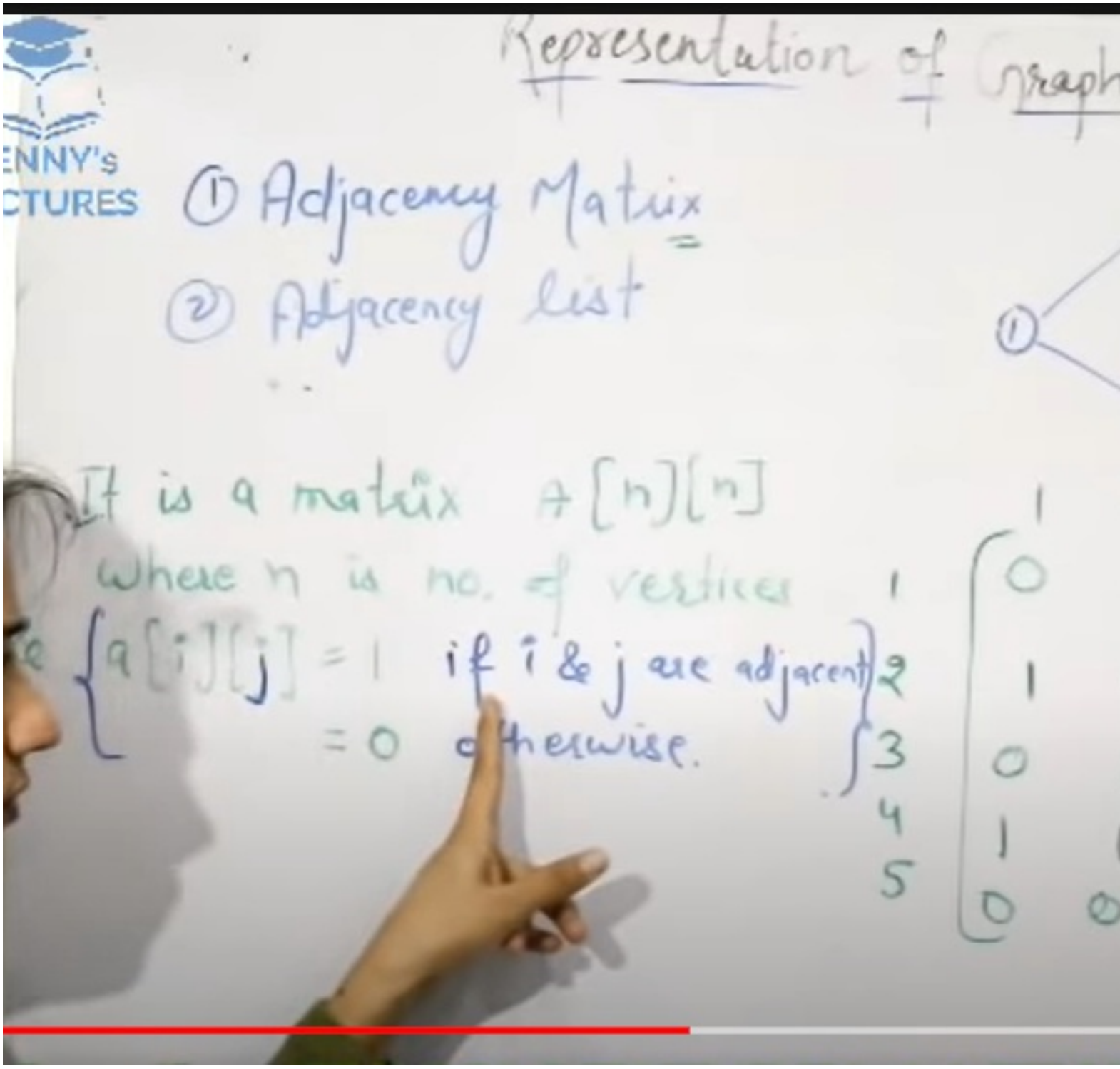
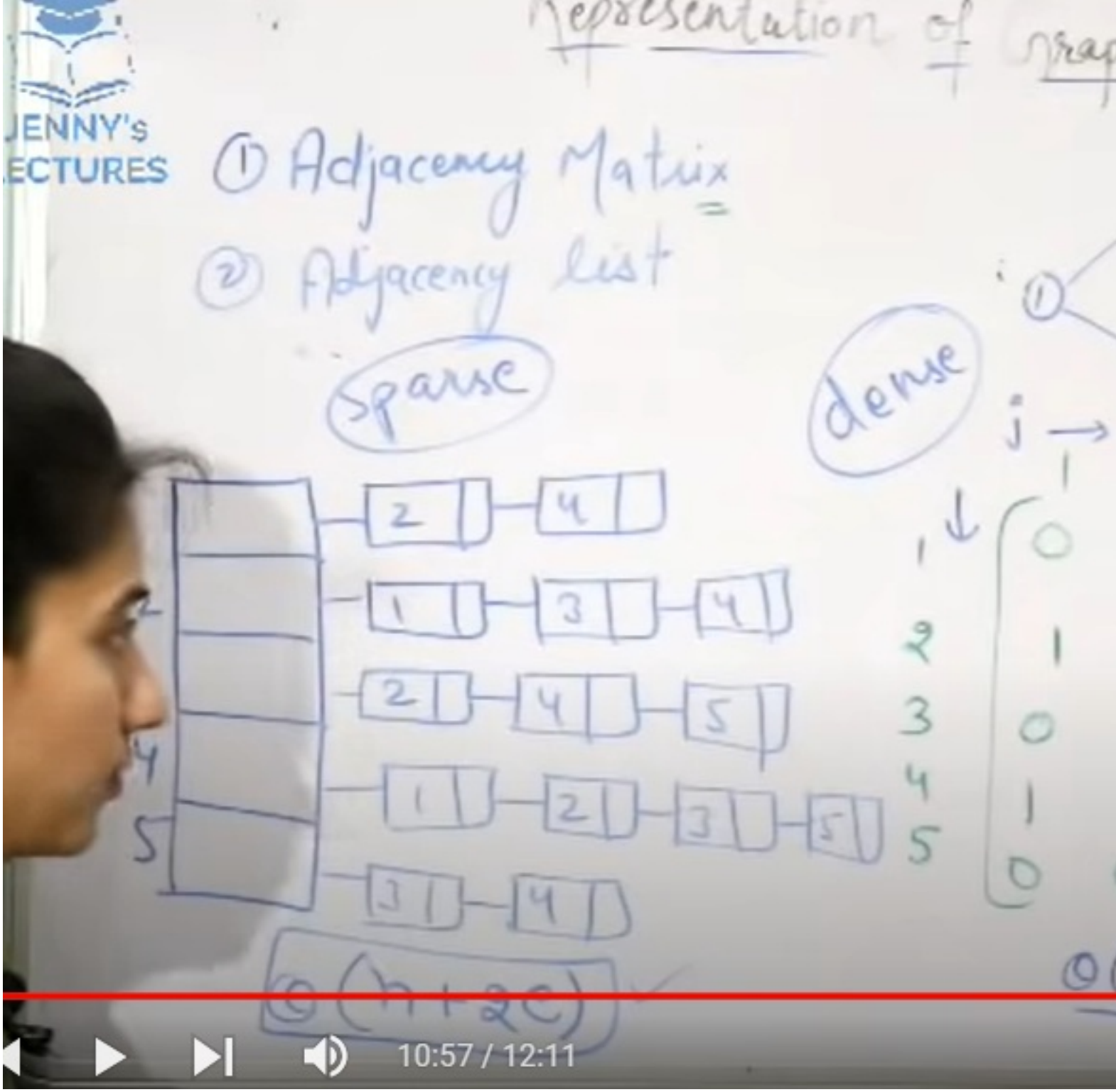
DAA

TREE

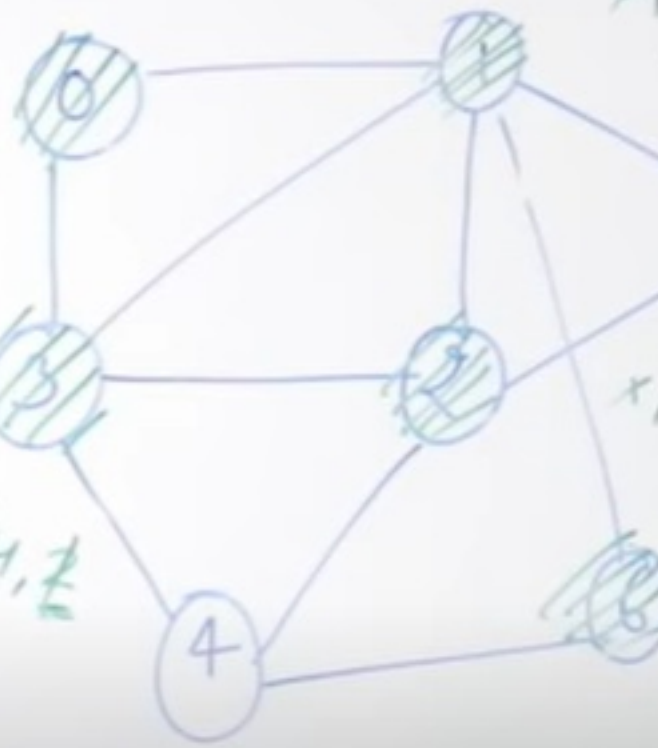

Representation of Graph
ENNY's
CTURES
① Adjacency Matrix
② Adjacency list
It is a matrix A[n][n]
where n is no. of vertices
a[i][j] = 1 if i & j are adjacent
= 0 otherwise.
1
2
3
4
5
1
0
1
0
1
0


JENNY's
LECTURES
Representation of Graph
① Adjacency Matrix
② Adjacency list
sparse
dense
2
4
1
3
4
2
4
5
1
2
3
5
3
4
O(n+2e)
i
j
1
2
3
4
5
0
1
0
1
0
1
0
10:57 / 12:11

Graph Traversals

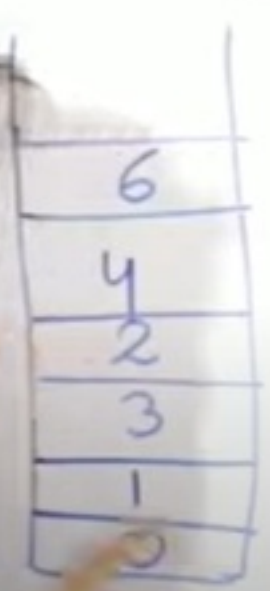
Queue:-

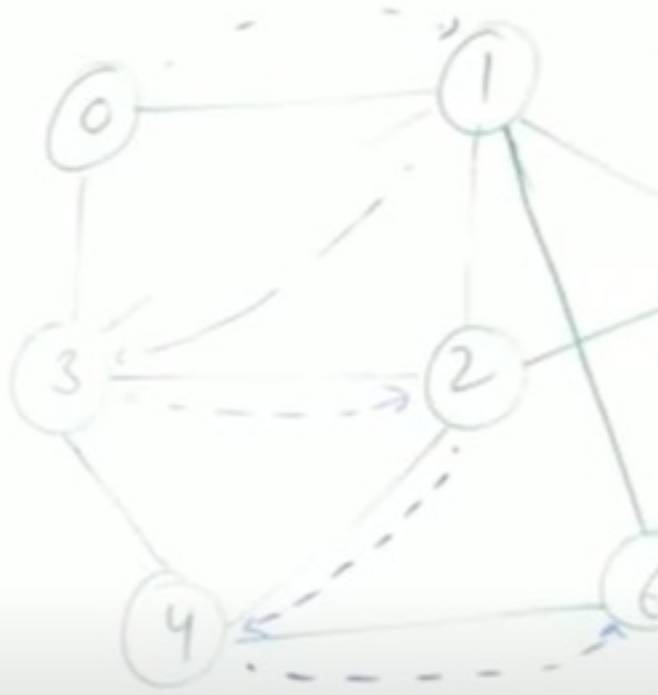
Result → 0 1 3 2 5 6 4



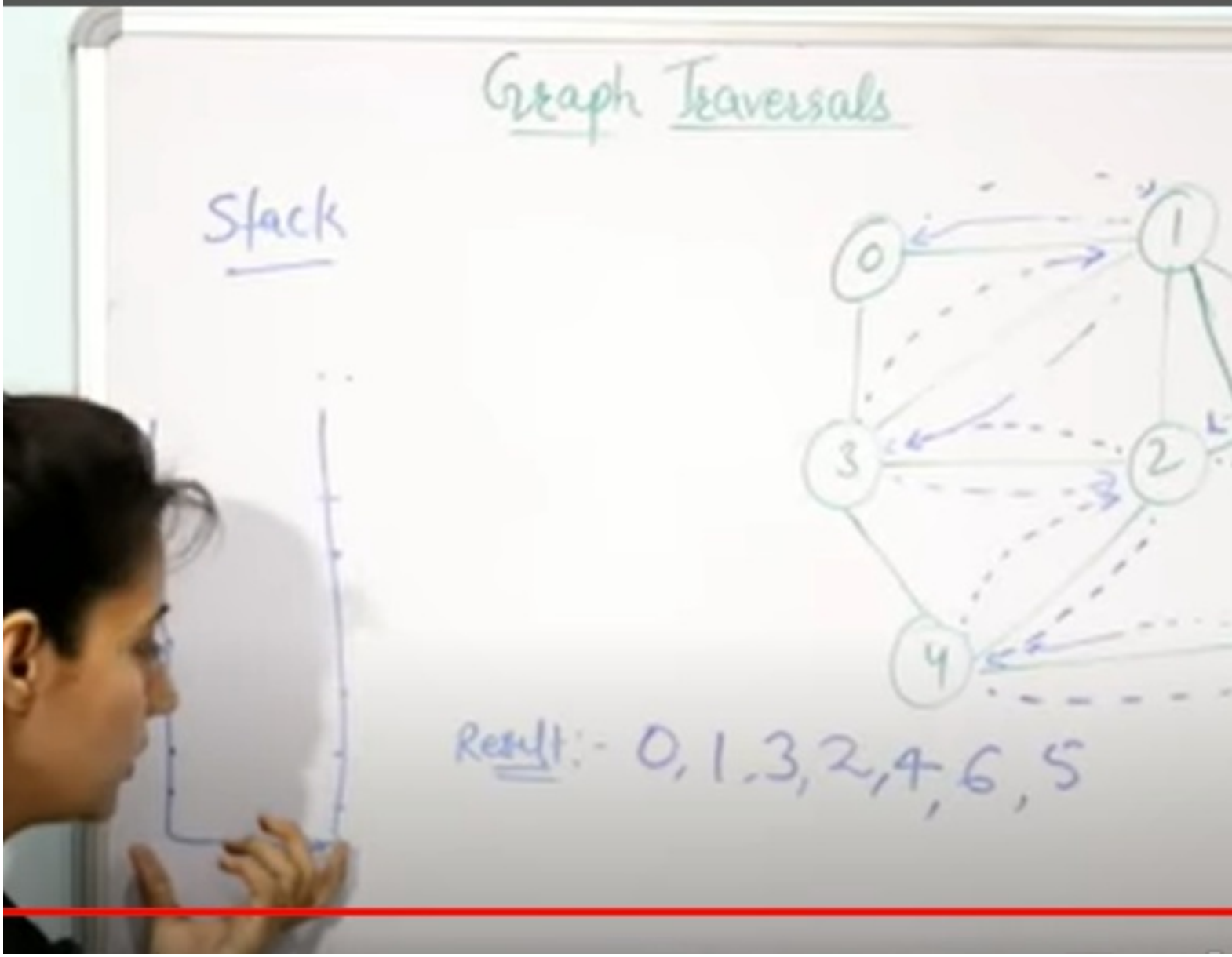
Graph Traversals

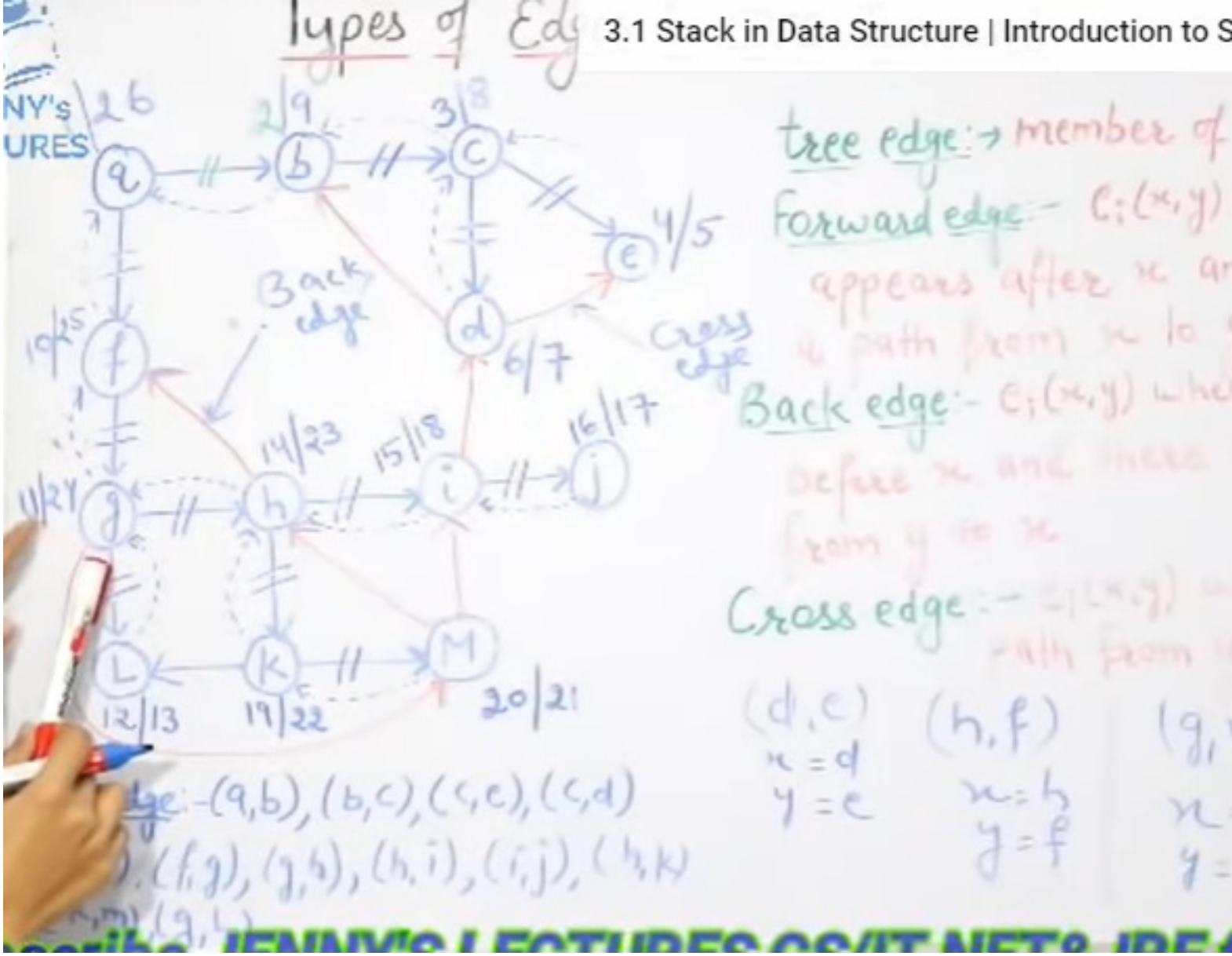
Stack

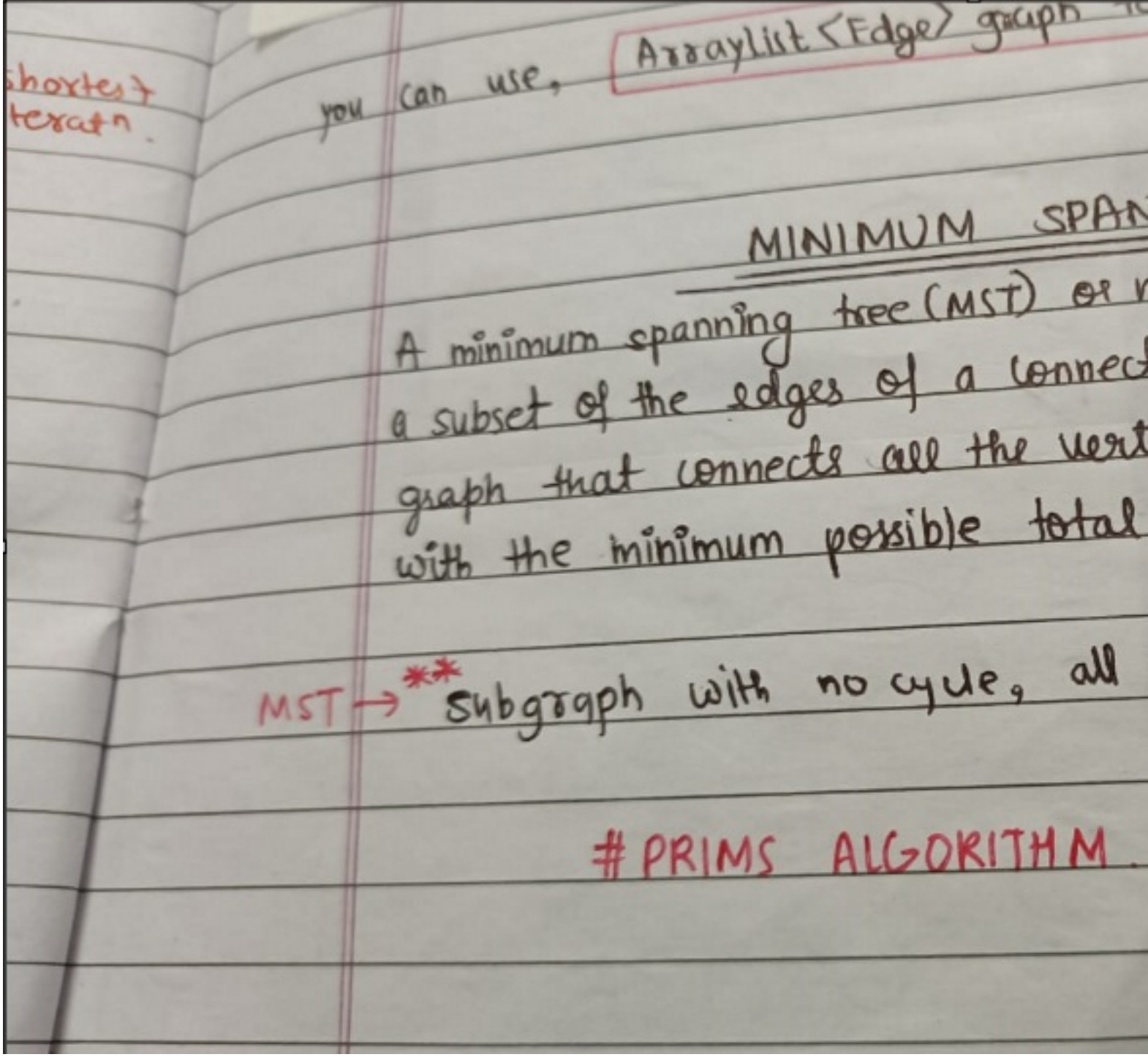




Result:- 0, 1, 3, 2, 4, 6


Graph Traversals
Stack
0
1
3
2
4
Result:- 0, 1, 3, 2, 4, 6, 5


Types of Edg
3.1 Stack in Data Structure | Introduction to S
NY's
URES
tree edge:→ member of
Forward edge:- e:(x,y)
appears after x
Back edge:- e:(x,y) whe
Cross edge:-
Back edge
Cross edge
(d,e)
x=d
y=e
(h,f)
x=h
y=f
(a,b),(b,c),(c,e),(c,d)
(f,g),(g,h),(h,i),(i,j),(h,k)



Prism algo

The algorithm starts with an empty spanning tree. The id
vertices. The first set contains the vertices already includ
contains the vertices not yet included. At every step, it co
connect the two sets and picks the minimum weight edge
the edge, it moves the other endpoint of the edge to the s

How does Prim's Algorithm Work?

