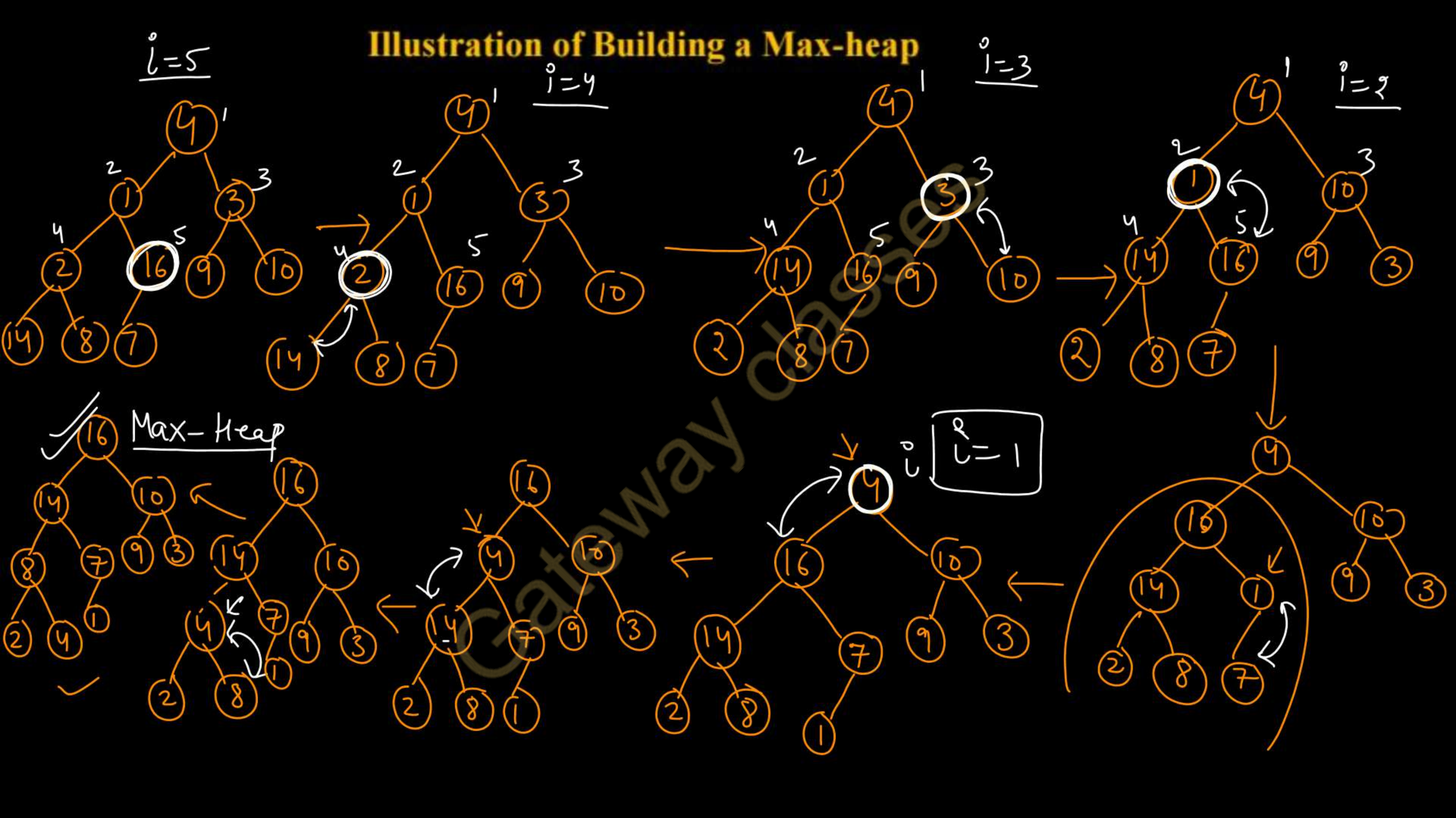
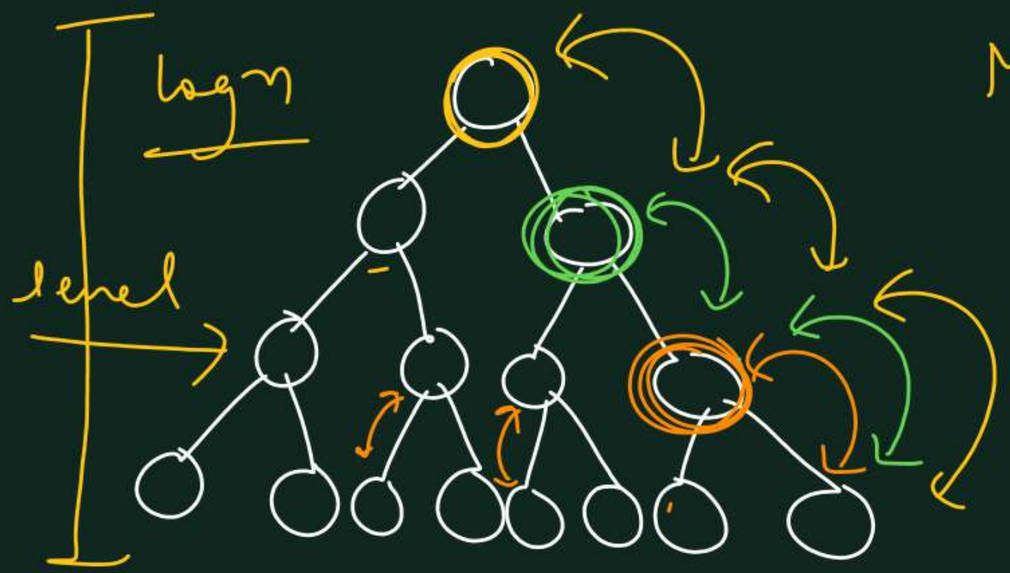


Illustration of Building a Max-heap



Time Complexity of Build-MaxHeap — $O(n)$



Max No. of nodes at height $h = \left\lceil \frac{n}{2^{h+1}} \right\rceil$

$n = 15$

$\frac{15}{2^{1+1}} = \frac{15}{4} = 3 \Rightarrow (4)$

work done at node on specific height $(h) = O(h)$

$$T(n) \Rightarrow \sum_{h=0}^{\log n} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot O(h) \Rightarrow T(n) = O(n)$$

Heap Sort

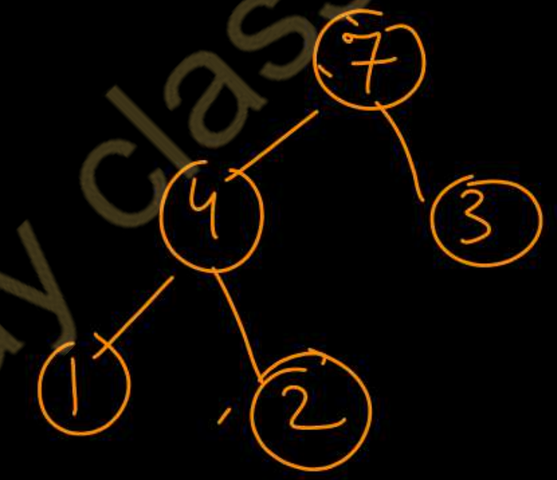
```
HEAP-SORT(A, n)
{
    Build-Max-Heap(A, n)
    for i <-- n down to 2
        exchange A[1] <---> A[i]
        heap-size[A] = heap-size[A] -1
        MAX-HEAPIFY (A,1, i-1)
}
```

n = 5, 4, 3, 2

always apply Maxheapify on root.

Example: Sort the given array using heap sort.

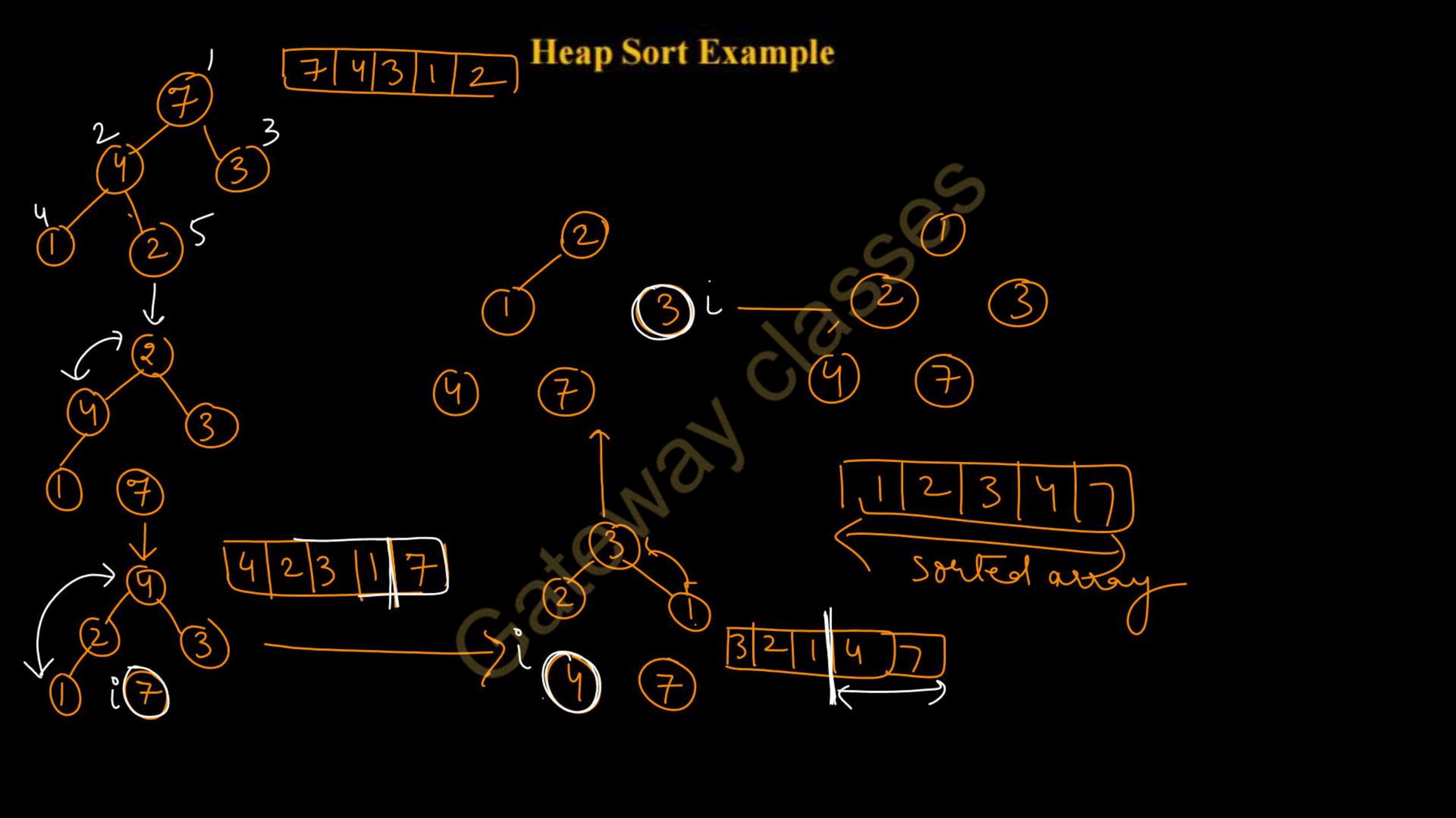
1	2	3	4	5
7	4	3	1	2



n = 5

- Advantages of Heap Sort: In-Place algo
- Disadvantages of Heap Sort: It is not a stable sort