The working of Prim's algorithm can be described by using

Step 1: *Determine an arbitrary vertex as the starting ver*

Step 2: *Follow steps 3 to 5 till there are vertices that ar*

as fringe vertex).

Step 3: *Find edges connecting any tree vertex with the f*

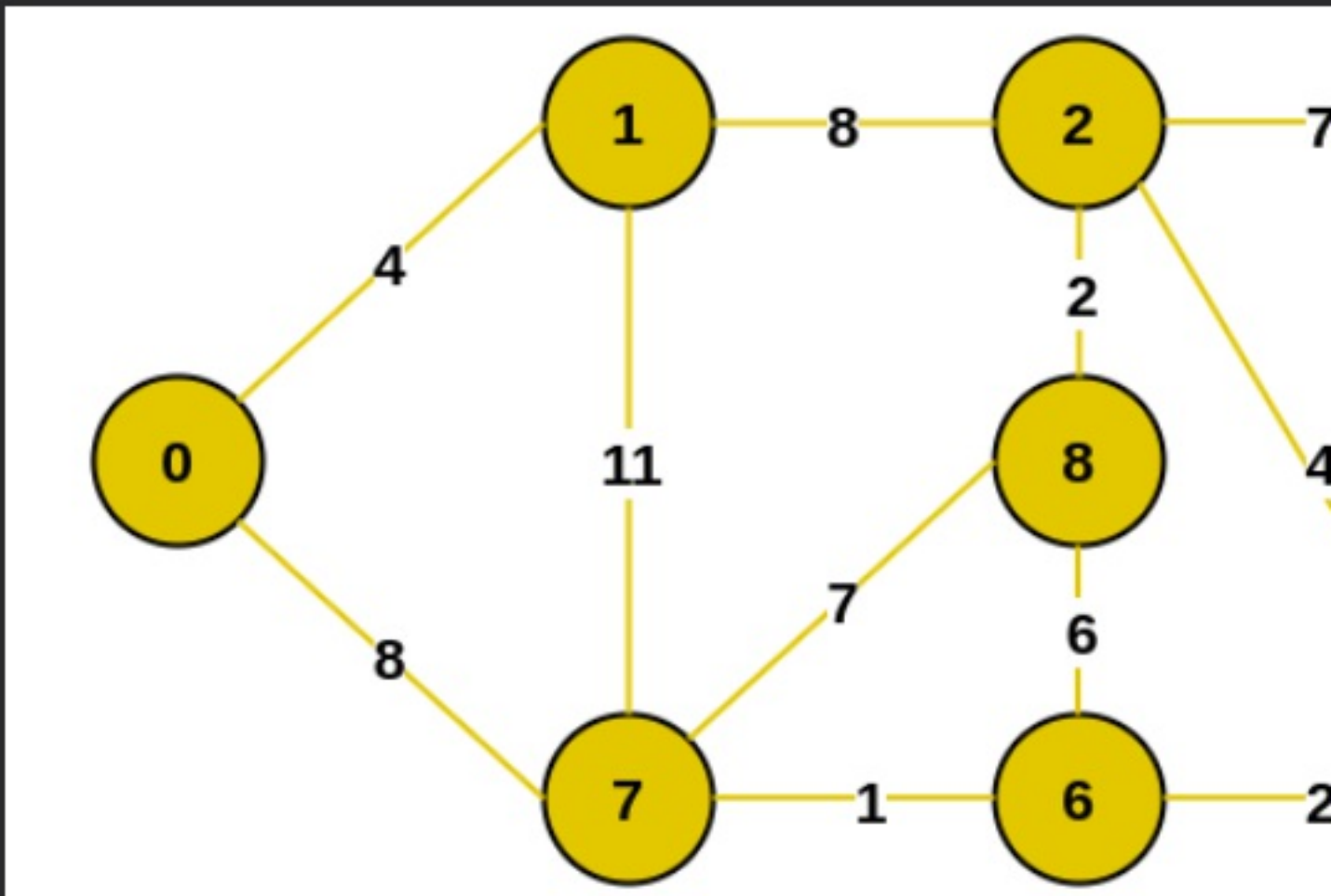
Step 4: *Find the minimum among these edges.*

Step 5: *Add the chosen edge to the MST if it does not fo*

Step 6: *Return the MST and exit*

Illustration of Prim's Algorithm:

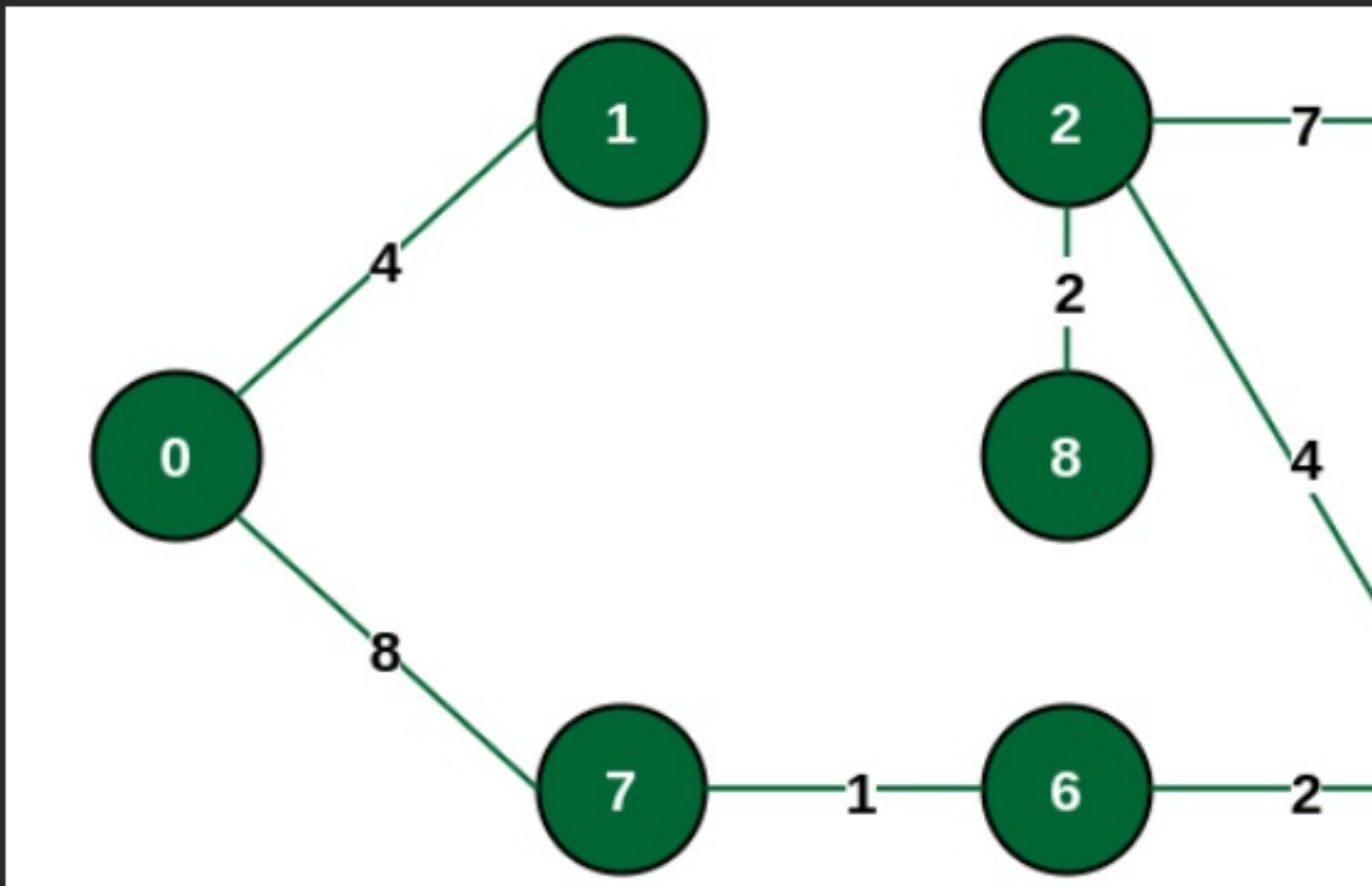
Consider the following graph as an example for which w
Spanning Tree (MST).



Example of a Graph

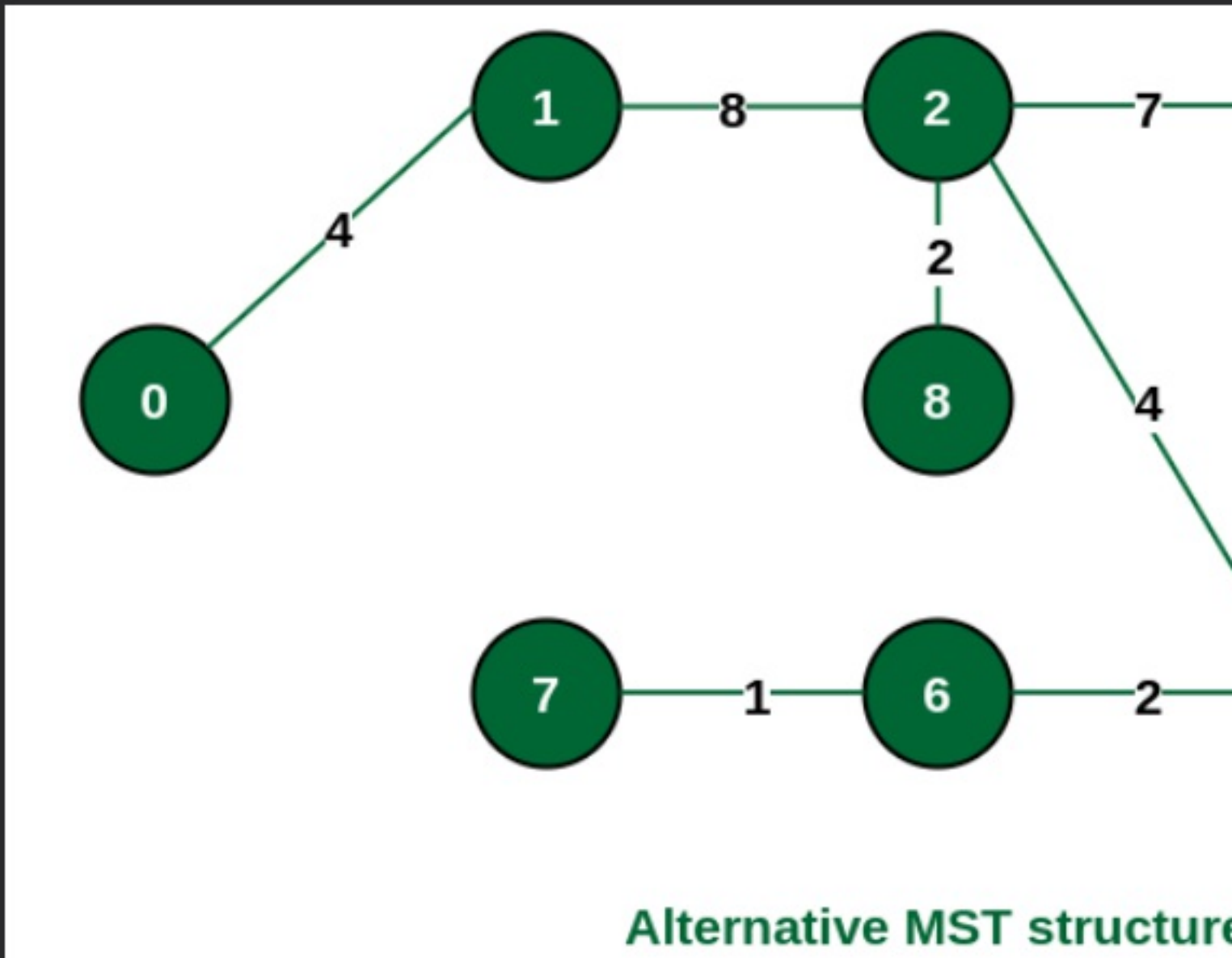
The final structure of the MST is as follows and the weigh

*8 + 1 + 2 + 4 + 2 + 7 + 9) = **37**.*



The final structure of MS

Note: *If we had selected the edge {1, 2} in the third step th*
following.



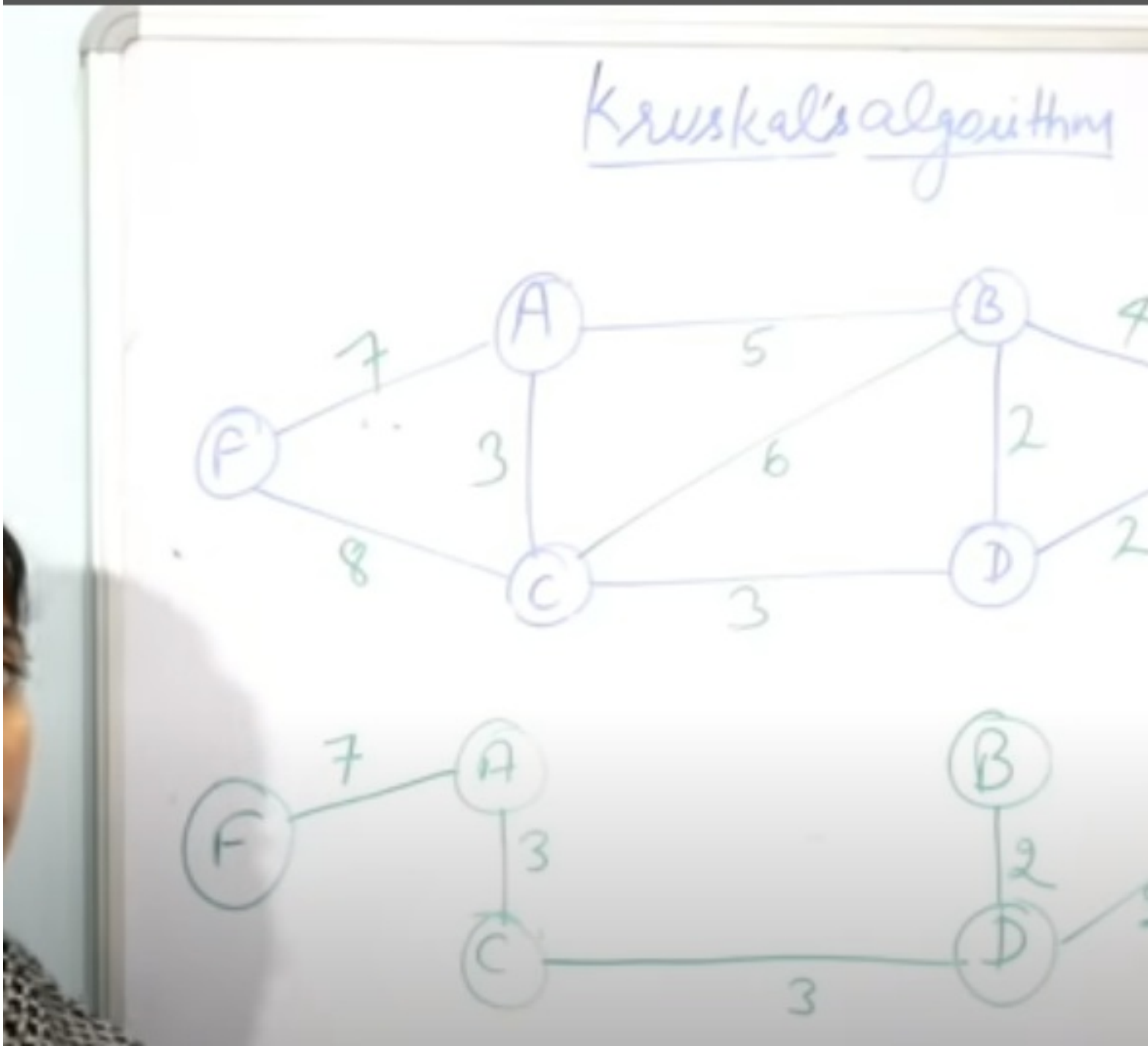
Single Source Shortest Path **Kruskals Algorithm**

How to find MST using Kruskal's algorit

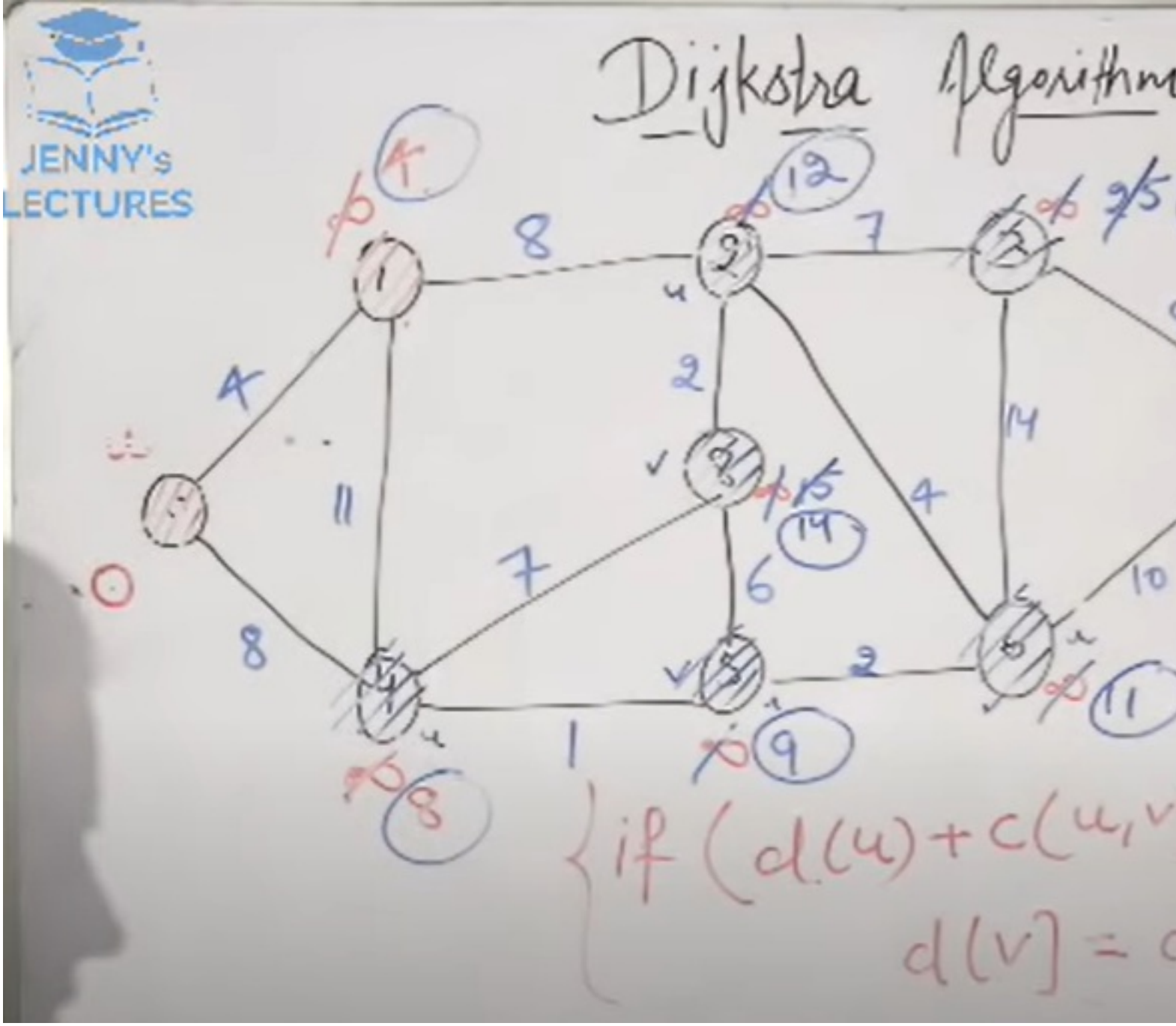
Below are the steps for finding MST using Kruskal's algori

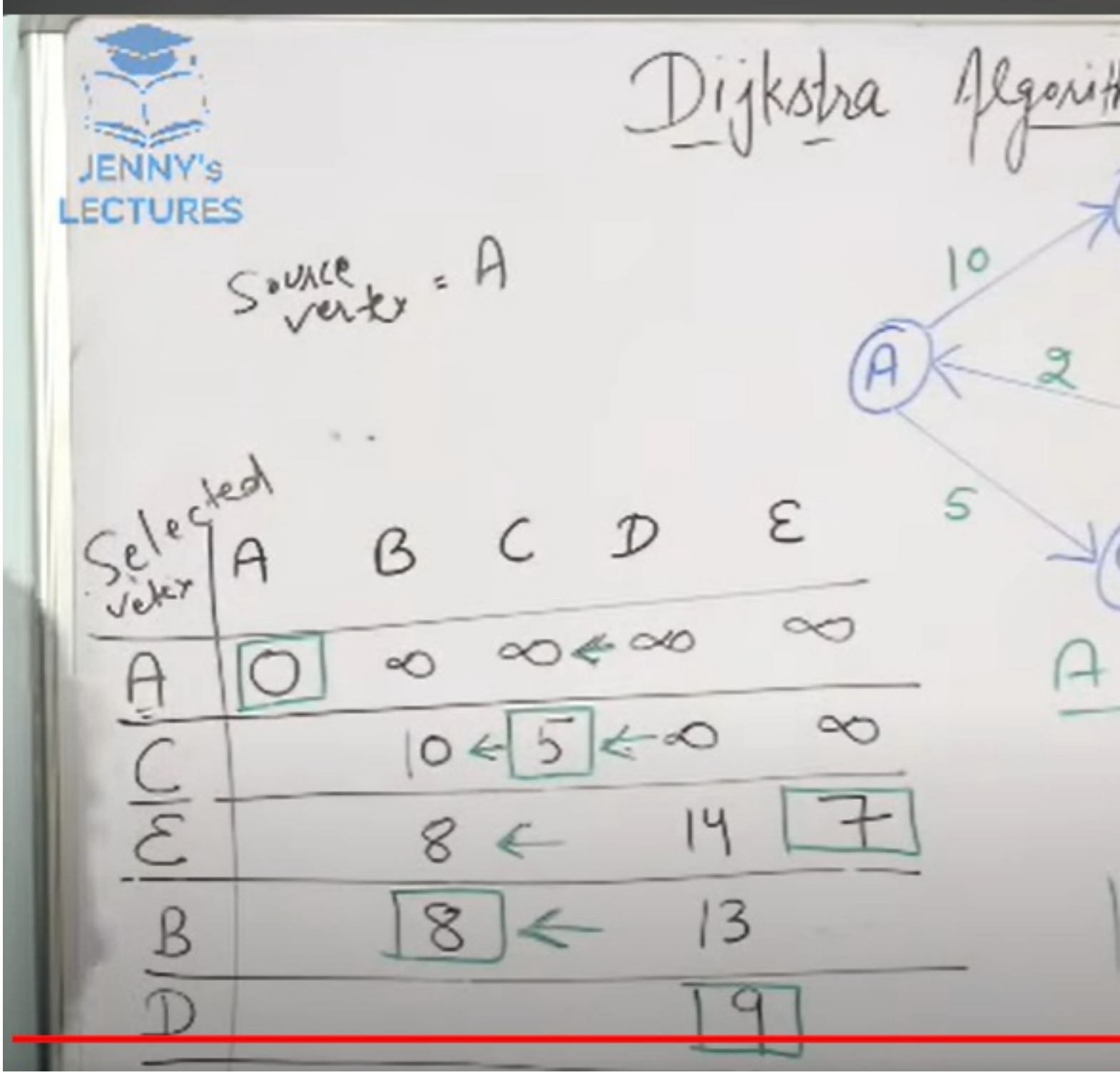
1. Sort all the edges in non-decreasing order of their weig
2. Pick the smallest edge. Check if it forms a cycle with th
   cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are (V-1) edges in the spann

Time Complexity: O(E * logE) or O(E * logV)



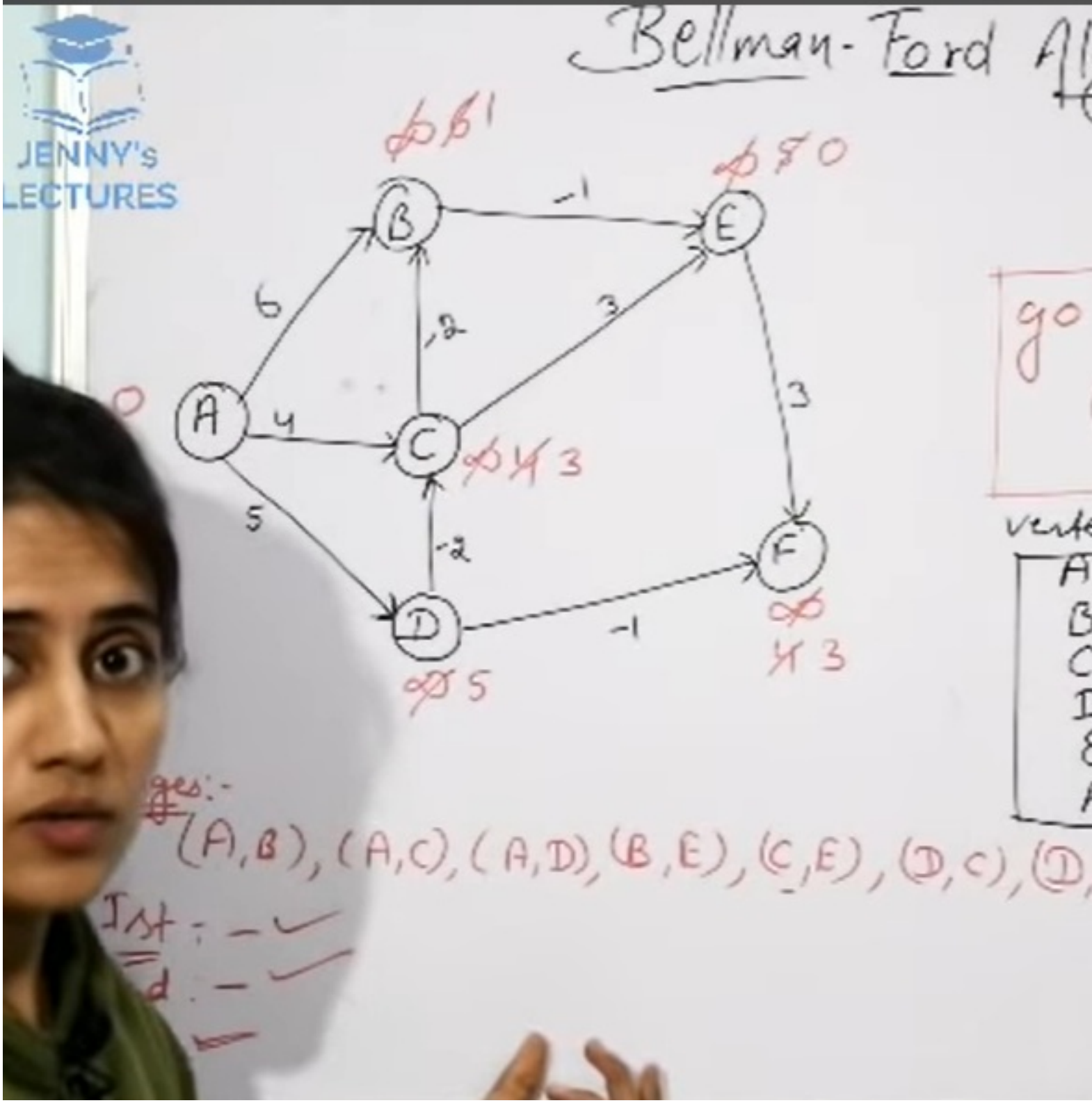
Dijksta Algorithm - Single Source Shortest Path

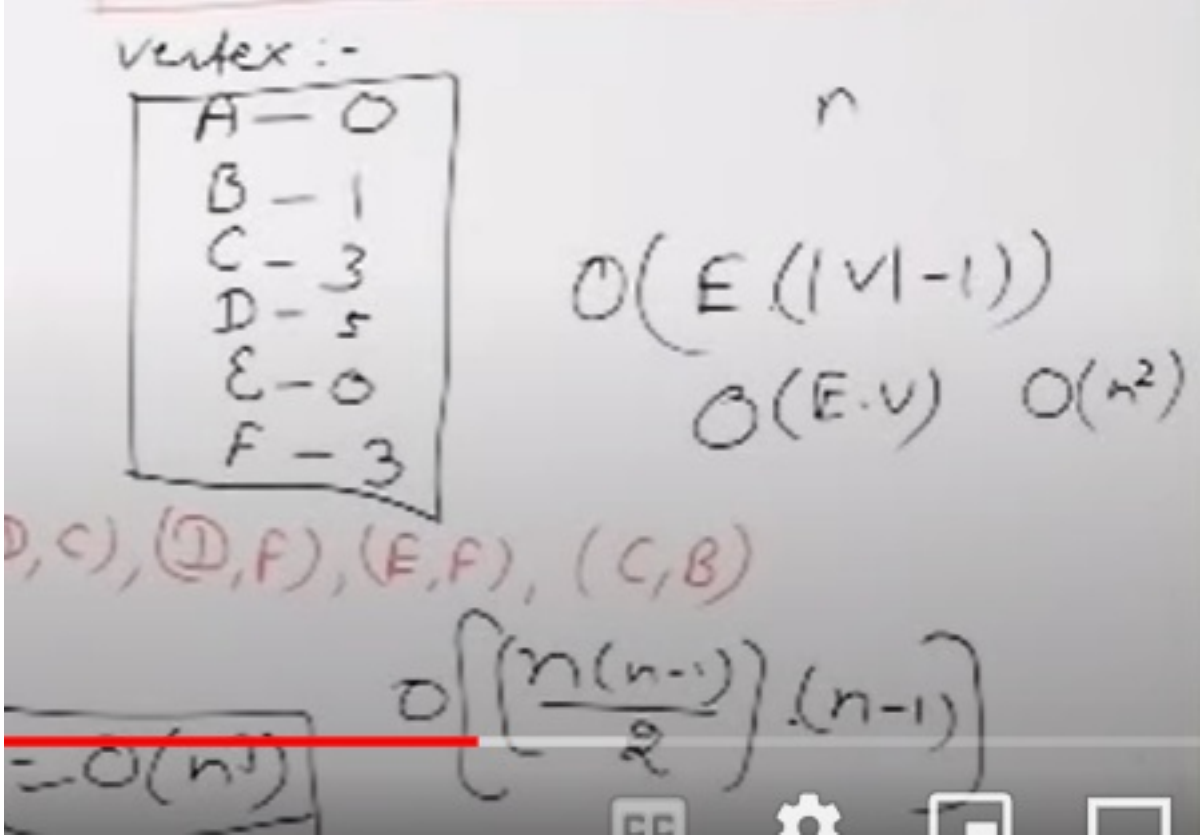

Dijkstra Algorithm
JENNY's LECTURES
8
7
4
2
14
11
7
6
10
8
2
1
4
12
14
9
11
8
if (d(u) + c(u,v
d(v] =



DIJKSTRA NOT ALWAYS GIVE RIGHT ANSWER IN CASE OF ANY EDGE HAVING NEG WEIGHT FOR THAT CASE WE USE BELLMAN FORD .THIS GIVE CORRECT BUT IT IS SLOWER THAN DIJKSTRA

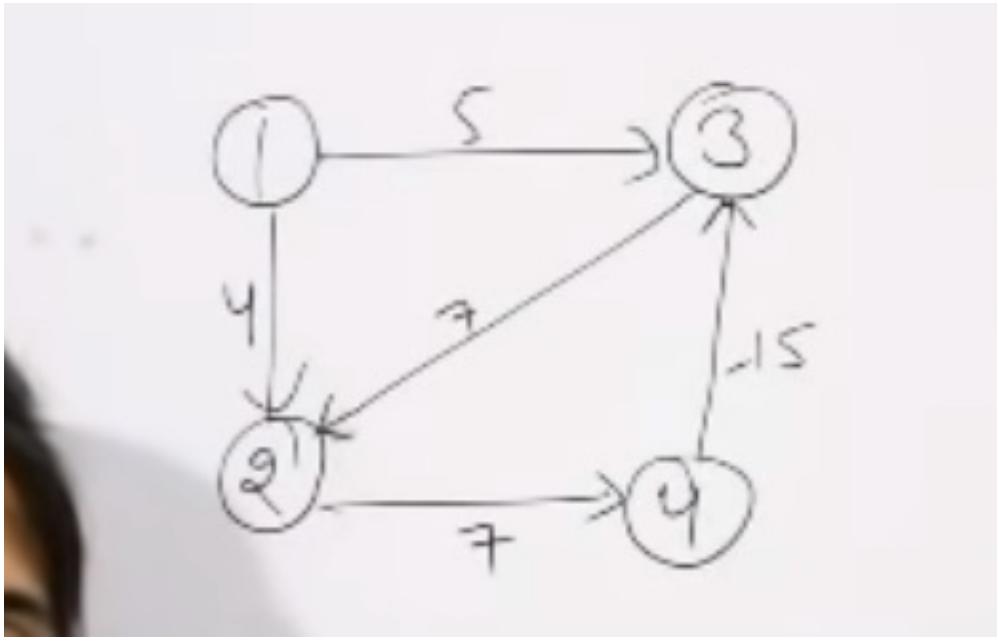
Bellman Ford Algorithm-Single Source Shortest Path

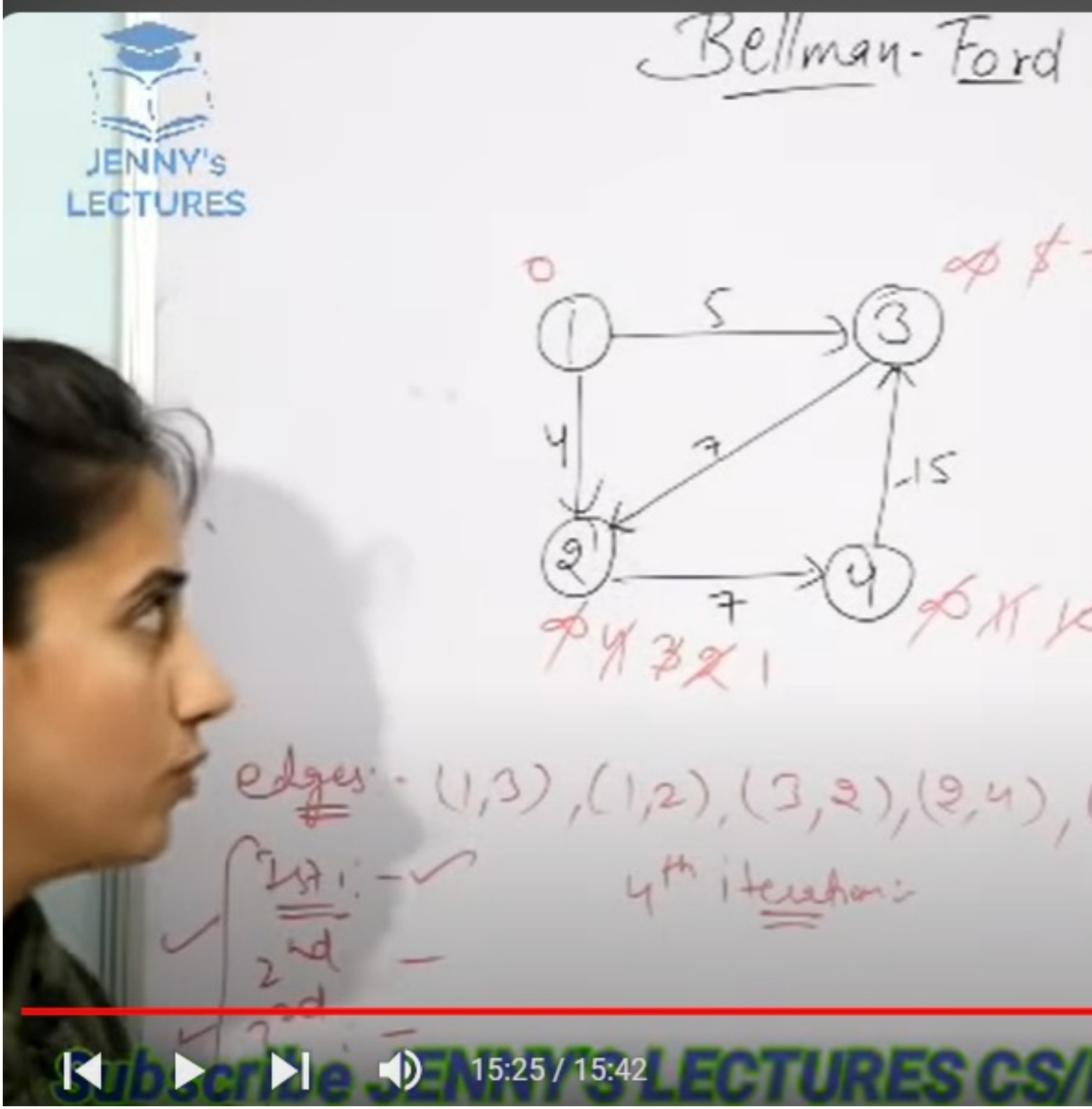

Bellman-Ford
JENNY's LECTURES
B
E
A
C
D
F
6
-1
2
3
4
3
5
-2
-1
go
(A,B), (A,C), (A,D), (B,E), (C,E), (D,C), (D,



IT WILL NOT RUN IF ANY GRAPH CONTAIN ANY NEGATIVE WEIGTH CYCLE OR LOOP

EXAMPLE-




Bellman-Ford
JENNY's LECTURES
5
4
7
-15
7
edges
(1,3), (1,2), (3,2), (2,4),
4th iteration
15:25 / 15:42

Floyd

All Pair Shortest Path (Fl

$$D^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 9 & -4 & \infty \\ 6 & 0 & 2 & 2 \\ \infty & & 0 & \\ \infty & \cdots & & 0 \end{bmatrix} \end{array}$$

$D^0[2,3]$ $\quad D^0[2,1] + D^0[1,3]$

$\infty \quad > \quad 6 + (-4)$ (2)

$D^0[2,4] = D^0[2,1] + D^0[1,4]$

$2 \quad < \quad 6 + \infty : \infty$

$D^0[3,2] : D^0[3,1] + D^0[1,2]$

$5 \quad < \quad \infty + 9 : \infty$

All Pair Shortest Path (Floy

$$D^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 9 & -4 & \infty \\ 6 & 0 & 2 & 2 \\ \infty & 5 & 0 & \infty \\ \infty & \infty & 1 & 0 \end{bmatrix} \end{array}$$

$$D^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 9 & -4 & \\ 6 & 0 & 2 & 2 \\ & 5 & 0 & \\ & \infty & & 0 \end{bmatrix} \end{array}$$

$D^1[1,3] : D^1[1,2] + D^1[2,3]$

$-4 \quad < \quad 9 + 2 = 11$

$D^1[1,4] : D^1[1,2] + D^1[2,4]$

All Pair Shortest Path (Floyd

D^1 =

	1	2	3	4
1	0	9	-4	∞
2	6	0	2	2
3	∞	5	0	∞
4	∞	∞	1	0

$D^2[1,2] = D^2[1,3] + D^2$

$9 > -4 + 5$

$D^2[1,4] = D^2[1,3] + D^2($

$-4 +$

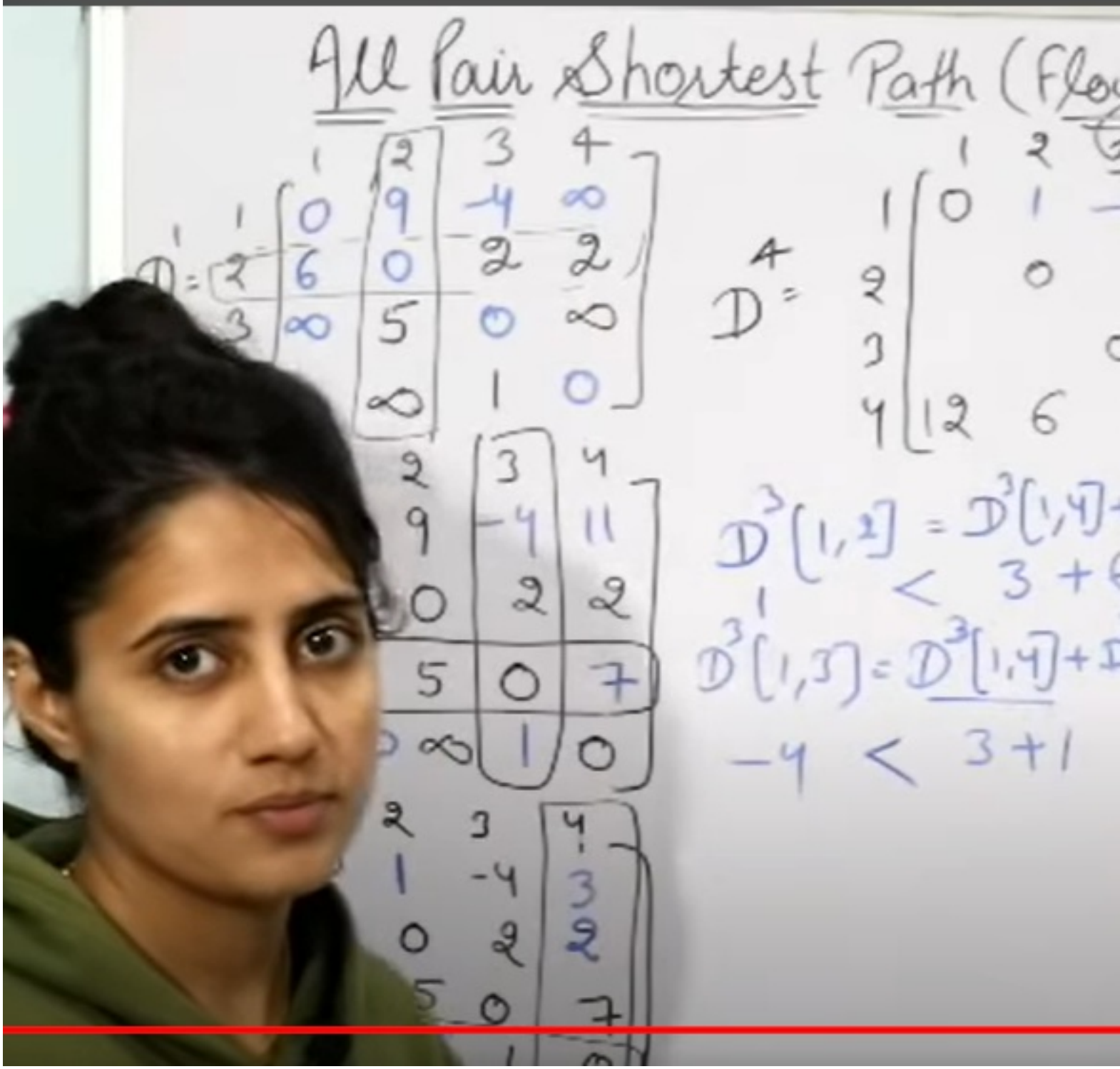
11

D^2 =

	1	2	3	4
1	0	9	-4	11
2	6	0	2	2
3	11	5	0	7
4	∞	∞	1	0

D^3 =

	1	2	3	4
1	0	1	-4	
2		0	2	
3	11	5	0	7
4			1	0


All Pair Shortest Path (Flo
D¹ =
D⁴ =
D³[1,2] = D³[1,4]
< 3 +
D³[1,3] = D³[1,4]+
-4 < 3+1