Heap Sort Example



Heap sort Time Complexity

```
HEAP-SORT(A, n)
{
    Build-Max-Heap(A, n)
    for i <-- n down to 2
        exchange A[1] <---> A[i]
        heap-size[A] = heap-size[A] -1
        MAX-HEAPIFY (A,1, i-1)
}
```

Build-Max-Heap(A, n) → $O(n)$

for i <-- n down to 2 → $(n-1)$

MAX-HEAPIFY (A,1, i-1) → $O(\log n)$

$T(n) = O(n) + (n \times \log n)$

$T(n) = O(n \log n)$

Time complexity:

Best case: O(n log n)

Worst case: O(n log n)

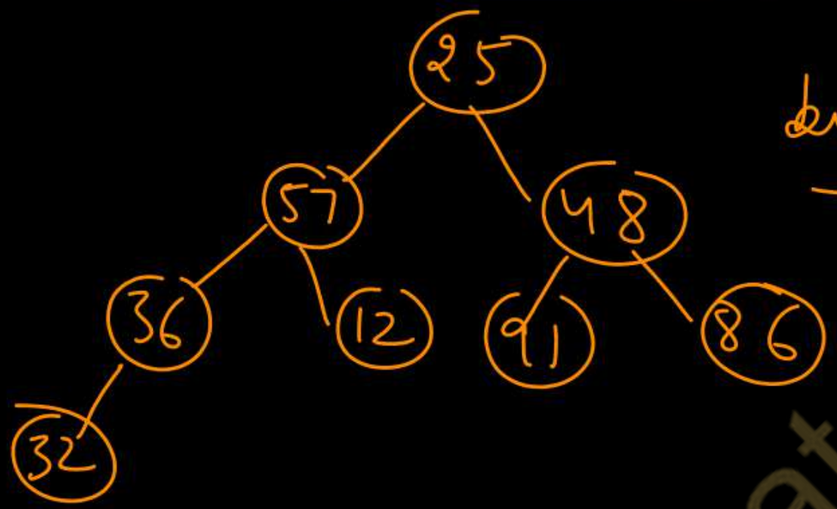
Average case: O(n log n)

Space complexity: heapsort is an in-place designed sorting algorithm so Therefore, we don't use any extra space, so

Space Complexity is: O(1)

AKTU PYQs

1. What is stable and unstable sorting? Sort the given list of elements{25,57,48,36,12,91,86,32} using heap sort.(AKTU 2021-22)



build - Max Heap

Then Apply Heap sort.

AKTU

B.Tech 5th Sem

GW GATEWAY CLASSES

CS IT & CS Allied

DAA : Design & Analysis Of Algorithm

UNIT-1: Introduction

Lecture-12

Today's Target

- Sorting in Linear Time: Counting Sort
- AKTY PYQs

By Dr. Nidhi Parashar Ma'am
- M.Tech Gold Medalist
- Net Qualified

Counting Sort

- Non-comparison based Sorting
- Efficient when the range of input values is small compared to the number of elements to be sorted.
- Depends on a key assumption: Numbers to be sorted are integers in {0…..k)
- Linear sorting algorithm with asymptotic complexity O(n + k)
- Time efficient and stable algorithm.
- It is not an in-place sorting algorithm as it requires extra additional space O(k).

- Idea: count the **frequency** of each distinct element in the input array and use that information to place the elements in their correct sorted positions.

Counting Sort

√ Imp Input final

```
COUNTING-SORT(A, B, n, k)
  let C[0..k] be a new array
  for i = 0 to k
      C[i] = 0
  for j = 1 to n
      C[A[j]] = C[A[j]] + 1
  for i = 1 to k
      C[i] = C[i] + C[i − 1]
  for j = n downto 1
      B[C[A[j]]] = A[j]
      C[A[j]] = C[A[j]] − 1
```

No of (A) elements

Max value in Input array

O(K)

freq. count loop

for updated C array

Example: Sort the given array using ~~heap~~ counting sort.

2	5	3	0	2	3	0	3

n = 8

K = 5

C[0...K] ⇒ Frequency Array

A[j] = 2

C[A[j]] = C[2]

Counting Sort Example AKTU-2023-24

Example: Sort the given array using counting sort.