All Pair Shortest Path (Floy

D^1 =

	1	2	3	4
1	0	9	-4	∞
2	6	0	2	2
3	∞	5	0	∞
4	∞	∞	1	0

D^4 =

	1	2	3
1	0	1	-
2	6	0	
3	11	5	0
4	12	6	

D^2 =

	1	2	3	4
1	0	9	-4	11
2	6	0	2	2
3	11	5	0	7
4	∞	∞	1	0

$D^K[i,j] = \min\{D^K$

$D^3[1,2] = D^3[1,$

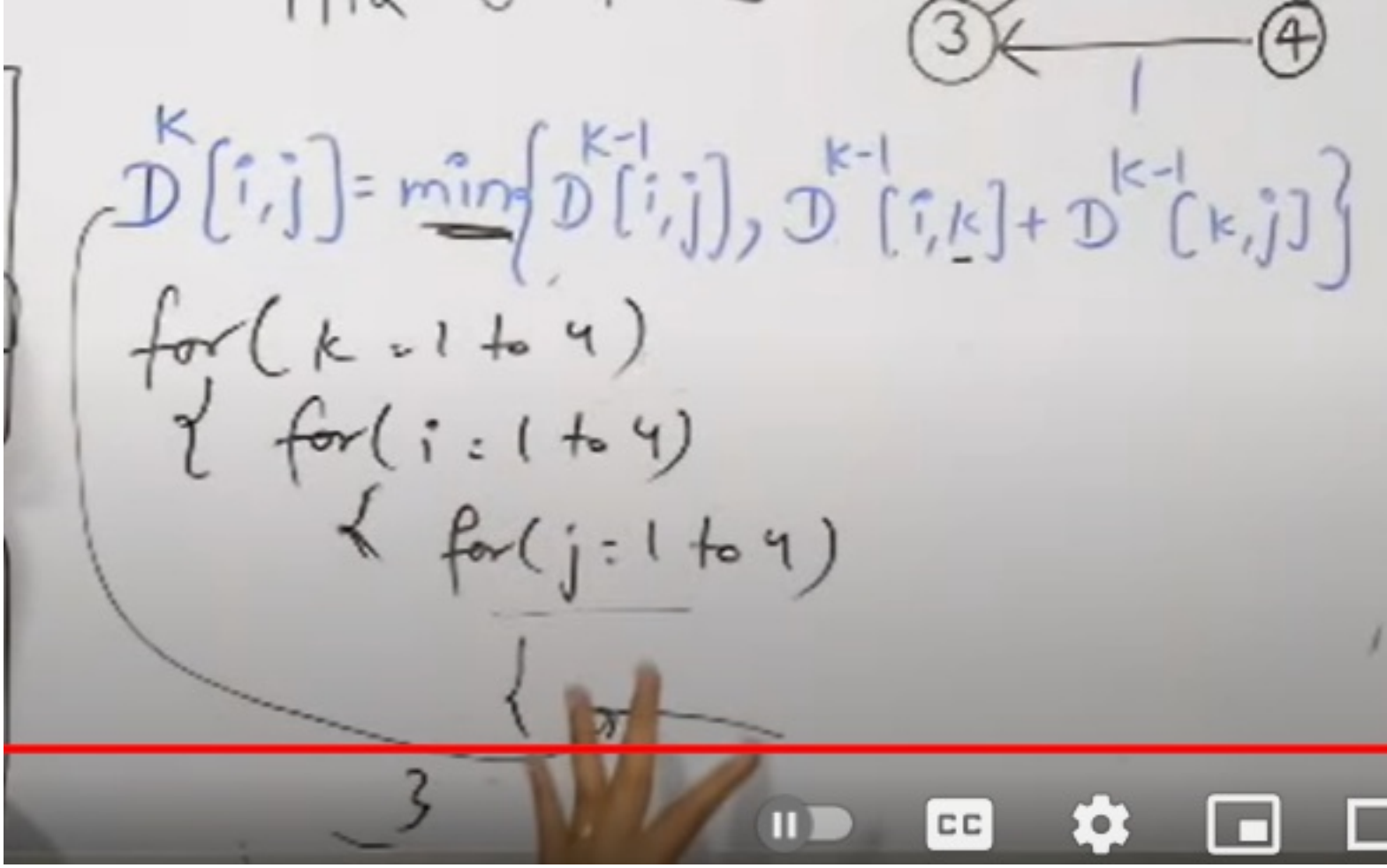
① < 3

D^3 =

	1	2	3	4
1	0	1	-4	3
2			2	2
			0	7
			1	0

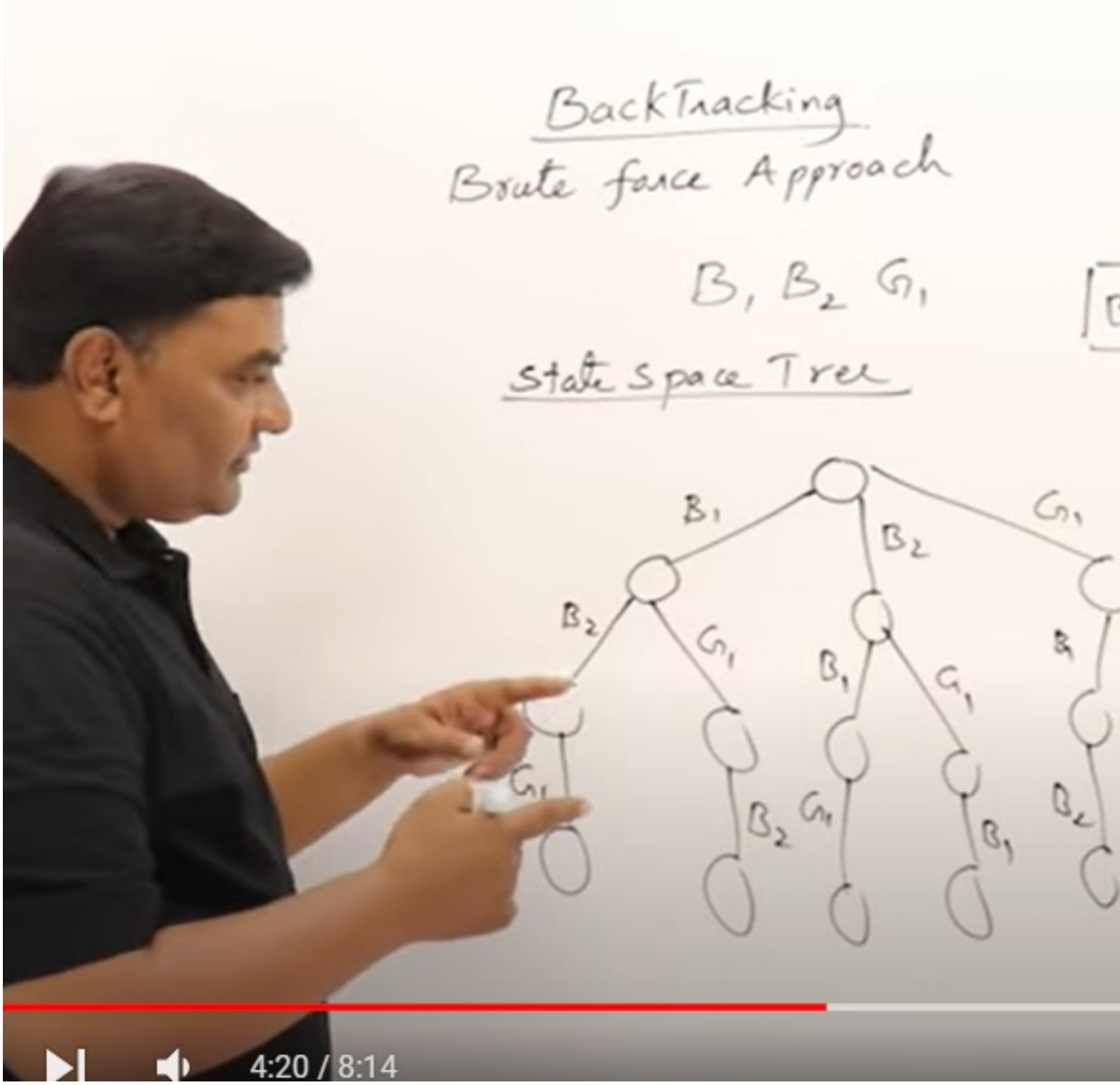
k <= 4
i = 1
j = 1

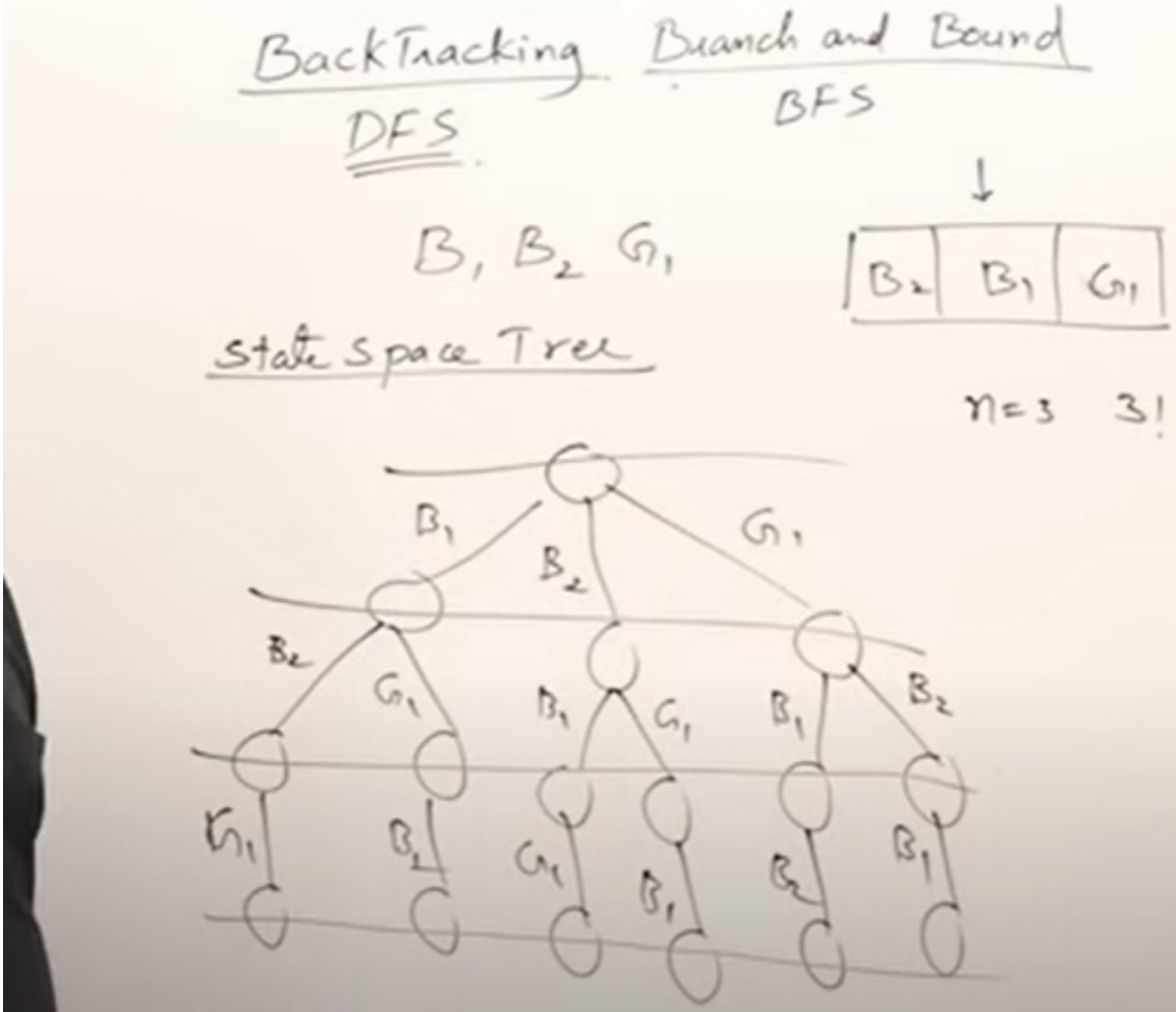
28:59 / 31:22 Jenny's Lectures



0/1 knapsack problem-Dynamic Programming

BACKING TRACKING


BackTracking
Brute force Approach
B, B2 G1
State Space Tree
B1
B2
G1
B2
G1
B1
G1
G1
B2
G1
B1
B2
4:20 / 8:14



What is Backtracking?

Backtracking is a problem-solving algorithmic technique incrementally by trying **different options** and **undoing** t commonly used in situations where you need to explore problem, like searching for a path in a maze or solving p reached, the algorithm backtracks to the previous decisi until a solution is found or all possibilities have been ex

Pseudocode for Backtracking

The best way to implement backtracking is through recurs
summarised as per the given Pseudocode:

```
void FIND_SOLUTIONS( parameters):
    if (valid solution):
        store the solution
        Return
    for (all choice):
        if (valid choice):
            APPLY (choice)
            FIND_SOLUTIONS (parameters)
            BACKTRACK (remove choice)
    Return
```

Recursion	
Recursion does not always need backtracking	Backtrack
Solving problems by breaking them into smaller, similar subproblems and solving them recursively.	Solving p exploring
Controlled by function calls and call stack.	Manage
Applications of Recursion: Tree and Graph Traversal, Towers of Hanoi, Divide and Conquer Algorithms, Merge Sort, Quick Sort, and Binary Search.	**Applic** problem, R Problem,

Applications of Backtracking

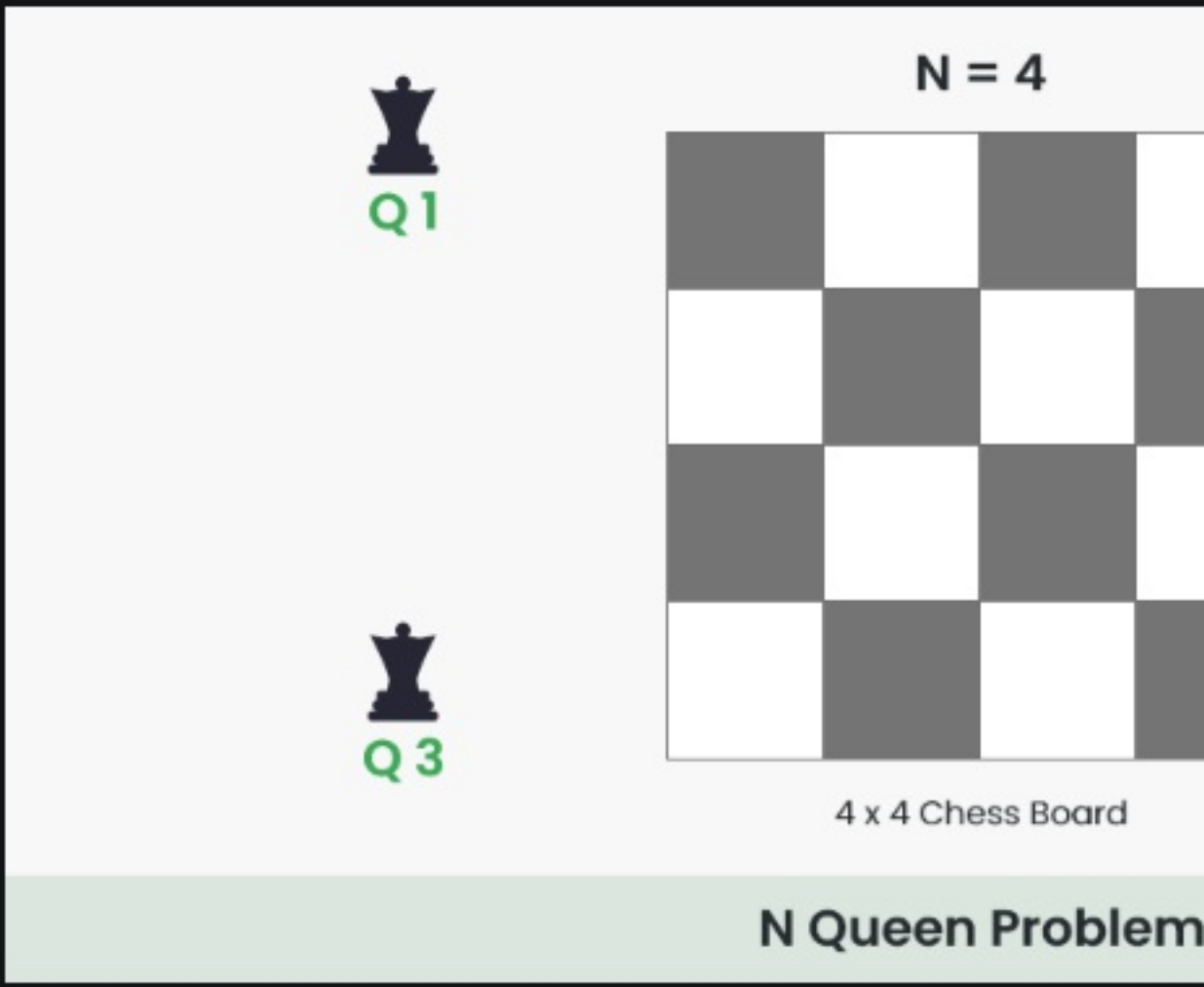
- Creating smart bots to play Board Games such as Chess
- Solving mazes and puzzles such as N-Queen problem.
- Network Routing and Congestion Control.
- Decryption
- Text Justification

4 Queens Problem

Read | Discuss | Courses | Practice

The **4 Queens** Problem consists in placing four queens or
queens attack each other. That is, no two queens are allo
same column or the **same diagonal**.

We are going to look for the solution for n=4 on a 4 x 4 c



4 Queens Problem using Backtracking A

> *Place each queen one by one in different rows, startin*
> *placing a queen in a row, check for clashes with alread*
> *there is no clash then mark this row and column as pa*
> *queen. In case, if no safe cell found due to clashes, the*
> *placement of recent queen) and return false.*

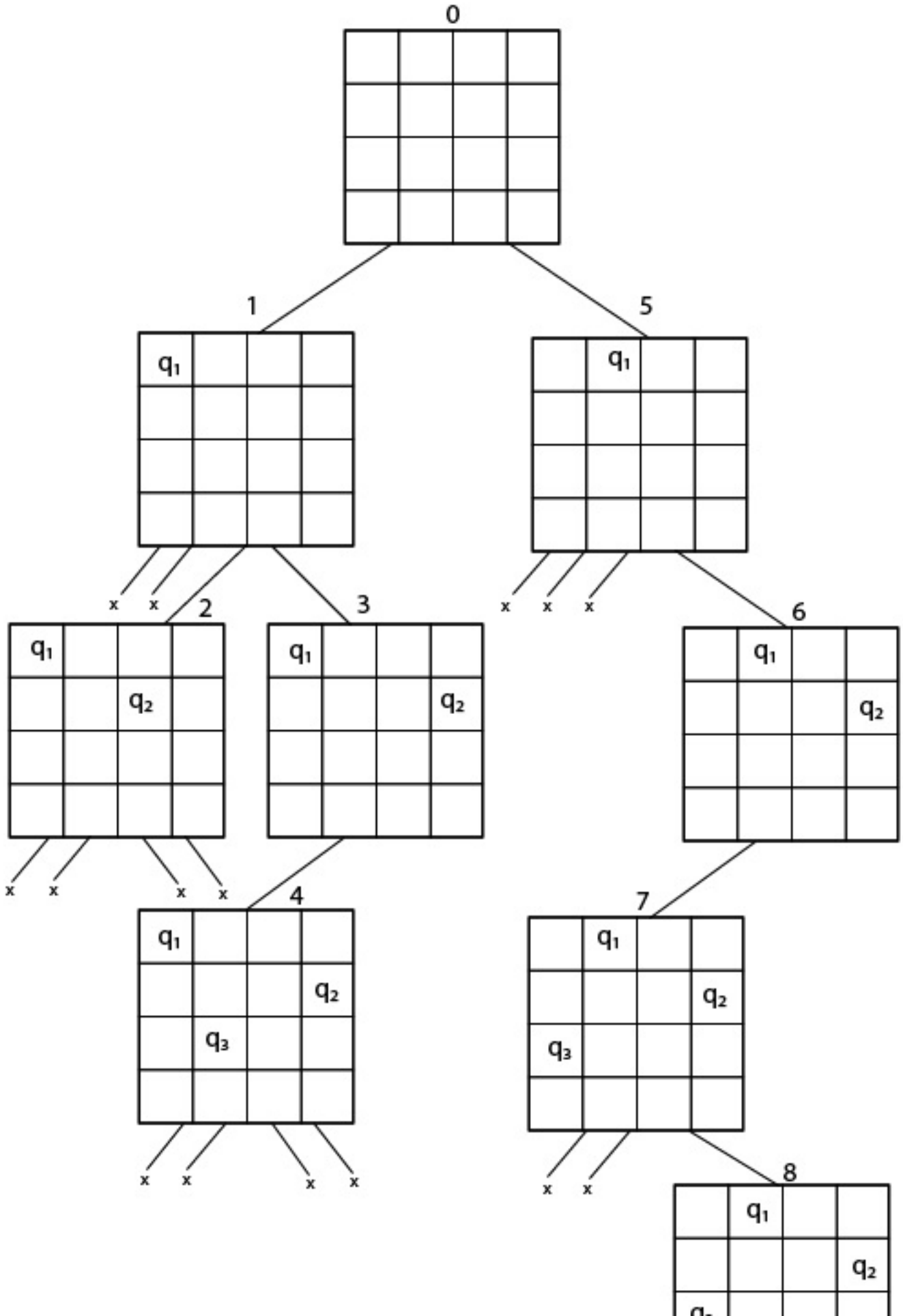
Follow the steps below to implement the idea:

- Make a recursive function that takes the state of the boa parameter.
- Start in the topmost row.
- If all queens are placed, return **true**
- For every row.
  - Do the following for each column in current row.
    - If the queen can be placed safely in this column
      - Then mark this **[row, column]** as part of the so placing queen here leads to a solution.
    - If placing the queen in **[row, column]** leads to a so
    - If placing queen doesn't lead to a solution then un back and try other columns.
- If all columns have been tried and nothing worked, retur

Time Complexity: O(N!)