A

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

$n = 8$
$K = 5$

C

0	1	2	3	4	5
2	2	4	7	7	8
1	3	6			7
0	2	5			
		4			

$C[A[j]] = C[A[j]] + 1$

$j = 0$

$A[j] = 2$

$C[A[j]] = C[2]$

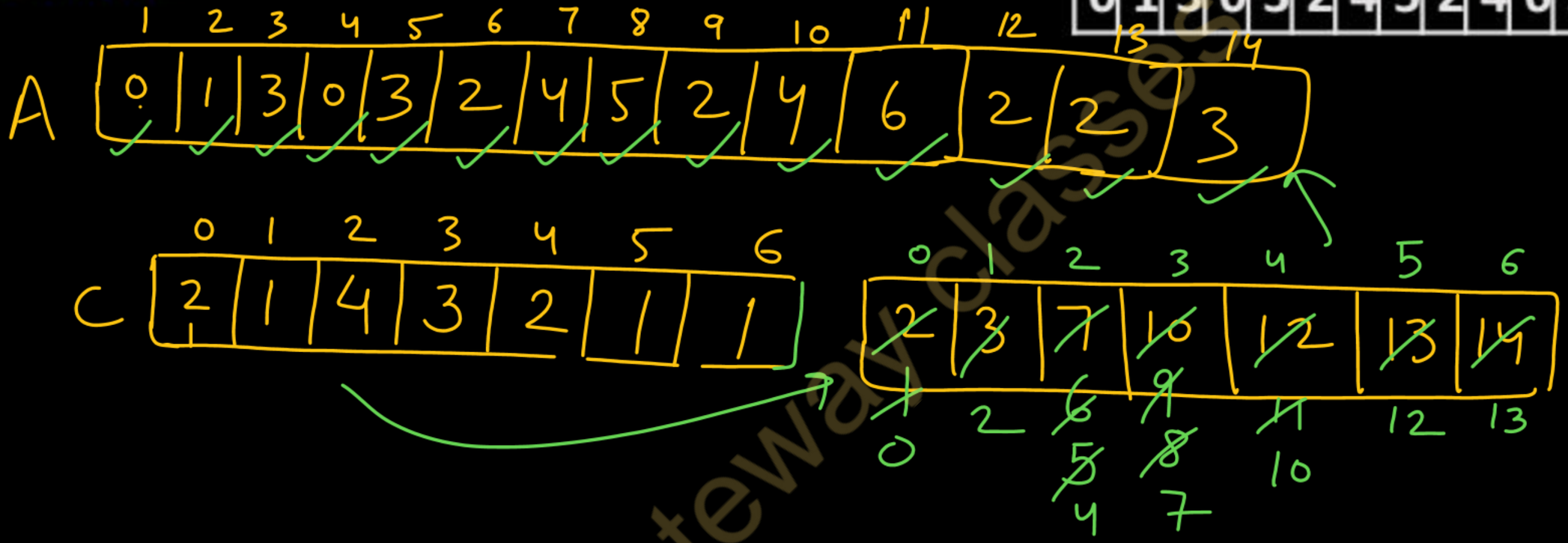
B

1	2	3	4	5	6	7	8
0	0	2	2	3	3	3	5

Counting Sort Example AKTU 2022-23

Example: Sort the given array using counting sort.

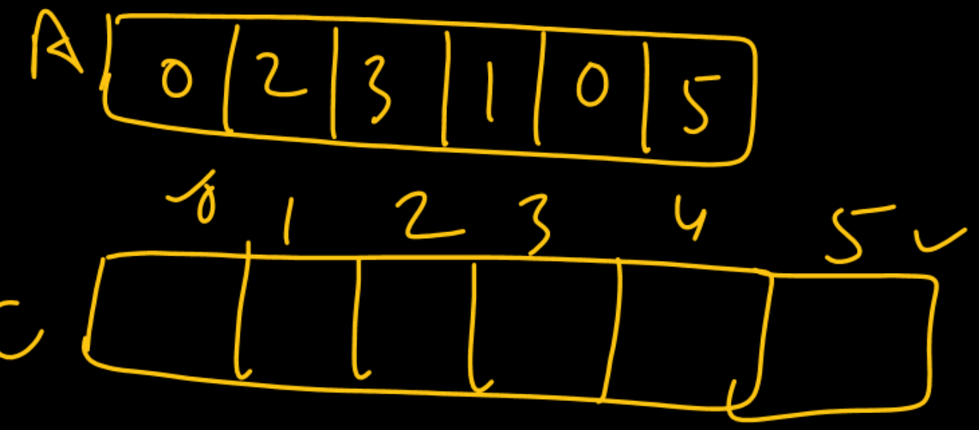
0	1	3	0	3	2	4	5	2	4	6	2	2	3



Disadvantages

- Counting Sort is inefficient if the range of key value (k) is very large.
- If the input array is already sorted then also it will require an additional array to store the sorted elements and still process it completely.
- Only sort discrete values like integer, as frequency range cannot be constructed for other values.

Because Array values are used as Indexes in intermediate steps.



AKTU PYQs

1. Write the algorithm for counting Sort? Sort the given list of elements{ 4,0,2,0,1,3,5,4,1,3,2,3} using counting sort. (AKTU 2020-21)
2. Write the algorithm for counting Sort? Sort the given list of elements{ 2,5,3,0,2,3,0,3} using counting sort. (AKTU 2023-24)
3. Write the algorithm for counting Sort? Sort the given list of elements{ 0,1,3,0,3,2,4,5,2,4,6,2,2,3} using counting sort. (AKTU 2022-23)

AKTU

B.Tech 5th Sem

GATEWAY CLASSES

CS IT & CS Allied

DAA : Design & Analysis Of Algorithm

UNIT-1: Introduction

Lecture-13

Today's Target

Sorting in Linear Time:

- Radix Sort
- Bucket Sort

Shell Sort

Comparison of sorting techniques

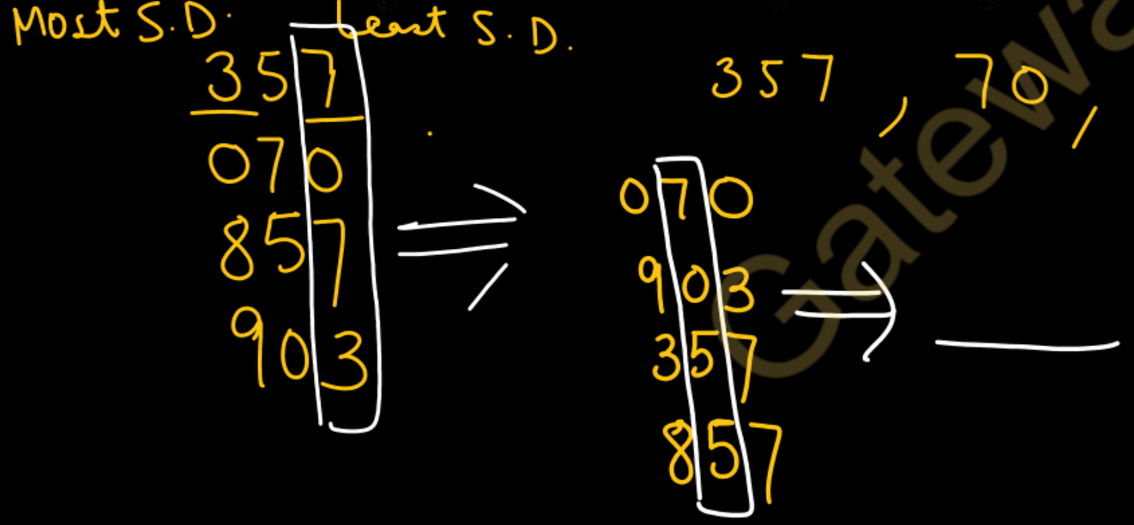
- AKTY PYQs

By Dr. Nidhi Parashar Ma'am

- M.Tech Gold Medalist
- Net Qualified

Radix Sort

- **Radix Sort** is a linear sorting algorithm
- Sorts elements by processing them digit by digit.
- Efficient sorting algorithm for integers. just like counting sort.
- Digit by digit sorting is performed that is started from the least significant digit to the most significant digit using any stable sorting algorithm (counting sort).
- It uses arrays to sort the input array so it is not an in-place sorting algorithm.