Auxiliary Space: O(N)

The implicit tree for 4 - queen problem for a solution (2, 4, 1, 3) is as follows:



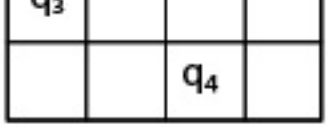


Fig shows the complete state space for 4 - queens problem. But we can use back

chessboard, in such a way that no two queens attack eac

- To solve this problem generally Backtracking algorithm o

Backtracking -

- In backtracking, start with one possible move out of man problem.
- If it can solve the problem with the selected move then it backtrack and select some other move and try to solve it
- If none of the moves works out the claim that there is no

Algorithm for N-Queens Problem u

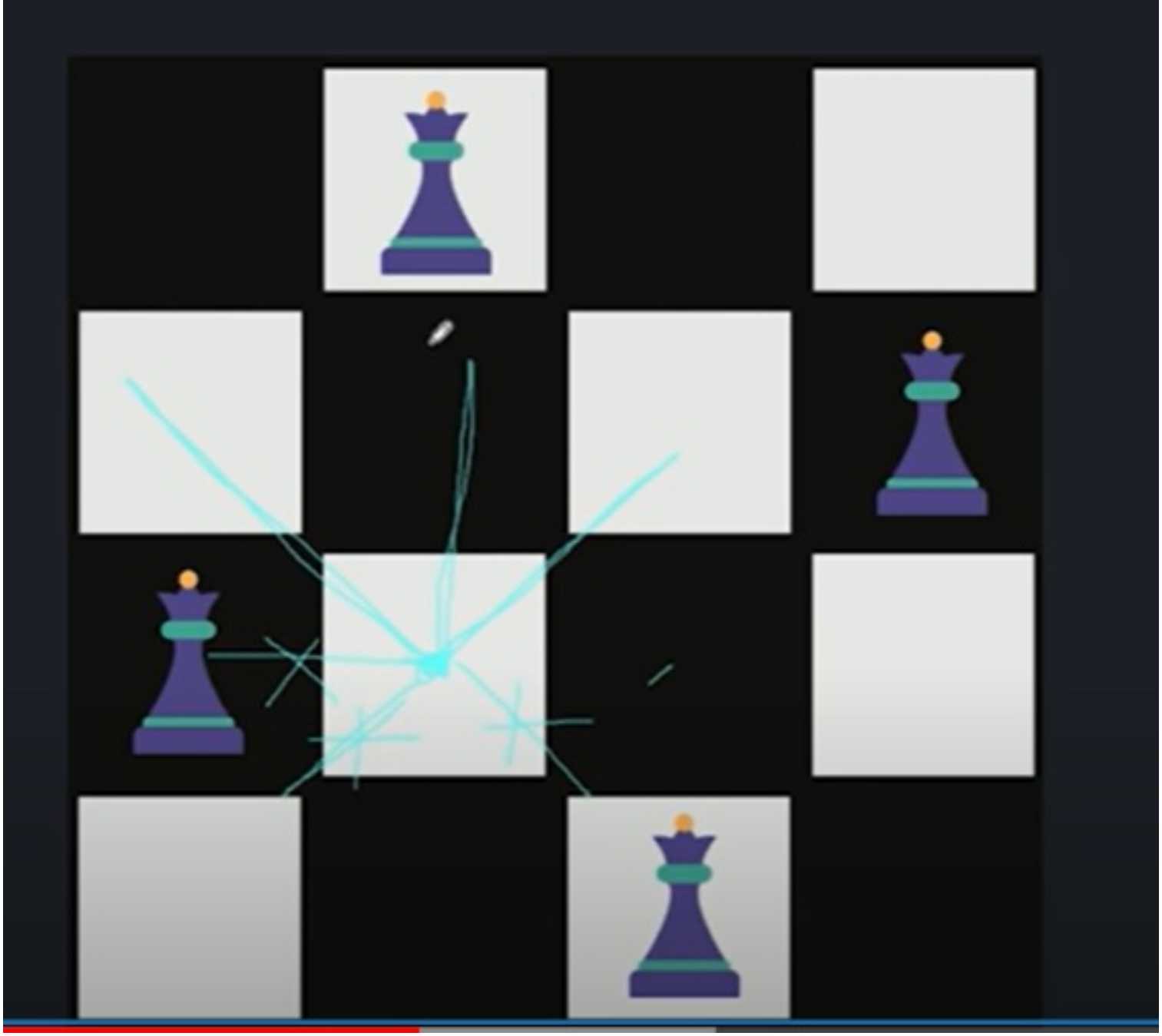
Step 1 - *Place the queen row-wise, starting from the left-most c*

Step 2 - *If all queens are placed then return true and print the s*

Step 3 - *Else try all columns in the current row.*

- ***Condition 1 -*** *Check if the queen can be placed safely in t [Row, Column] in the solution matrix as 1 and try to chec placing the queen here leads to a solution or not.*
- ***Condition 2 -*** *If placing the queen [Row, Column] can lead the solution for each queen's position.*
- ***Condition 3 -*** *If placing the queen cannot lead to the solu the solution matrix as 0, BACKTRACK, and go back to con*

Step 4 - *If all the rows have been tried and nothing worked, retu*



8 queen problem

Read | Discuss | Courses | Practice

The eight queens problem is the problem of placing eight q
that none of them attack one another (no two are in the sa
generally, the n queens problem places n queens on an n×
solutions for the problem. Backtracking | Set 3 (N Queen P
Queen Problem) You can find detailed solutions at
http://en.literateprograms.org/Eight_queens_puzzle_(C)

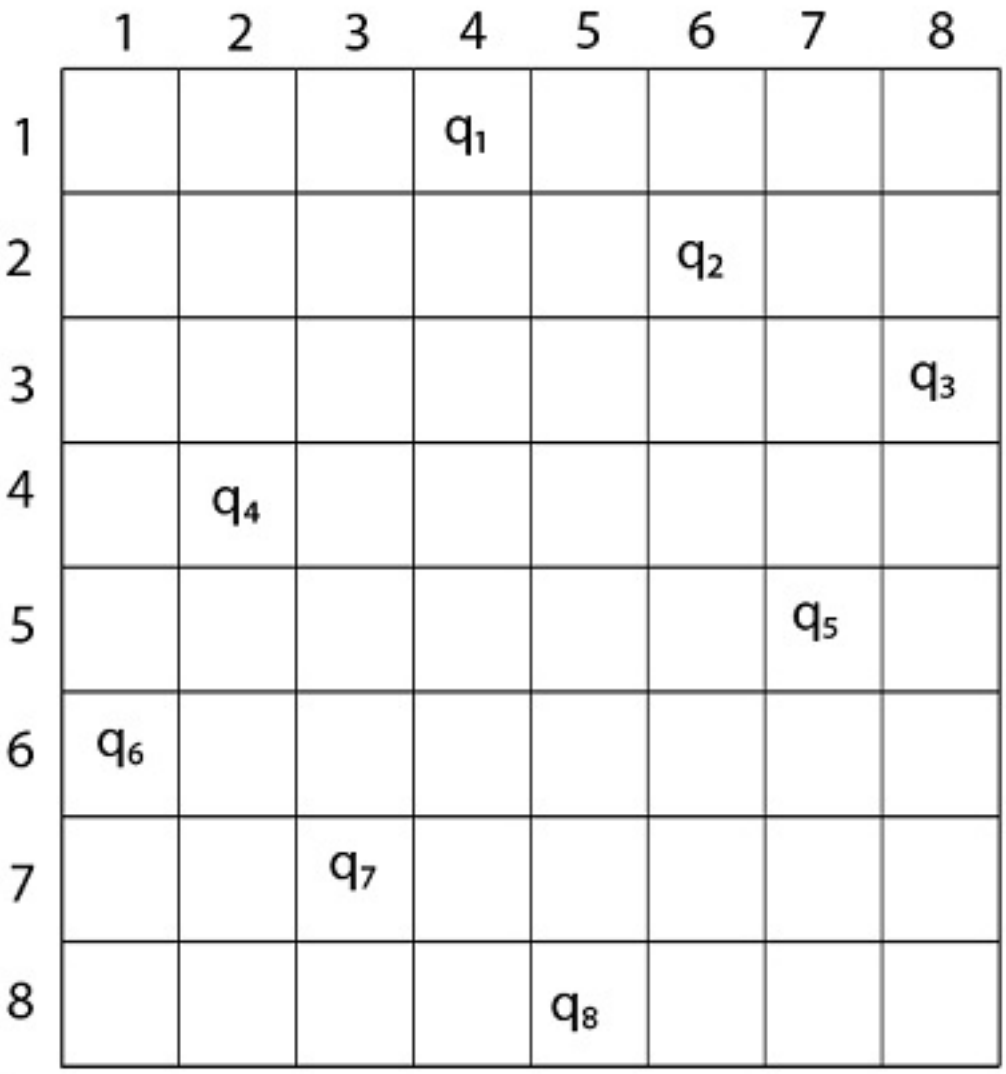
Explanation:

- This **pseudocode uses a backtracking algorithm** to find
  which consists of placing 8 queens on a chessboard in su
  threaten each other.
- The algorithm starts by placing a queen on the first colu
  column and places a queen in the **first safe** row of that c
- If the algorithm reaches the 8th column and all **queens a**
  prints the board and returns true.
  If the algorithm is unable to place a queen in a safe posit
  to the previous column and tries a different row.
- The "isSafe" function **checks** if it is safe to place a queen
  checking if there are any queens in the same row, diagon
- It's worth to notice that this is just a high-level **pseudocc**
  depending on the specific implementation and language

Time Complexity : O((m + q) log^2 n)

Space Complexity : O((m + q) log n)

One possible solution for 8 queens problem is shown in fig:



Thus, the solution **for** 8 -queen problem **for** (4, 6, 8, 2, 7, 1, 3, 5).

f two queens are placed at position (i, j) and (k, l).

Then they are on same diagonal only **if** (i - j) = k - l or i + j = k + l.

The first equation implies that j - l = i - k.

The second equation implies that j - l = k - i.

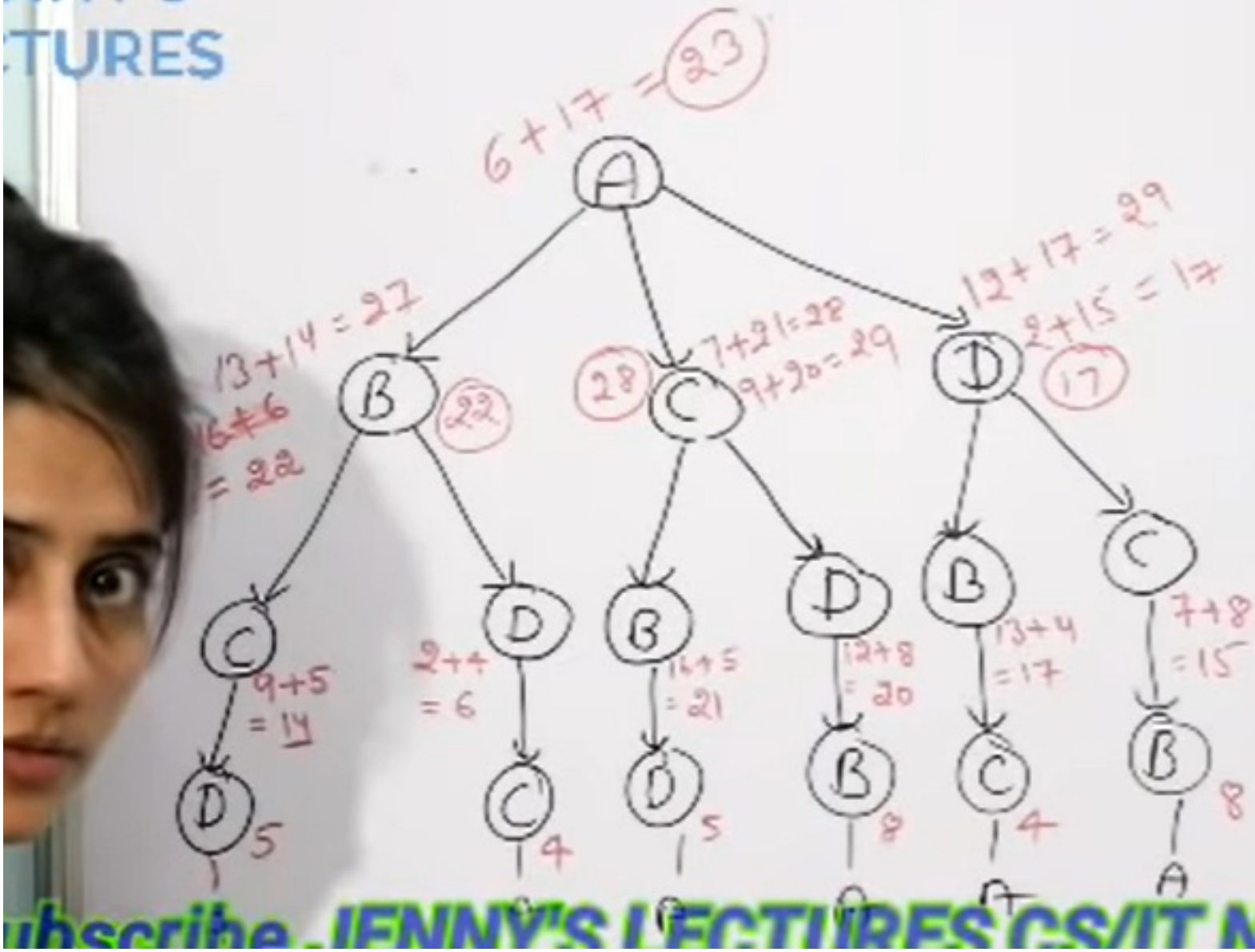
Therefore, two queens lie on the duplicate diagonal **if** and only **if** |j-l|=|i-k|

Parameter	Backtracking	
Approach	Backtracking is used to find all possible solutions available to a problem. When it realises that it has made a bad choice, it undoes the last choice by backing it up. It searches the state space tree until it has found a solution for the problem.	Branc it reali solu
Traversal	Backtracking traverses the state space tree by DFS(Depth First Search) manner.	Bran
Function	Backtracking involves feasibility function.	
Problems	Backtracking is used for solving Decision Problem.	Bra
Searching	In backtracking, the state space tree is searched until the solution is obtained.	In Bra whe
Efficiency	Backtracking is more efficient.	
Applications	Useful in solving N-Queen Problem, Sum of subset, Hamilton cycle problem, graph coloring problem	Useful
Solve	Backtracking can solve almost any problem. (chess, sudoku, etc).	

Travelling sales man

Travelling Salesman Problem

$$g(i,S) = \min_{j \in S} \left(w(i,j) + g(j, S-j) \right)$$



Travelling Salesman Problem

$$g(i,S) = \min_{j \in S} \{ w(i,j) + g(j, S-\{j\}) \}$$

$i = A$

$$g(A,\{B,C,D\}) = \min \begin{cases} w(A,B) + g(B,\{C,D\}) \\ w(A,C) + g(C,\{B,D\}) \\ w(A,D) + g(D,\{B,C\}) \end{cases}$$

$$g(B,\{C,D\}) = \begin{cases} w(B,C) + g(C,\{D\}) = 13+14 = \\ w(B,D) + g(D,\{C\}) = 16+6 = \end{cases}$$

$$g(C,\{D\}) = w(C,D) + g(D,\phi)$$
$$= 9+5 = \boxed{14}$$

$$g(D,\{C\}) = w(D,C) + g(C,\phi)$$
$$= 2+4 = 6$$

Travelling Salesman Problem

$$g(i,S) = \min_{j \in S} [w(i,j) + g(j, S-j)]$$

$i = A$

$$g(A,\{B,C,D\}) = \min[w(A,B) + g(B,\{C,D\}) = 1$$
$$w(A,C) + g(C,\{B,D\}) = 1$$
$$w(A,D) + g(D,\{B,C\}) = 6$$

$$g(\ ,\{B,C\}) = \min[w(D,B) + g(B,\{C\})] = 13+17$$
$$w(D,C) + g(C,\{B\}) = 2+15$$
$$= 17$$

$$g(B,\{C\}) = \min[w(B,C) + g(C,\phi)]$$
$$= 13+4 = 17$$

$$g(C,\{B\}) = \min[w(C,B) + g(B,\phi)]$$
$$= 7+8 = 15$$

Travelling Salesman Problem

$$g(i,S) = \min_{j \in S} [w(i,j) + g(j, S-j)]$$

$i = A$

$$g(A,\{B,C,D\}) = \min \begin{cases} w(A,B) + g(B,\{C,D\}) \\ w(A,C) + g(C,\{B,D\}) \\ w(A,D) + g(D,\{B,C\}) \end{cases}$$

$= 16+2$
$= 38$

$= 11+28$
$= 39$

$= 6+17$
$= 23$

$g(B,\{C,D\}) = \min \begin{cases} w(B,C) + g(C,\{D\}) \\ w(B,D) + g(D,\{C\}) \end{cases} = 12+17 =$

$= 9+15$
$= 24$

$= 13+15$
$= 28$

$g(C,\{D\}) = \min [w(C,D) + g(D,\phi)]$
$= 9+4 = 13$

$g(D,\{C\}) = \min [w(D,C) + g(C,\phi)]$
$= 13+4 = 17$

$g(C,\{B\}) = \min [w(C,B) + g(B,\phi)]$
$= 7+8 = 15$

$i =$
$S =$

$A \to D \to C \to B$ $= 23$

Naive Solution:

1) Consider city 1 as the starting and ending point.

2) Generate all (n-1)! Permutations of cities.

3) Calculate the cost of every permutation and keep tra

4) Return the permutation with minimum cost.

Time Complexity: Θ(n!)

Travelling Salesperson Algorithm

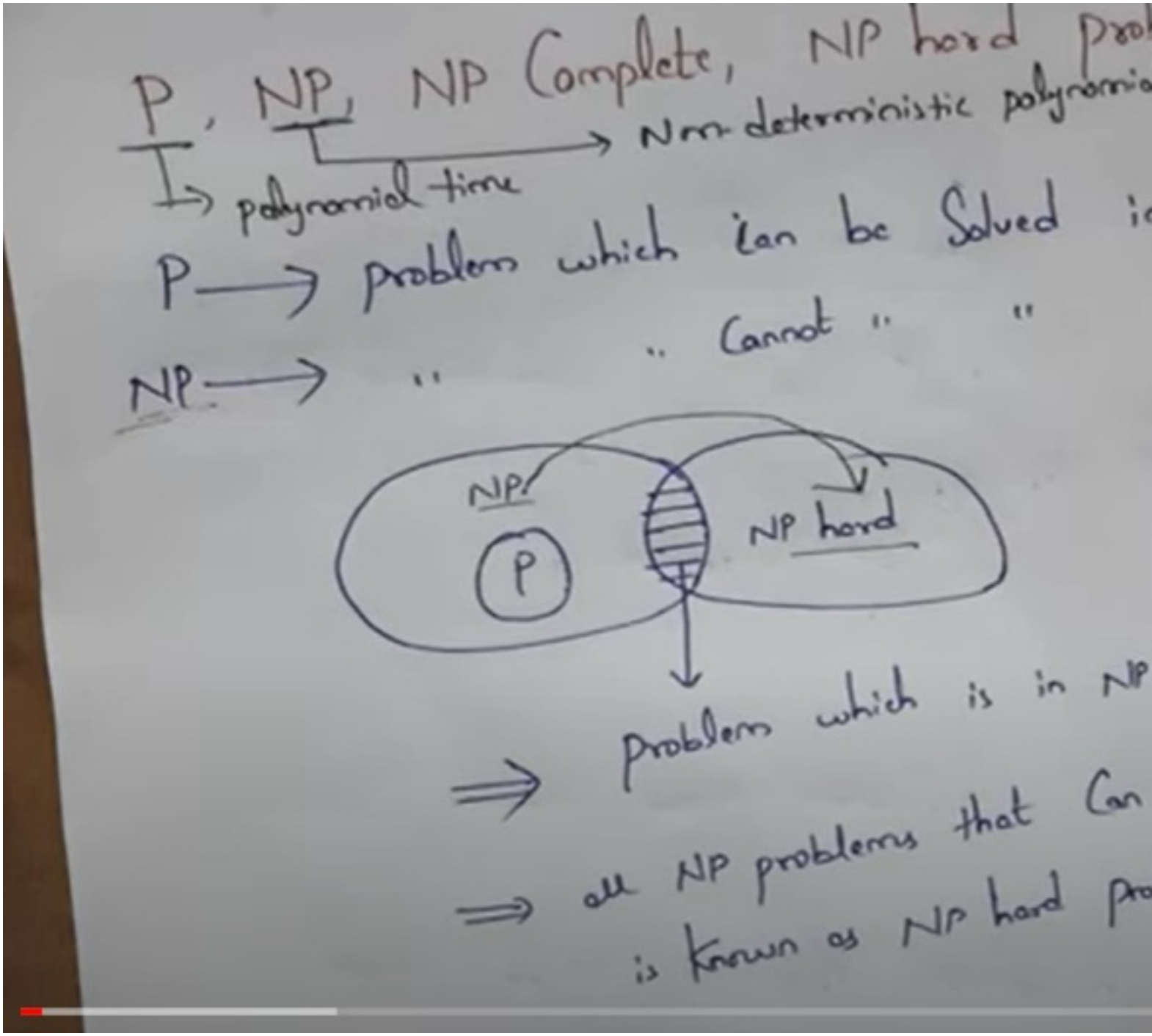
As the definition for greedy approach states, we need to solution locally to figure out the global optimal solution. algorithm are the graph G {V, E}, where V is the set of v of edges. The shortest path of graph G starting from one the same vertex is obtained as the output.

Algorithm

- Travelling salesman problem takes a graph G {V, E} declare another graph as the output (say G') which the salesman is going to take from one node to ano
- The algorithm begins by sorting all the edges in the the least distance to the largest distance.
- The first edge selected is the edge with least distan two vertices (say A and B) being the origin node (sa
- Then among the adjacent edges of the node other t (B), find the least cost edge and add it onto the out
- Continue the process with further nodes making sur in the output graph and the path reaches back to th
- However, if the origin is mentioned in the given prob solution must always start from that node only. Let example problems to understand this better.

```
Algorithm: Traveling-Salesman-Problem
C ({1}, 1) = 0
for s = 2 to n do
   for all subsets S ∈ {1, 2, 3, … , n} of size s and containing 1
      C (S, 1) = ∞
for all j ∈ S and j ≠ 1
      C (S, j) = min {C (S – {j}, i) + d(i, j) for i ∈ S and i ≠ j}
Return minj C ({1, 2, 3, …, n}, j) + d(j, i)
```

NP-Hard and NP-Complete Problems



Depending on Computing time

Algorithms

Polynomial time

takes less time

Ex: linear Search. $\div n$

binary Search $\div \log n$

Exponential

takes

Ex ① 0/1 Kn

② traveling

Problems which don't be Solved in polyn

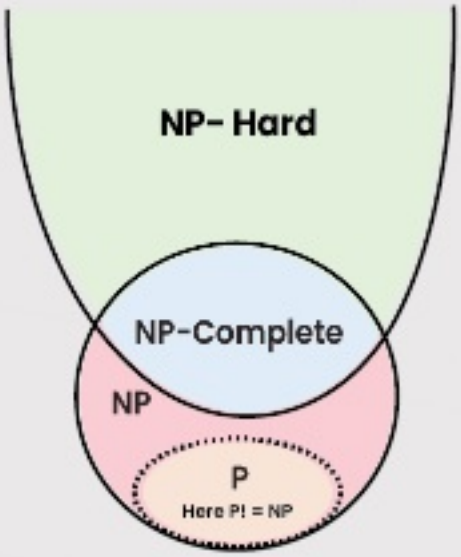
classified into two types

Problems which don't be Solved in
classified into two types

NP Complete:
A problem that is NP Complete has the prop
polynomial time if and only if all other
in polynomical time.

NP hard: If an NP hard problem can be Solved
problems can be Solved in polynomial

COMPLEXITY
Classes



In computer science, there exist some problems whose solutions are
Classes. In complexity theory, a Complexity Class is a set of problem
on how much time and space they require to solve problems and ver
resources required to solve a problem.

Types of Complexity Classes

This article discusses the following complexity classes:

1. **P Class**
2. **NP Class**
3. **CoNP Class**
4. **NP-hard**
5. **NP-complete**

P Class

The P in the P class stands for **Polynomial Time.** It is the c
can be solved by a deterministic machine in polynomial tim

Features:

- The solution to **P problem**s is easy to find.
- **P** is often a class of computational problems that are so
  theory as well as in practice. But the problems that can

This class contains many problems:

1. Calculating the greatest common divisor.
2. Finding a maximum matching.
3. Merge Sort

NP Class

The NP in NP class stands for **Non-deterministic Polynomial Time**. non-deterministic machine in polynomial time.

Features:

- The solutions of the NP class are hard to find since they are being easy to verify.
- Problems of NP can be verified by a Turing machine in polynomia

Example:

Let us consider an example to better understand the **NP class**. Supp having unique employee **IDs**. Assume that there are **200** rooms avai together, but the CEO of the company has the data of some employe This is an example of an **NP** problem. Since it is easy to check if the satisfactory or not i.e. no pair taken from the coworker list appears o scratch seems to be so hard as to be completely impractical.

It indicates that if someone can provide us with the solution to the p time. Thus for the **NP** class problem, the answer is possible, which c

This class contains many problems that one would like to be able to

1. Boolean Satisfiability Problem (SAT).
2. Hamiltonian Path Problem.
3. Graph coloring.

NP-hard class

An NP-hard problem is at least as hard as the hardest problem in NP and it is a reduces to NP-hard.

Features:

- All NP-hard problems are not in NP.
- It takes a long time to check them. This means if a solution for an NP-hard p whether it is right or not.
- A problem A is in NP-hard if, for every problem L in NP, there exists a polyn

Some of the examples of problems in Np-hard are:

1. **Halting problem.**
2. **Qualified Boolean formulas.**
3. **No Hamiltonian cycle.**

NP-complete class

A problem is NP-complete if it is both NP and NP-hard. NP-complete problems

Features:

- NP-complete problems are special as any problem in NP class can be transf polynomial time.
- If one could solve an NP-complete problem in polynomial time, then one co

Some example problems include:

1. Hamiltonian Cycle.
2. Satisfiability.
3. Vertex cover.

Complexity Class	Characteristic feature
P	Easily solvable in polynomial time.
NP	Yes, answers can be checked in polynomial tim
Co-NP	No, answers can be checked in polynomial tim
NP-hard	All NP-hard problems are not in NP and it takes a long time
NP-complete	A problem that is NP and NP-hard is NP-compl

UNIT 1

Recursion

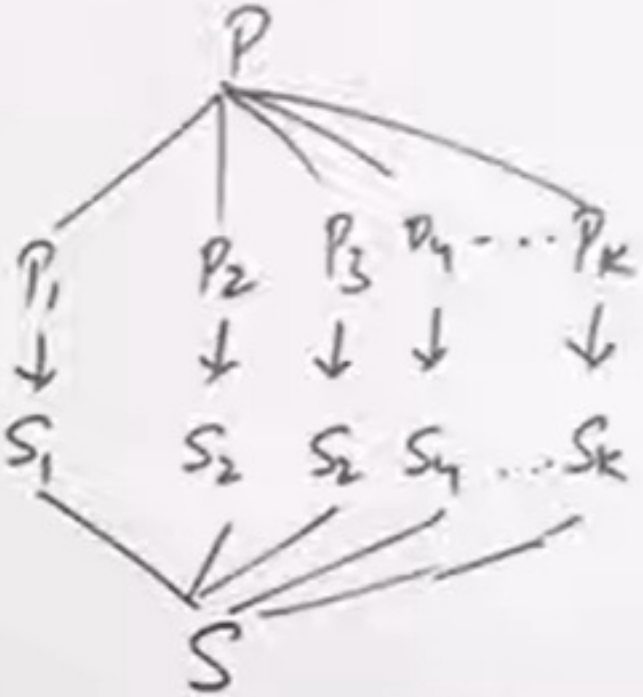
The process in which a function calls itself directly or indirectly
the function which calls itself is known as a recursive function.
recursion here. In function, there must be a base condition to te
the function.

The time complexity of recursion

The time complexity of recursion depends on the number of t
a function calls itself two times then its time complexity is O(
then its time complexity is O(3 ^ N) and so on.

"Divide and Conquer"

```
DAC(P)
{
  if (Small (P))
  {
    S(P);
  }
  else
  {
    divide P into P1 P2 P3 .... PK
    Apply DAC(P1), DAC(P2) ..... DAC(PK)
    Combine( DAC(P1), DAC(P2) .... DAC(PK)
  }
}
```



Recurrence Relation

Recurrence Relation

10 20 30

i=1

BS(a, i, J, x) — 30

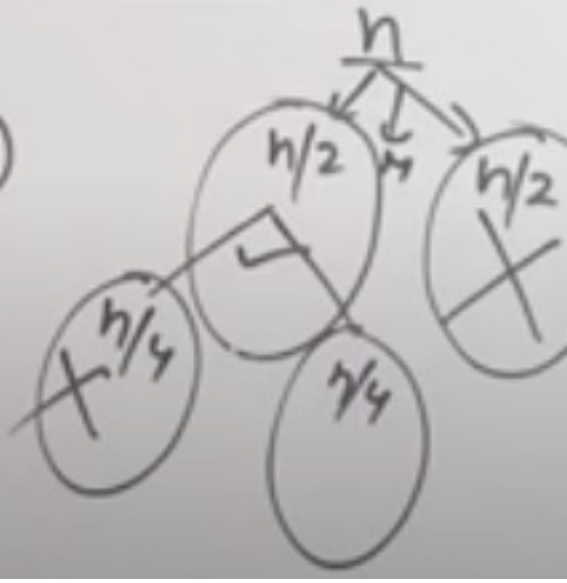
```
mid = (i+J)/2
if (a[mid] == x)
    return (mid);
else if (a[mid] > x)
    → BS(a, i, mid-1, x)
else
    → BS(a, mid+1, J, x)
```

$\frac{1+7}{2} = \frac{8}{2} = 4$

O(c)

10	20	30
1	2	3
i		J

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



Recurrence Relation

A recurrence is an equation or inequality that describes a functi

Recurrence Relation means to obtain a function defined on the na

For Example, the Worst Case Running Time T(n) of the MERGE SO

$$T(n) = \begin{cases} \theta(1) & \text{if } n=1 \\ 2T\left(\frac{n}{2}\right) + \theta(n) & \text{if } n>1 \end{cases}$$

There are four methods for solving Recurrence:

1. Substitution Method
2. Iteration Method
3. Recursion Tree Method
4. Master Method

Substitution Method:

We make a guess for the solution and then we use ma
the guess is correct or incorrect.

For example consider the recurrence T(n) = 2T(n/2)

We guess the solution as T(n) = O(nLogn). Now we
guess.

We need to prove that T(n) <= cnLogn. We can ass
smaller than n.

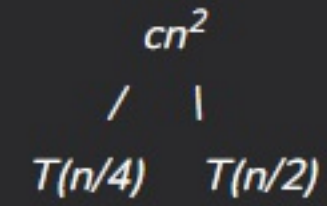
$$
\begin{aligned}
T(n) &= 2T(n/2) + n \\
&<= 2cn/2Log(n/2) + n \\
&= cnLogn - cnLog2 + n \\
&= cnLogn - cn + n \\
&<= cnLogn
\end{aligned}
$$

Recurrence Tree Method:

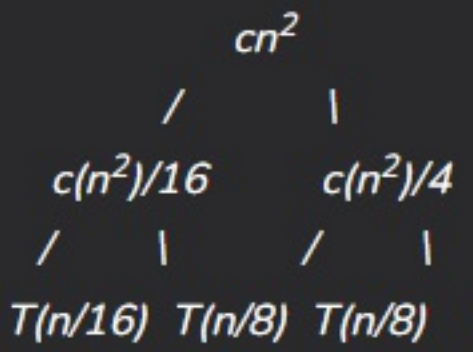
In this method, we draw a recurrence tree and calculate the time taken by eve
all levels. To draw the recurrence tree, we start from the given recurrence and
pattern is typically arithmetic or geometric series.

For example, consider the recurrence relation

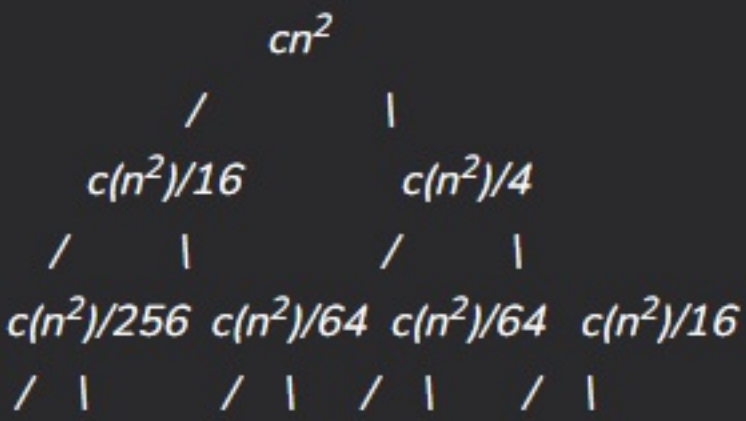
$T(n) = T(n/4) + T(n/2) + cn^2$



If we further break down the expression T(n/4) and T(n/2), we get the following recursion tree.



Breaking down further gives us following



To know the value of T(n), we need to calculate the sum of tree nodes level by level. If we sum the above tree level by level,

we get the following series T(n) = c(n^2 + 5(n^2)/16 + 25(n^2)/256) + ...
The above series is a geometrical progression with a ratio of 5/16.

To get an upper bound, we can sum the infinite series. We get the sum as (n^2)

Master Method:

Master Method is a direct way to get the solution. The master method works c
recurrences that can be transformed into the following type.

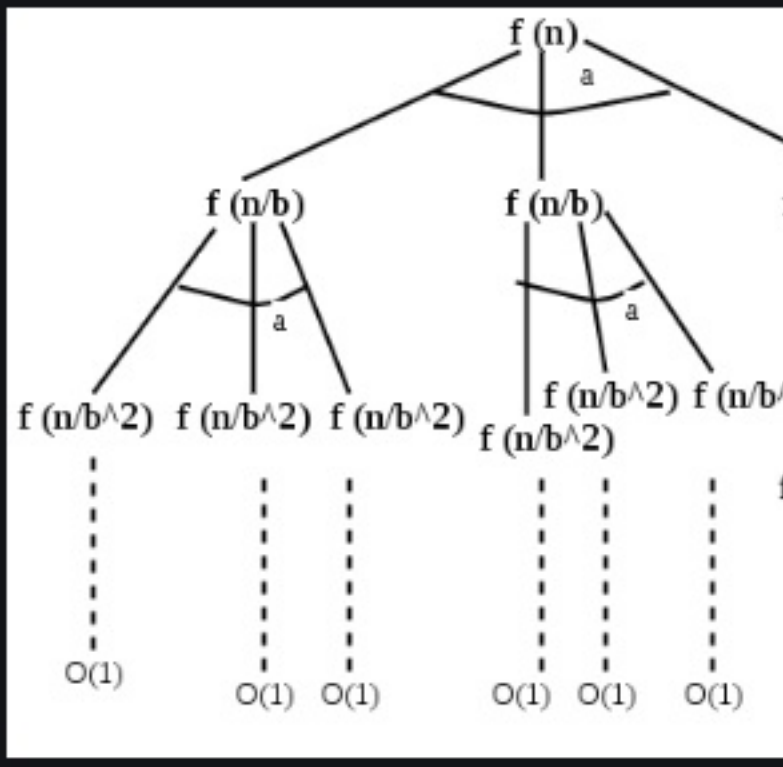
$T(n) = aT(n/b) + f(n)$ where $a >= 1$ and $b > 1$

There are the following three cases:

- If $f(n) = O(n^c)$ where $c < \text{Log}_b a$ then $T(n) = \Theta(n^{\text{Log}_b a})$
- If $f(n) = \Theta(n^c)$ where $c = \text{Log}_b a$ then $T(n) = \Theta(n^c \text{Log } n)$
- If $f(n) = \Omega(n^c)$ where $c > \text{Log}_b a$ then $T(n) = \Theta(f(n))$

How does this work?

The master method is mainly derived from the recurrence tree method. If we can see that the work done at the root is f(n), and work done at all leaves is Θ tree is $Log_b n$



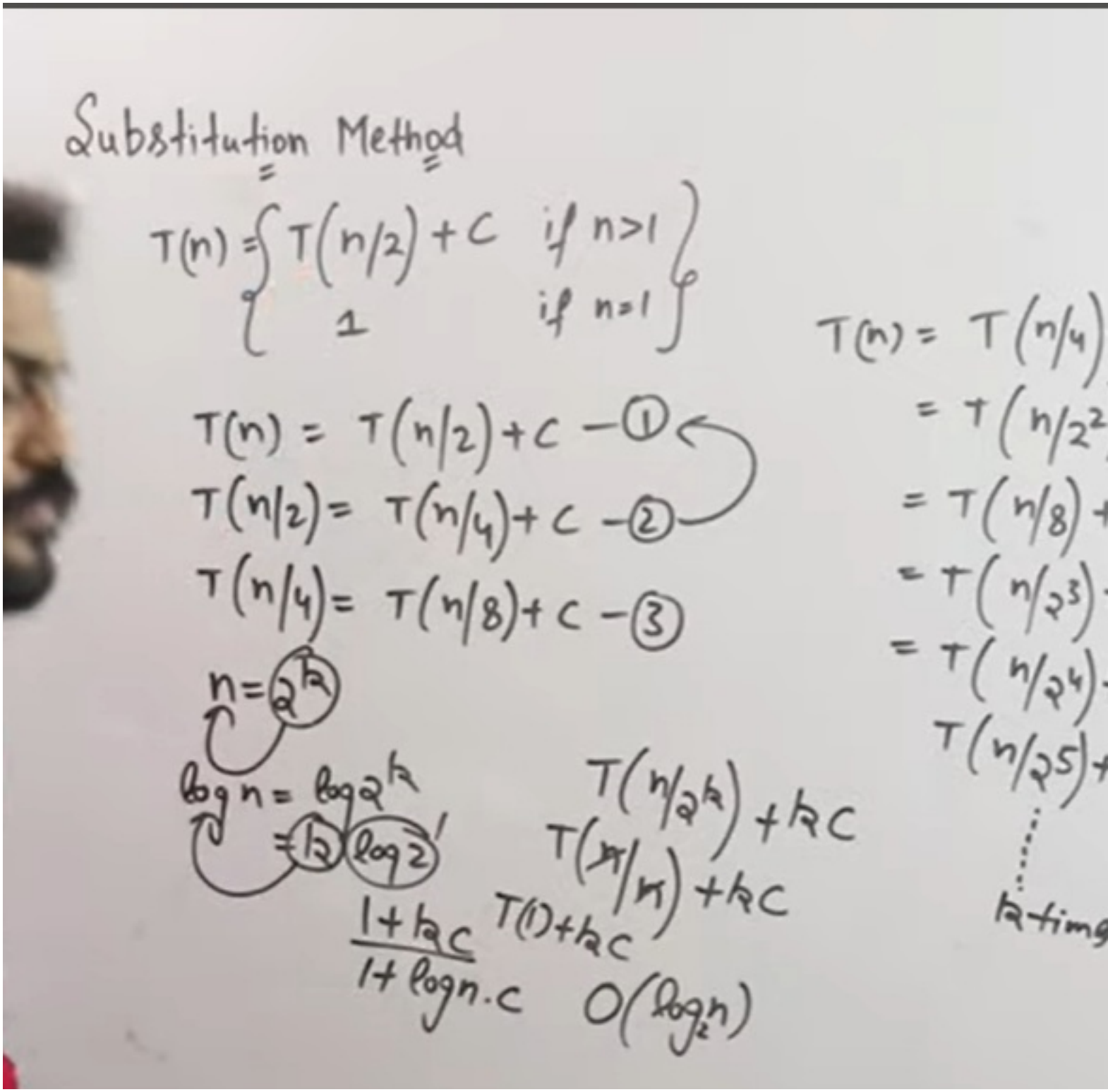
In the recurrence tree method, we calculate the total work done. If the work d dominant part, and our result becomes the work done at leaves (Case 1). If w then our result becomes height multiplied by work done at any level (Case 2) our result becomes work done at the root (Case 3).