Radix Sort

To sort d digits:

RADIX-SORT(A, d)

for i = 1 **to** d

use a stable sort to sort

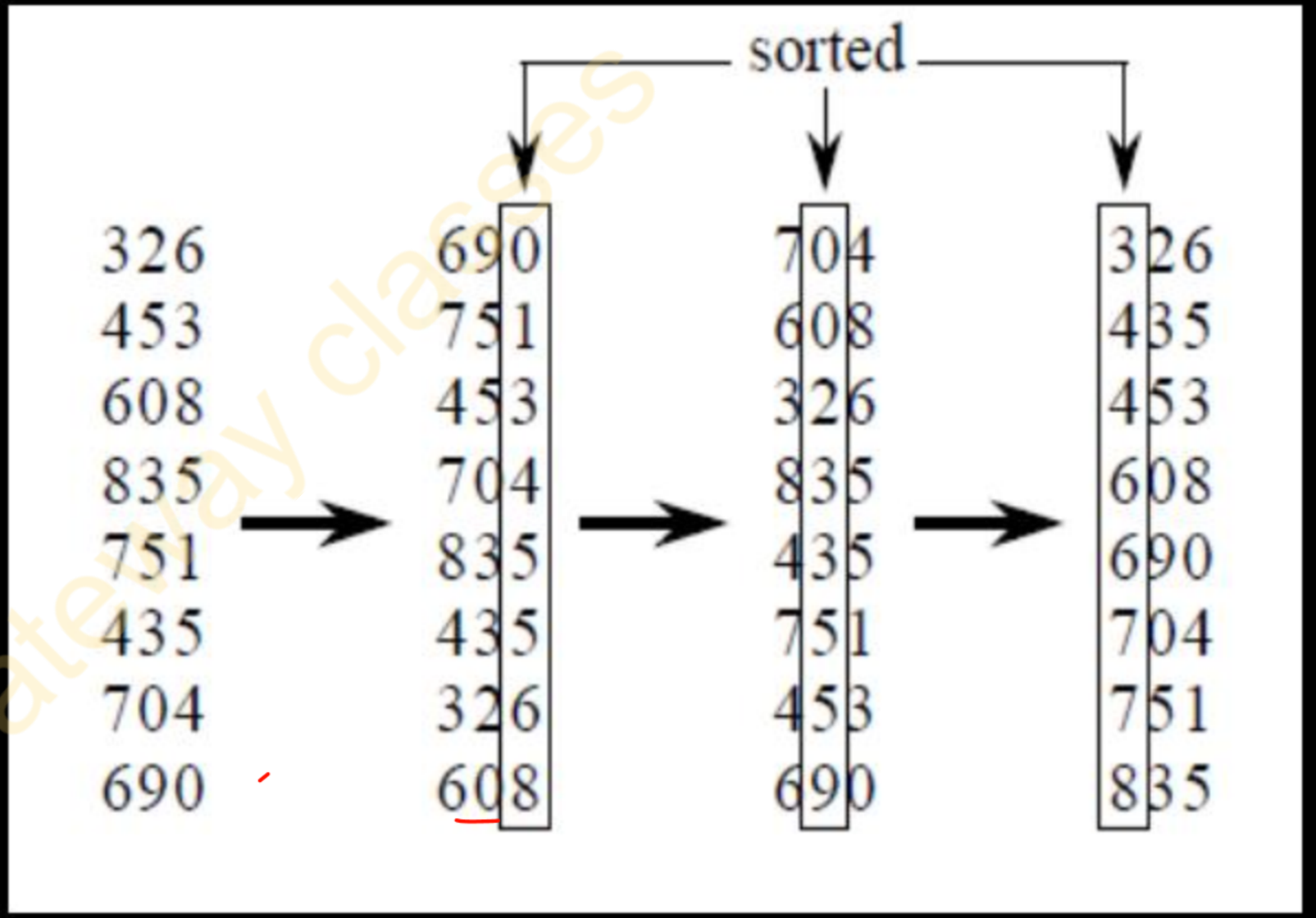
array A on digit i

If we use counting sort to sort digitwise!

$T(n) = O(d(n+K))$

if $(K \simeq n)$ then

$T(n) = O(n*d)$



Radix sort Time Complexity

- If radix sort uses counting sort as an intermediate stable sort, the time complexity is **O(d(n + k)).** If $k \leq n \Rightarrow T(n) = O(n * d)$
- Here, d is the number of digits and O(n + k) is the time complexity of counting sort.

Bucket Sort

- The basic procedure of performing the bucket sort is given as follows -
  - First, partition the range into a fixed number of buckets.
  - Then, toss every element into its appropriate bucket.
  - After that, sort each bucket individually by applying a sorting algorithm.
  - And at last, concatenate all the sorted buckets.

Bucket sort is commonly used -

- With floating-point values.
- When input is distributed uniformly over a range.