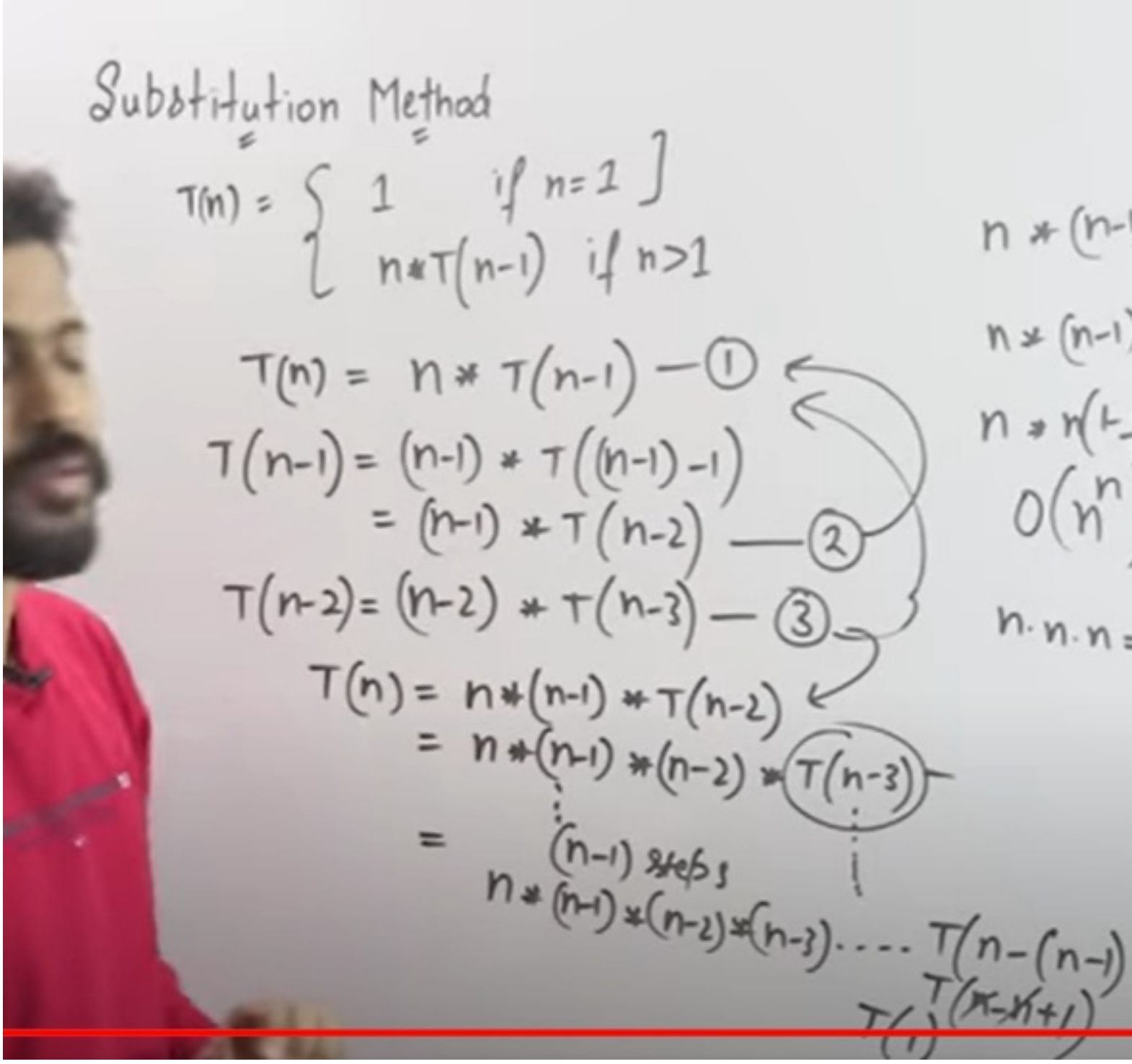
Examples of some standard algorithms whose time complexity can be eva

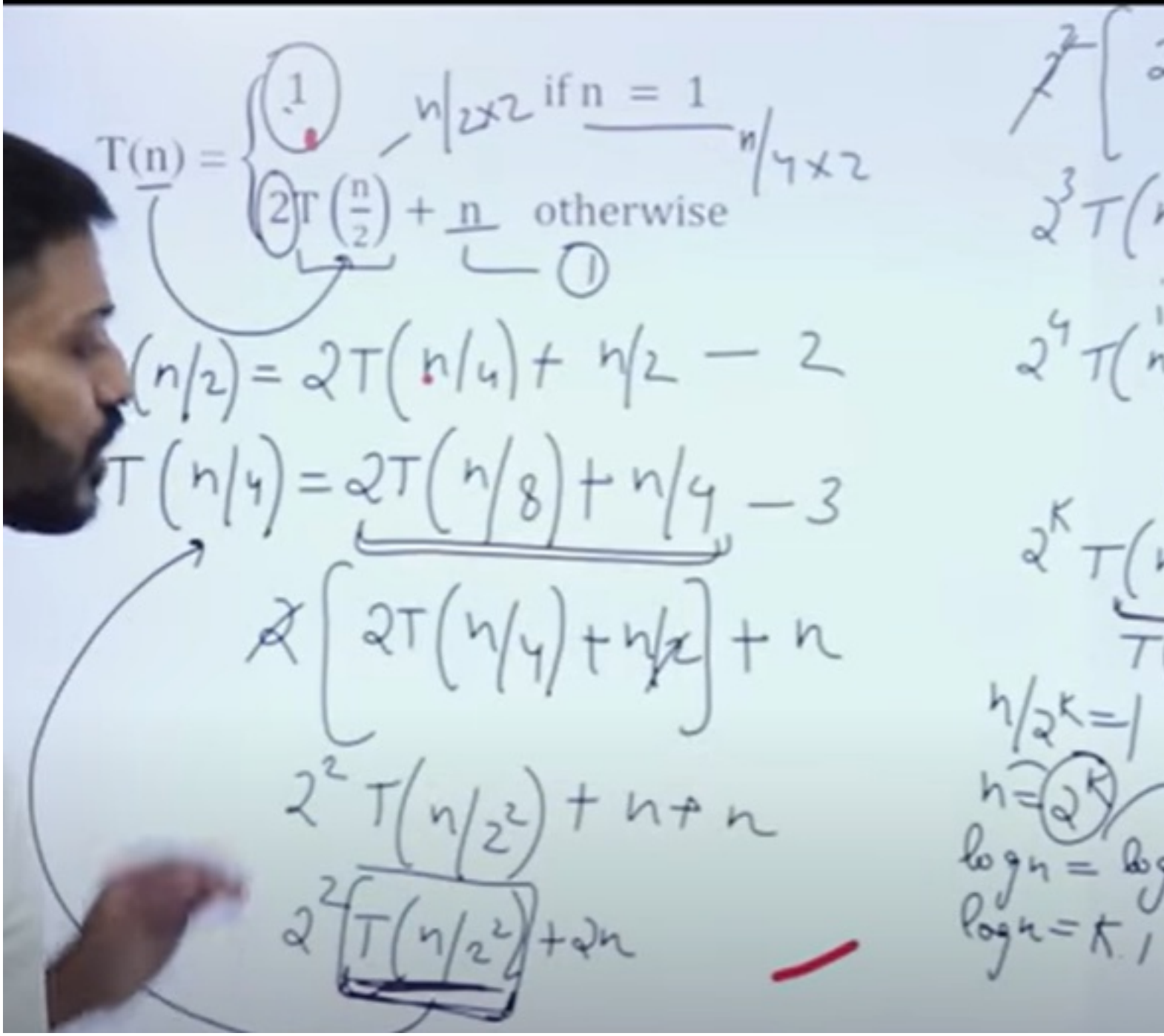
- Merge Sort: T(n) = 2T(n/2) + Θ(n). It falls in case 2 as c is 1 and $Log_b a]$ is a
- Binary Search: T(n) = T(n/2) + Θ(1). It also falls in case 2 as c is 0 and Log_b

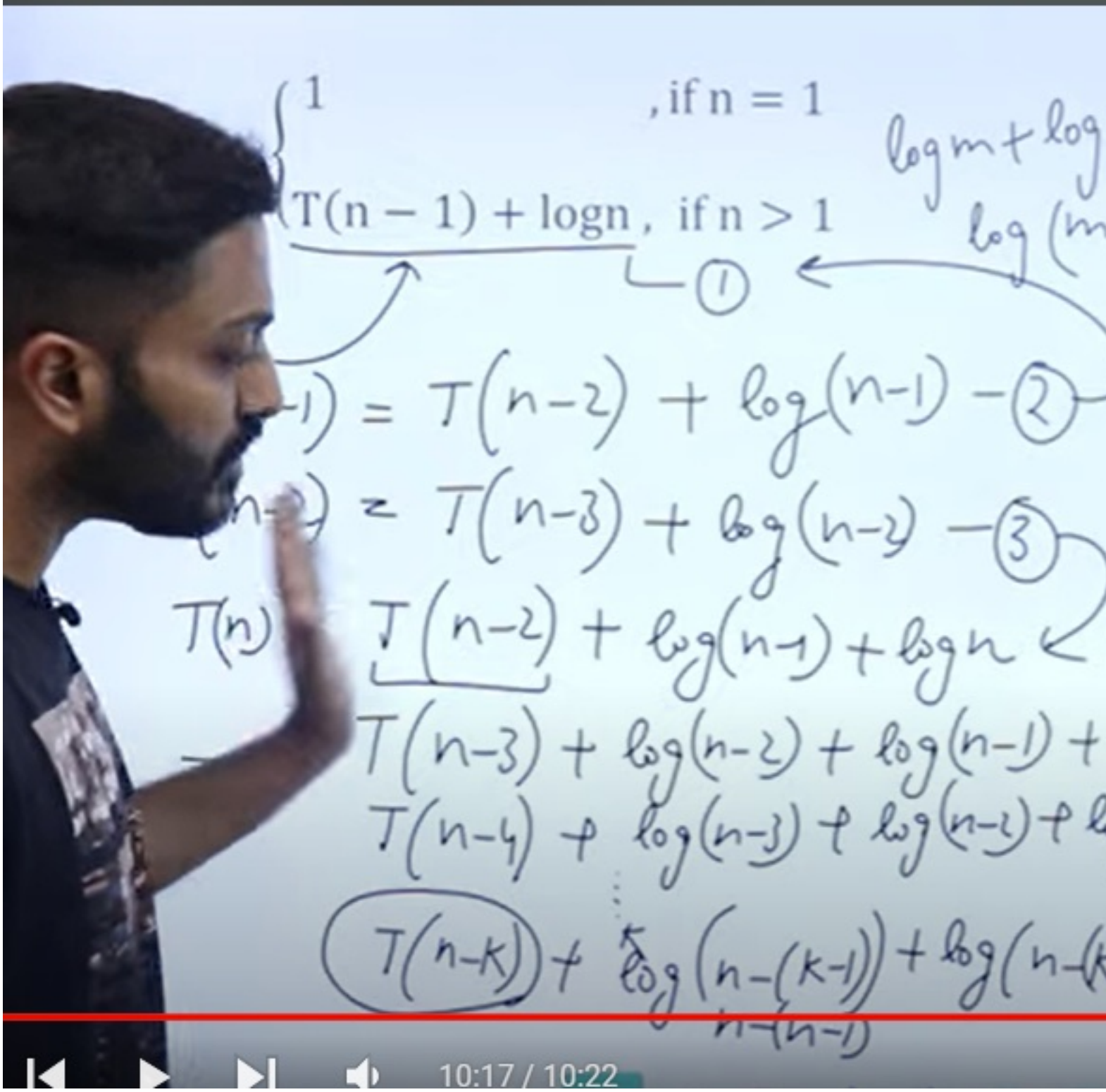
Notes:

- It is not necessary that a recurrence of the form T(n) = aT(n/b) + f(n) can be have some gaps between them. For example, the recurrence T(n) = 2T(n/2)
- Case 2 can be extended for f(n) = $\Theta(n^c Log^k n)$
  If f(n) = $\Theta(n^c Log^k n)$ for some constant k >= 0 and c = $Log_b a$, then T(n) = Θ(n




Substitution Method
T(n) = { 1 if n=1 ; n*T(n-1) if n>1 }
T(n) = n * T(n-1) — ①
T(n-1) = (n-1) * T((n-1)-1)
= (n-1) * T(n-2) — ②
T(n-2) = (n-2) * T(n-3) — ③
T(n) = n*(n-1) * T(n-2)
= n*(n-1)*(n-2)*T(n-3)
(n-1) steps
n*(n-1)*(n-2)*(n-3)..... T(n-(n-1))
T(1)
T(n-k+1)


T(n) = { 1 if n = 1
2T(n/2) + n otherwise
T(n/2) = 2T(n/4) + n/2 − 2
T(n/4) = 2T(n/8) + n/4 − 3
2[2T(n/4) + n/2] + n
2^2 T(n/2^2) + n + n
2^2 [T(n/2^2) + 2n]
n/2^k = 1
n = 2^k
log n = log
log n = k



Master Theorem

* Master Method (Theorem)

→ $T(n) = aT(n/b) + f(n)$

$\boxed{a \geq 1}$, $\boxed{b > 1}$

→ Solution is:

$$T(n) = n^{\log_b a}[U(n)] \checkmark$$

→ U(n) depends on R(n)

→ $R(n) = \frac{f(n)}{n^{\log_b a}} = \frac{n^2}{n^{\log_2 8}} = \frac{n^2}{n^3} = \frac{1}{n} = n^{-1}$

→ Relation between R(n) and U(n) is:

if R(n)	U(n)
→ n^n, $n>0$	$O(n^n)$
→ n^n, $n<0$	$O(1)$
→ $(\log n)^i$, $i \geq 0$	$\frac{(\log_2 n)^{i+1}}{i+1}$

1) $T(n) = 8T($

2) $T(n) = T($

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

$T(n) = 8$

$a = 8$

$T(n) = n^{a}$

$= n^{\log}$

$= n^3$

1) $T(n) = 8T(n/2) + n^2$

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

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

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

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

$= n^{\log_2 1}\, U(n) \Rightarrow n^0\, U(n) \Rightarrow U(n)$

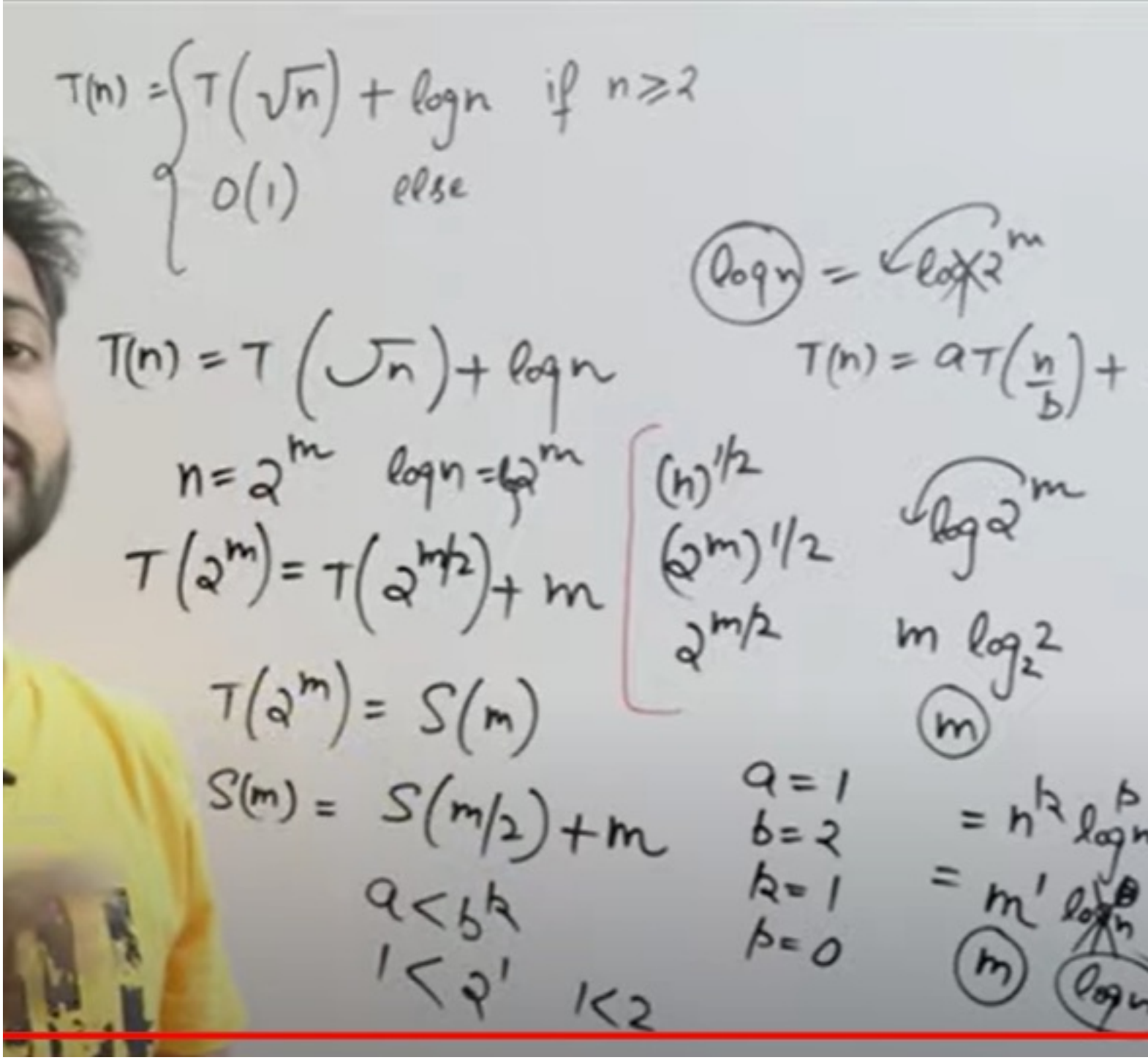
$k(n) = \dfrac{f(n)}{n^{\log_b a}} = (\log_2 n)^0 \cdot c$

$f(n)$	$U(n)$
$n>0$	$O(n^n)$
$n<0$	$O(1)$
$i \geq 0$	$(\log_2 n)^{i+1}$

$\dfrac{(\log_2 n)^{a+1}}{a+1} = (\log_2 n) \cdot c$

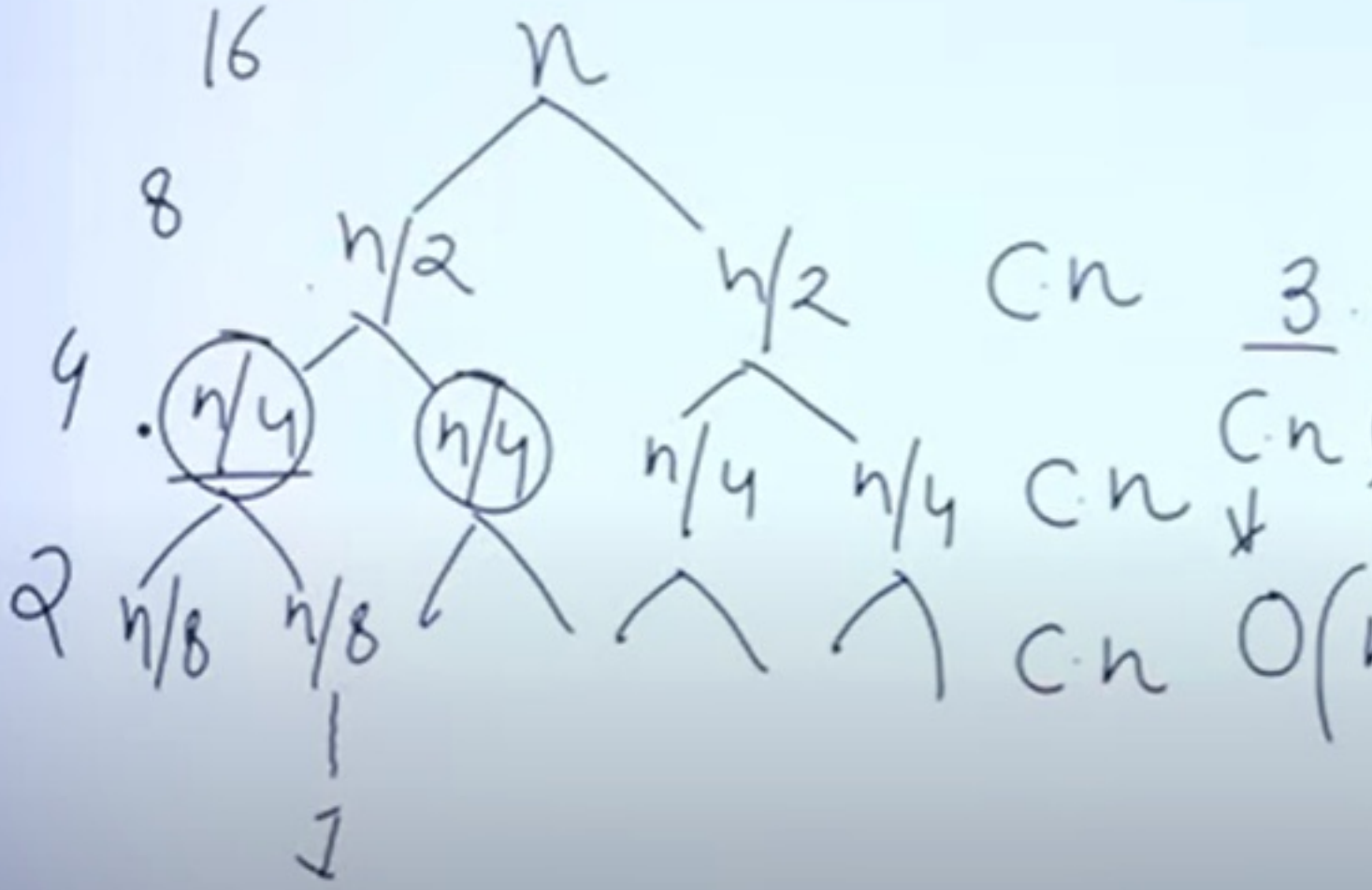
$= \log_2 n \cdot c$

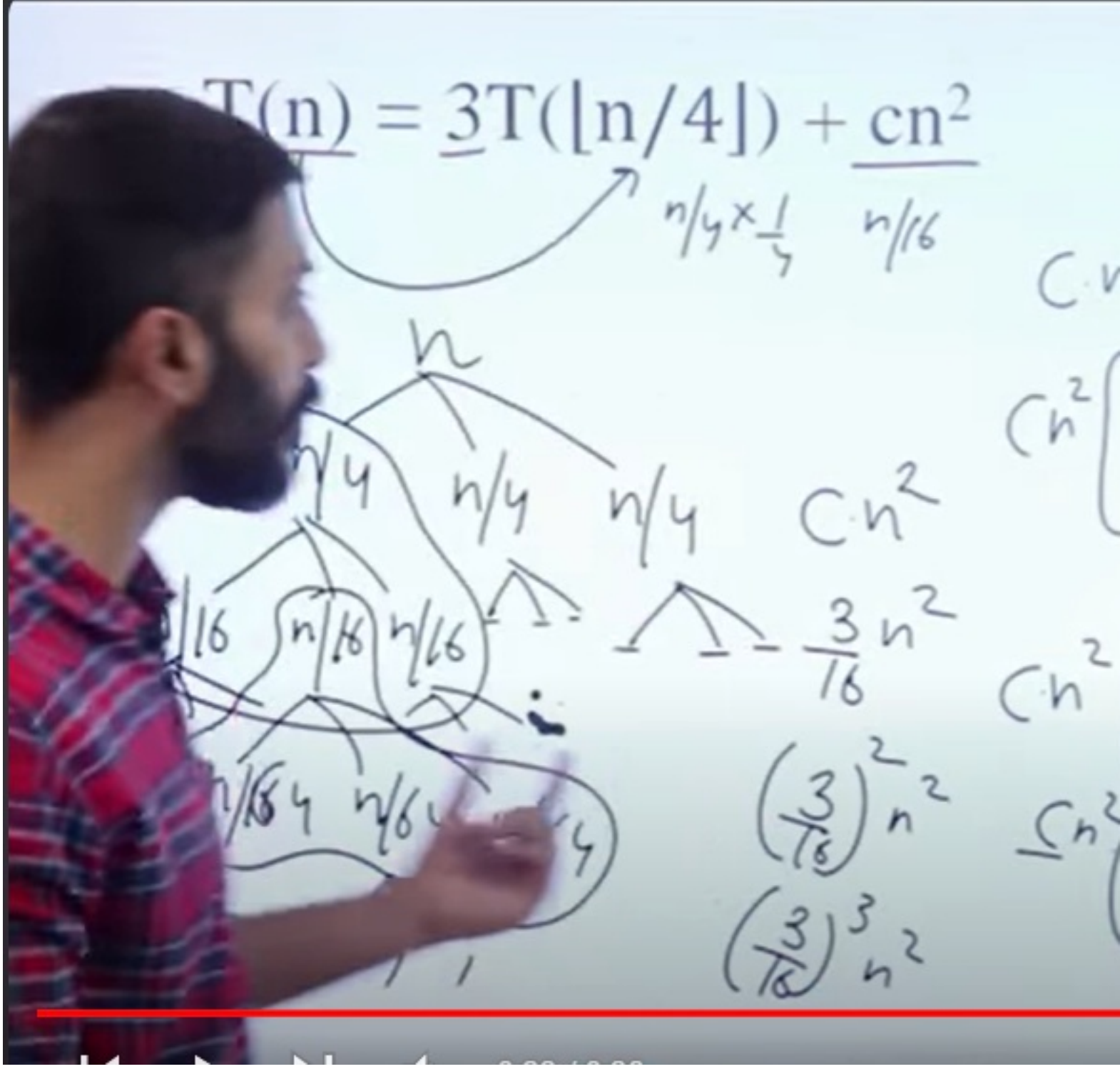
$= O(\log_2 n)$



Recursive Tree method

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




T(n) = 3T(⌊n/4⌋) + cn²
n/4 x 1/4
n/16
n
n/4
n/4
n/4
cn²
3/16 n²
(3/16)² n²
(3/16)³ n²
cn²
cn²

Name	Time Complexity		
	Best case	Average Case	Worst Case
Bubble	O(n)	-	$O(n^2)$
Insertion	O(n)	$O(n^2)$	$O(n^2)$
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$

Quick	O(log n)	O(n log n)	$O(n^2)$
Merge	O(n log n)	O(n log n)	O(n log n)
Heap	O(n log n)	O(n log n)	O(n log n)

Comparison between Merge and Quick Sort

'What is Algorithm'

→ Finite set of steps to solve a problem i

→ Analysis is process of comparing two algos

Priory Posteriori

APP. Exact

Ind. Dep.