Bucket Sort

```
Bucket Sort(A[ ])
  1. Let B[0....n-1] be a new array
  2. n=length[A]
  3. for i=0 to n-1
  4.         make B[i] an empty list
  5. for i=1 to n
  6.         Insert A[i] into list B[n *A[i ]]
  7. for i=0 to n-1
  8.         Sort list B[i] with insertion-sort
  9. Concatenate lists B[0], B[1],....., B[n-1]  together in order
End
```

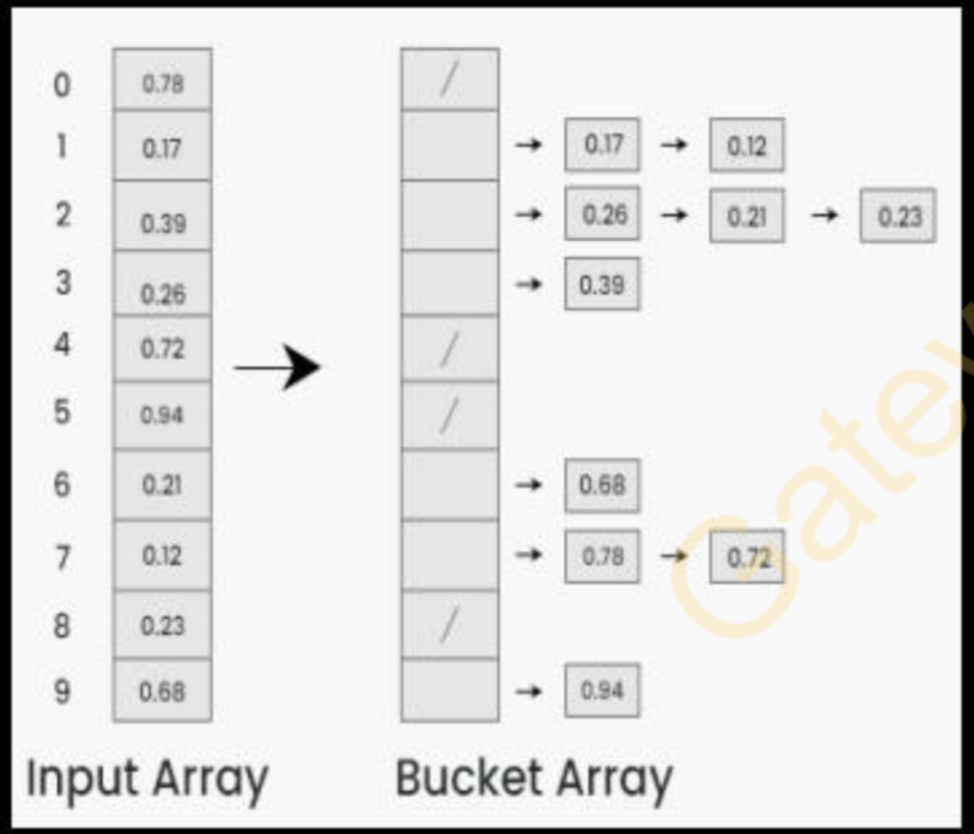
Array to be sorted.

Bucket Array

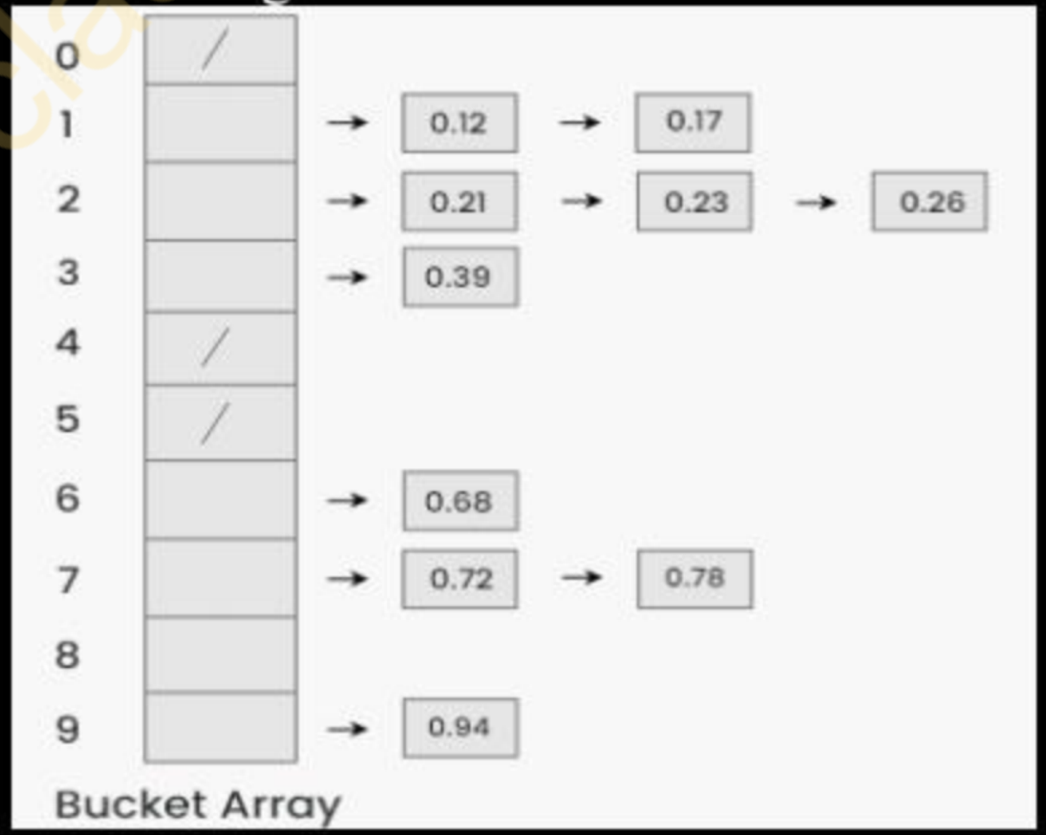
(n) → No. of elements in Input array

Step 1: Create an array of size 10, where each slot represents a bucket

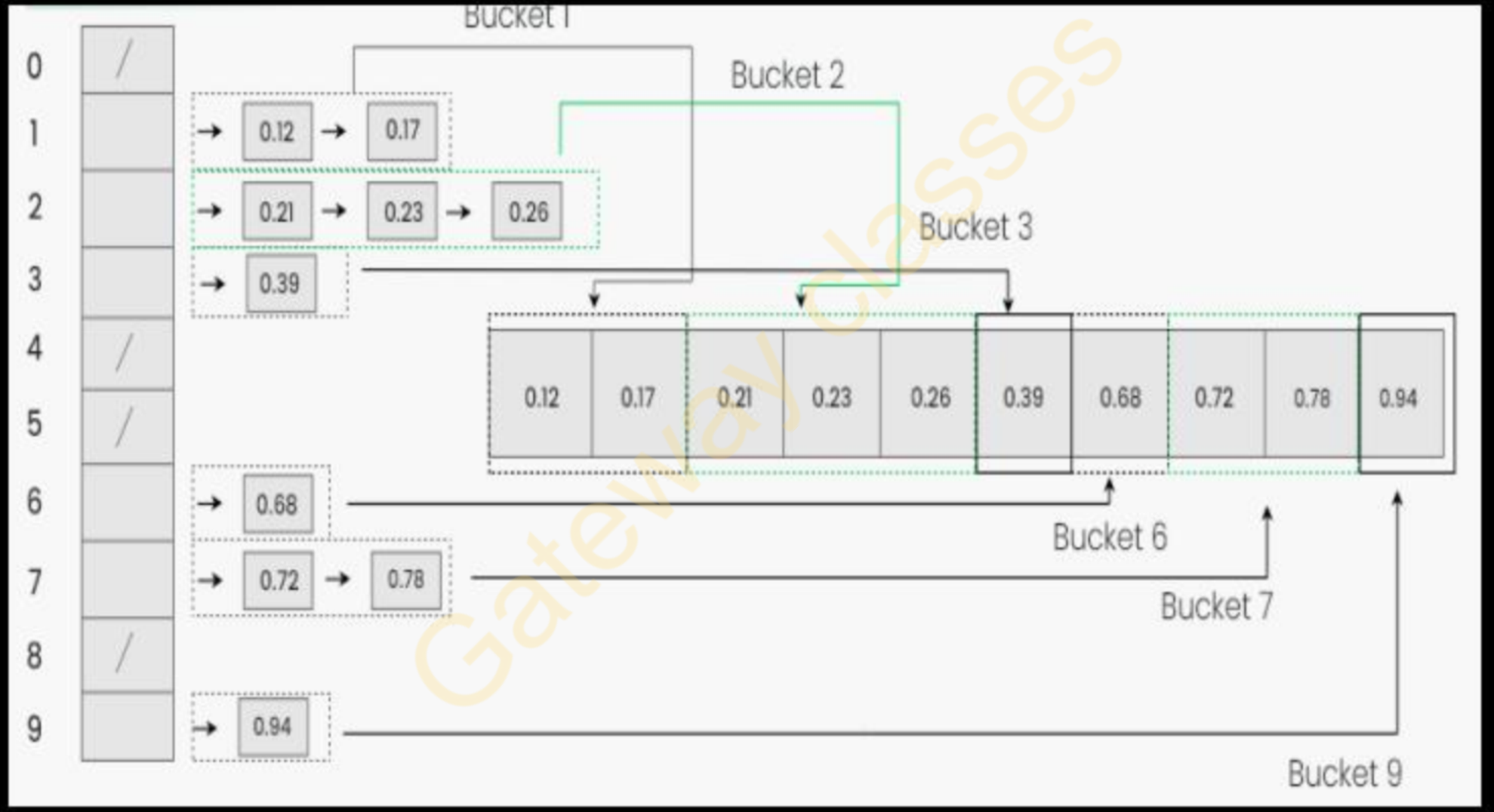
Step 2: Insert elements into buckets from input based on their range.



Step 3: Sort elements within each bucket using any stable sorting



Step 4: Gather the elements from each bucket and put them back into the original array.



The advantages of bucket sort are -

- Bucket sort reduces the no. of comparisons.
- It is asymptotically fast because of the uniform distribution of elements.

The limitations of bucket sort are -

- It is not useful if we have a large array because it increases the cost.
- It is not an in-place sorting algorithm, because some extra space is required to sort the buckets.

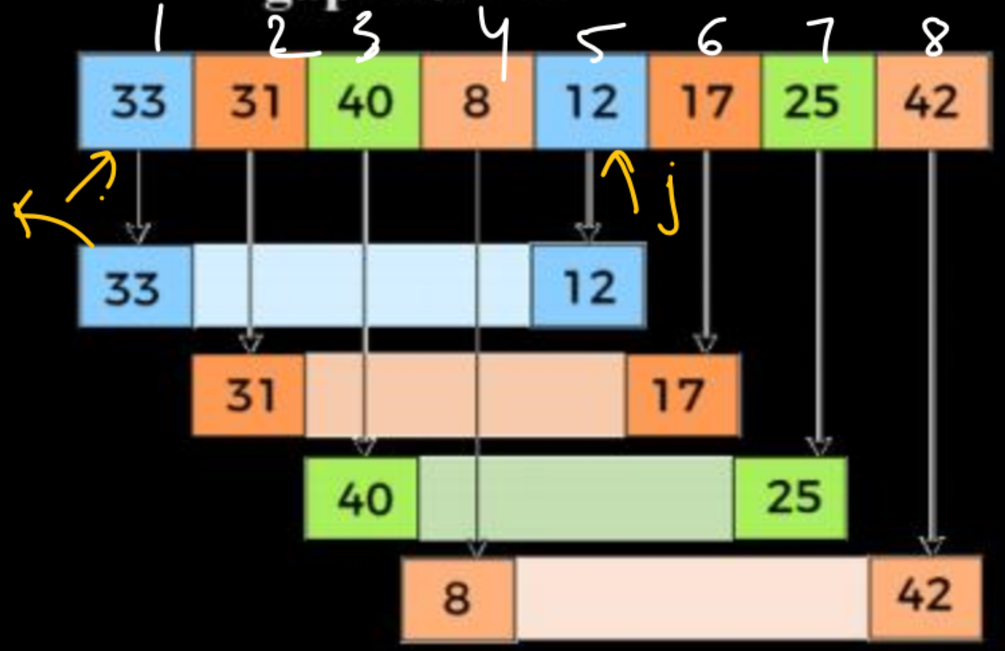
- The best and average-case complexity: $O(n + k)$ {when buckets get approx. equal number of items.}
- worst-case complexity of bucket sort is $O(n^2)$, {when one bucket gets all the items.}

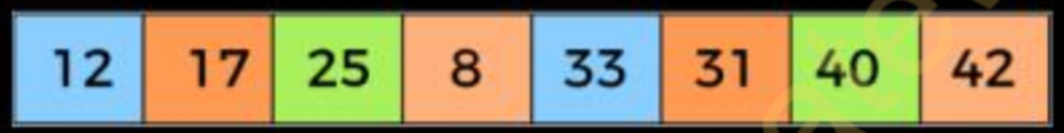
Shell Sort

- Extended version of insertion sort.
- Comparison-based and in-place sorting algorithm.
- In insertion sort, to move an element to a far-away position, many movements are required but shell sort overcomes this drawback by swapping of far-away elements .
- This algorithm first sorts the elements that are far away from each other, then it subsequently reduces the gap between them.

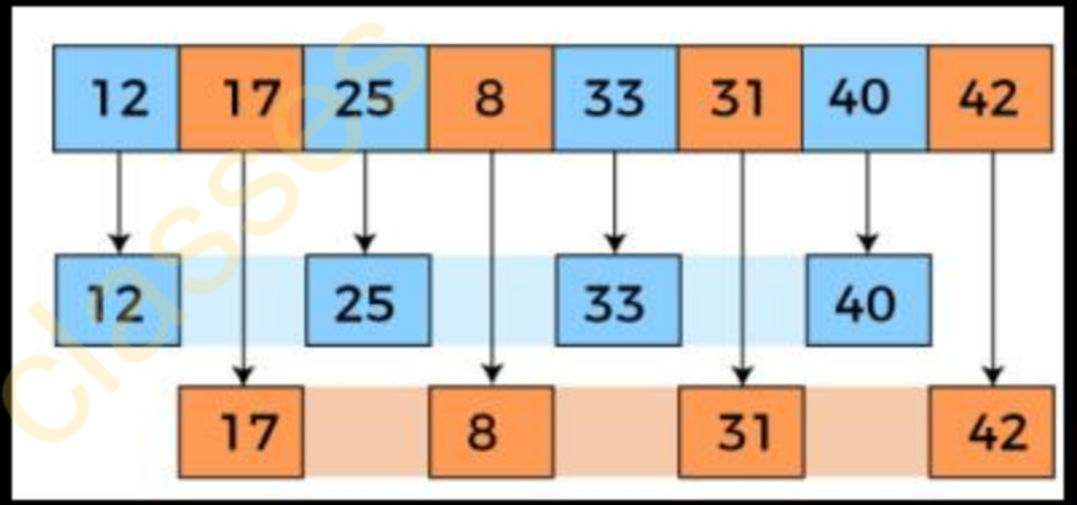
n/2 = 4

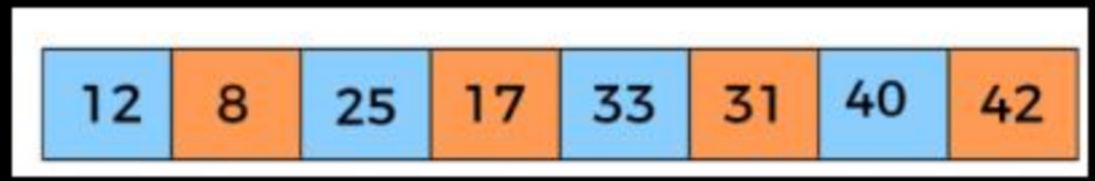
gap= n/2= 4





gap= 4/2= 2





gap= 2/2= 1

12	8	25	17	33	31	40	42
12	8	25	17	33	31	40	42
8	12	25	17	33	31	40	42
8	12	25	17	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	31	33	40	42
8	12	17	25	31	33	40	42
8	12	17	25	31	33	40	42

Insertion sort

```
Shell-Sort(arr, n)
for(gap = n/2; gap > 0; gap = gap / 2)
        for(j = gap; j<n; j++)
        {
                for(k = j-gap; k>=0; k -= gap)
                {
                        if( arr [k+ gap] >= arr [k])
                                break;
                        else
                        {
                                int temp;
                                temp = arr[ k+ gap];
                                arr [k+ gap] = arr [k];
                                arr [k] = temp;
                        }
                }
        }
```

// for gap values

- **Best Case Complexity - O(n*logn) when** the array is already sorted.
- **Average Case Complexity - O(n*logn) when** array elements are not properly ascending and not properly descending.
- **Worst Case Complexity - $O(n^2)$.** It occurs when the array elements are required to be sorted in reverse order.

V. Imp **Comparison of sorting algorithms**

Sorting Algorithm	Average Case	Best Case	Worst Case	Space Complexity	Stable Sort?
Insertion Sort	$O(n^2)$	O(n)	$O(n^2)$	O(1)	Yes
Quick Sort	O(n.log(n))	O(n.log(n))	$O(n^2)$	O(log(n))in best case O(n) in worst case	No
Merge Sort	O(n.log(n))	O(n.log(n))	O(n.log(n))	O(n)	Yes
Heap Sort	O(n.log(n))	O(n.log(n))	O(n.log(n))	O(1)	No
Counting Sort	O(n+k)	O(n+k)	O(n+k)	O(k)	Yes
Radix Sort	O(n*d)	O(n*d)	O(n*d)	O(n + k)	Yes
Bucket Sort	O(n+k)	O(n+k)	$O(n^2)$	O(n)	Yes