Unifa ----

fact(n)

S1: Read A 1

S2: Read B 1

S3: Sum = A+B 1

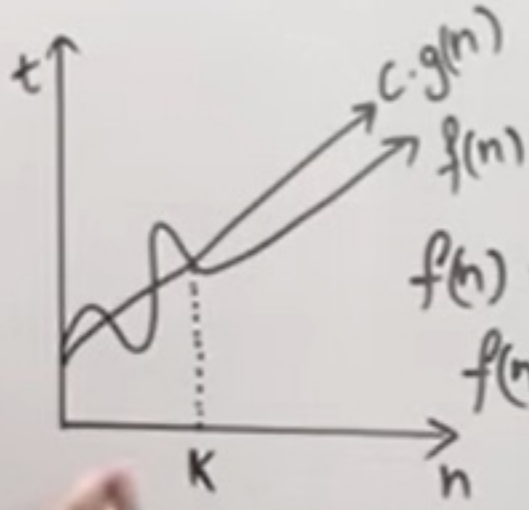
S4: Print (Sum) 1

(4)

```
main()
 int a,b,Sum;
 Scanf (a,b)
 Sum=a+b;   .4 sec
 Pf(Sum);   .3
            .?
```

Asymptotic Notation:

1) Big-oh (O)



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

$f(n) \leq c \cdot g(n)$

$c > 0$

$n \geq k$

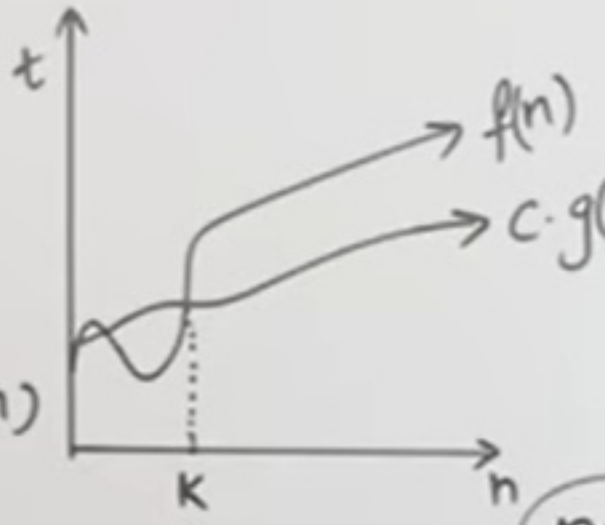
$k \geq 0$

Worst Case

Upper Bound (At most)

Least

2) Big-Omega (Ω)



→ Best Case

→ greatest Lower Bound (At Leas

$n \geq$

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

$2 \times (5)^2 + 5 \leq 3(5)^2$

$55 \leq 75$

$2n^2 + n \leq 3 \cdot n^2$

$n \leq n^2$

$f(n) = O(\ )$

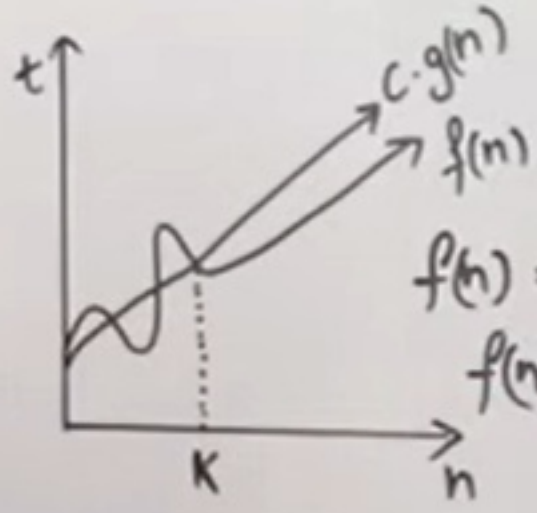
$2n^2 + n \leq c \cdot g$

$15n$

$n \geq 1$

Asymptotic Notation:

1) Big-oh (O)



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

$f(n) \leq c \cdot g(n)$

$c > 0$

$n \geq k$

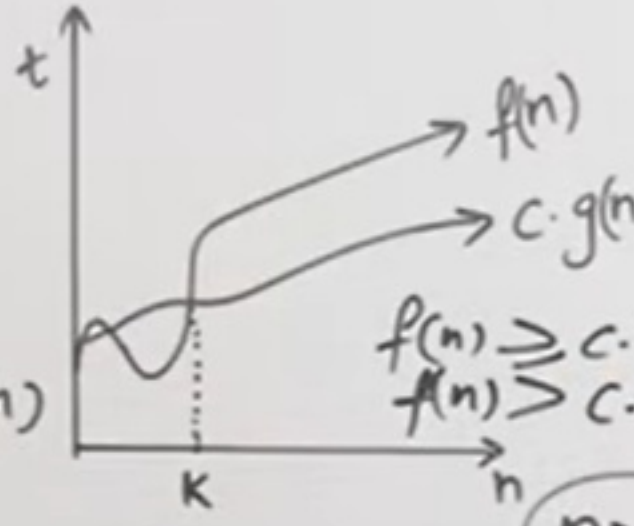
$k \geq 0$

→ Worst Case

→ Upper Bound (At most)

Least

2) Big-Omega (Ω)



$f(n) \geq c$

$f(n) > c$

$n \geq$

→ Best Case

→ greatest Lower Bound (At least

$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

$2n^2 \leq 2n^2 + n \leq 3n^2$

Reflexive | Sy

Big (O) ; $f(n) \leq c \cdot g(n)$, $a \leq b$ ✓

Big Omega (Ω) ; $f(n) \geq c \cdot g(n)$ $a \geq b$ ✓ ×

Theta (Θ) ; $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ ✓

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

Small (o) ; $f(n) < c \cdot g(n)$ $a = b$ ×

Small (ω) ; $f(n) > c \cdot g(n)$ $a < b$ ×

$a > b$ ×

$n^2 \leq n^3$ $a = a$ $a > b$ $b > a$

$n^3 > n^2 > n^3$

$1 < 2$ $2 < 1$ $a = b$ $b = a$

$n^2 = n^2$ $n^2 = n^2$

$a \leq b \leq c$

$a \geq b \geq c$

$a \leq c$

$a \geq b \geq c$

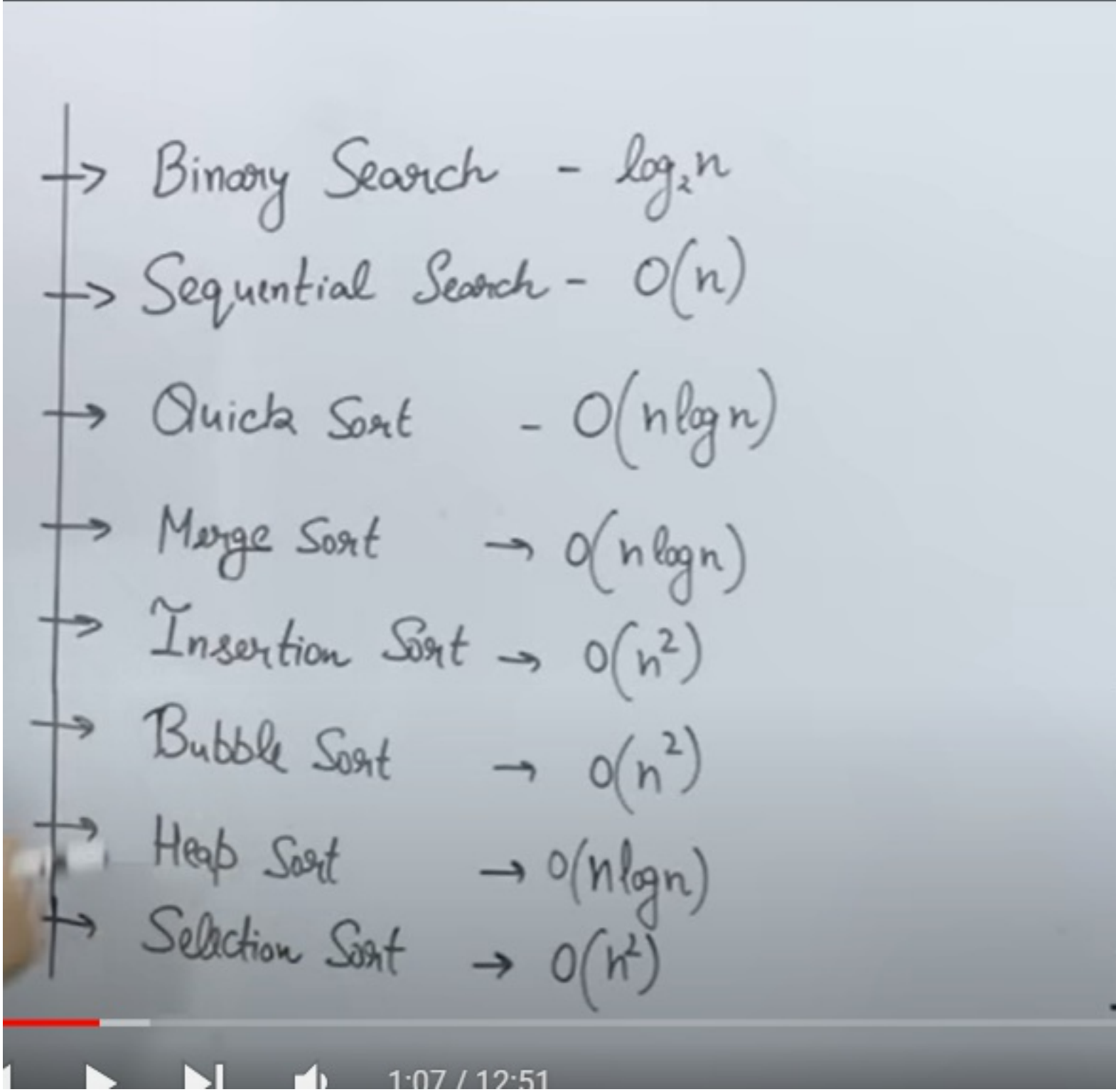
$f(n) \leq O(g(n))$

$f(n) = O(h(n))$

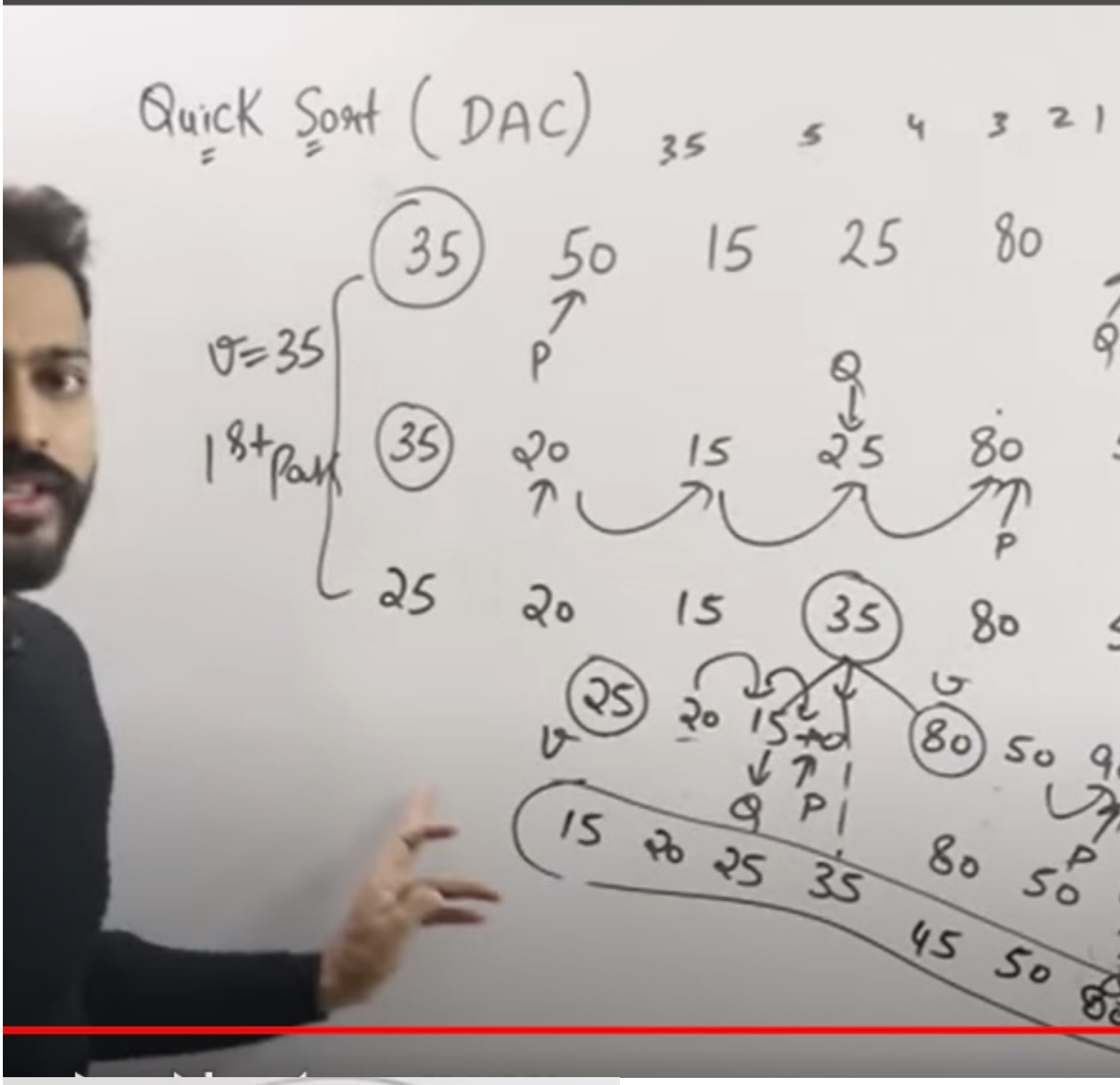
Comparison of various Time Complexities

$O(c) < O(\log \log n) < O(\log n) < O(n^{1/2}) < O(n) < O(n \log$

$O(1)$


Binary Search - $\log_2 n$
Sequential Search - $O(n)$
Quick Sort - $O(n \log n)$
Merge Sort → $O(n \log n)$
Insertion Sort → $O(n^2)$
Bubble Sort → $O(n^2)$
Heap Sort → $O(n \log n)$
Selection Sort → $O(n^2)$
1:07 / 12:51

Sorting



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

20 90 45 $+\infty$

Case	Time Comp
Best Case	O(n*logn)
Average Case	O(n*logn)
Worst Case	$O(n^2)$

Space Complexity
Stable

Quick Sort Pivot Algorithm

Based on our understanding of partitioning in quick sort, write an algorithm for it, which is as follows.

Step 1 – Choose the highest index value has pivot
Step 2 – Take two variables to point left and right of the li
Step 3 – left points to the low index
Step 4 – right points to the high
Step 5 – while value at left is less than pivot move right
Step 6 – while value at right is greater than pivot move le
Step 7 – if both step 5 and step 6 does not match swap le
Step 8 – if left ≥ right, the point where they met is new pi

Quick Sort Pivot Pseudocode

The pseudocode for the above algorithm can be derived as −

```
function partitionFunc(left, right, pivot)
   leftPointer = left
   rightPointer = right - 1

   while True do
      while A[++leftPointer] < pivot do
         //do-nothing
      end while

      while rightPointer > 0 && A[--rightPointer] > pivot do
         //do-nothing
      end while

      if leftPointer >= rightPointer
         break
      else
         swap leftPointer,rightPointer
      end if

   end while

   swap leftPointer,right
   return leftPointer

end function
```

Quick Sort Algorithm

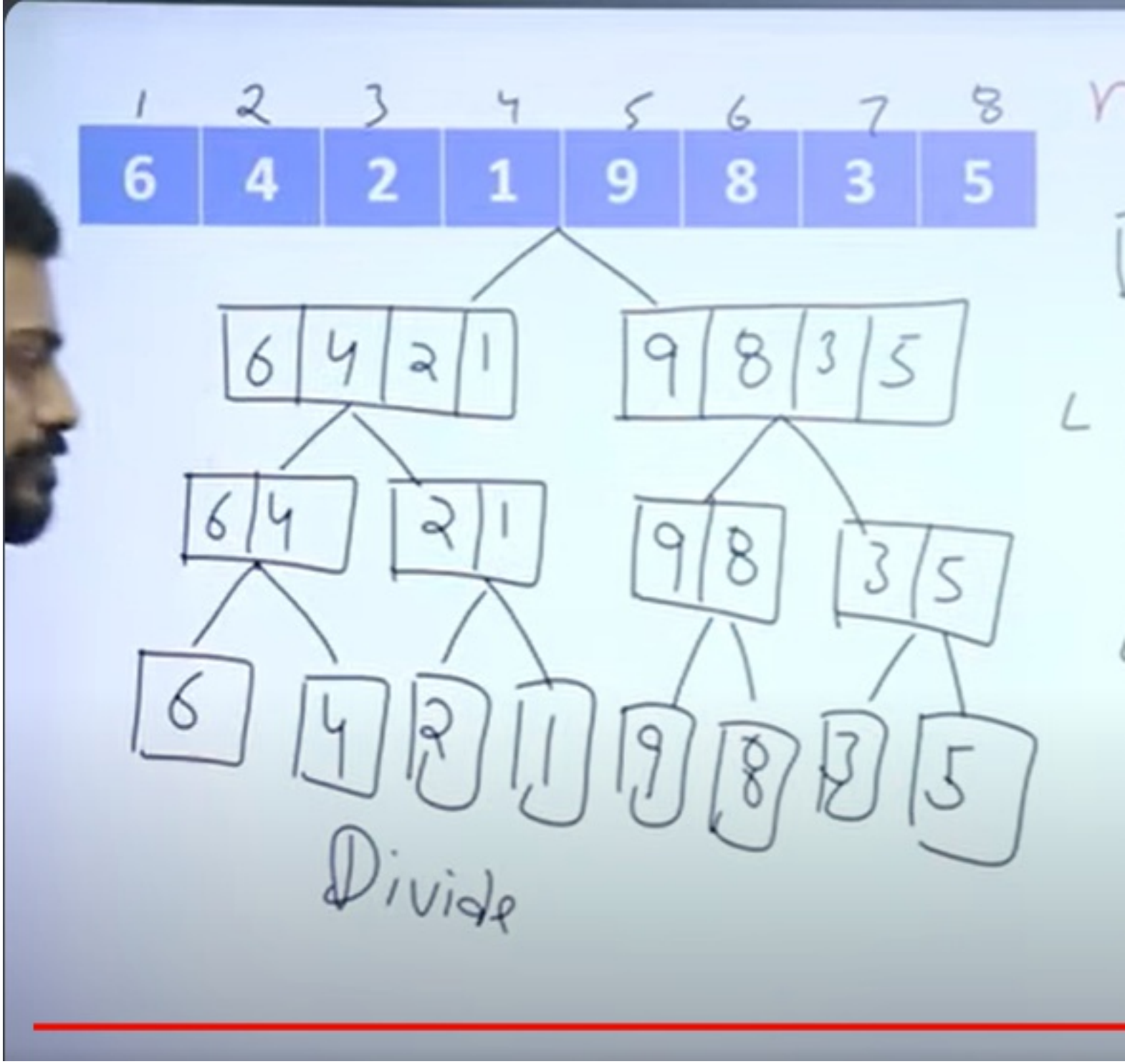
Using pivot algorithm recursively, we end up with smaller possible par
for quick sort. We define recursive algorithm for quicksort as follows -

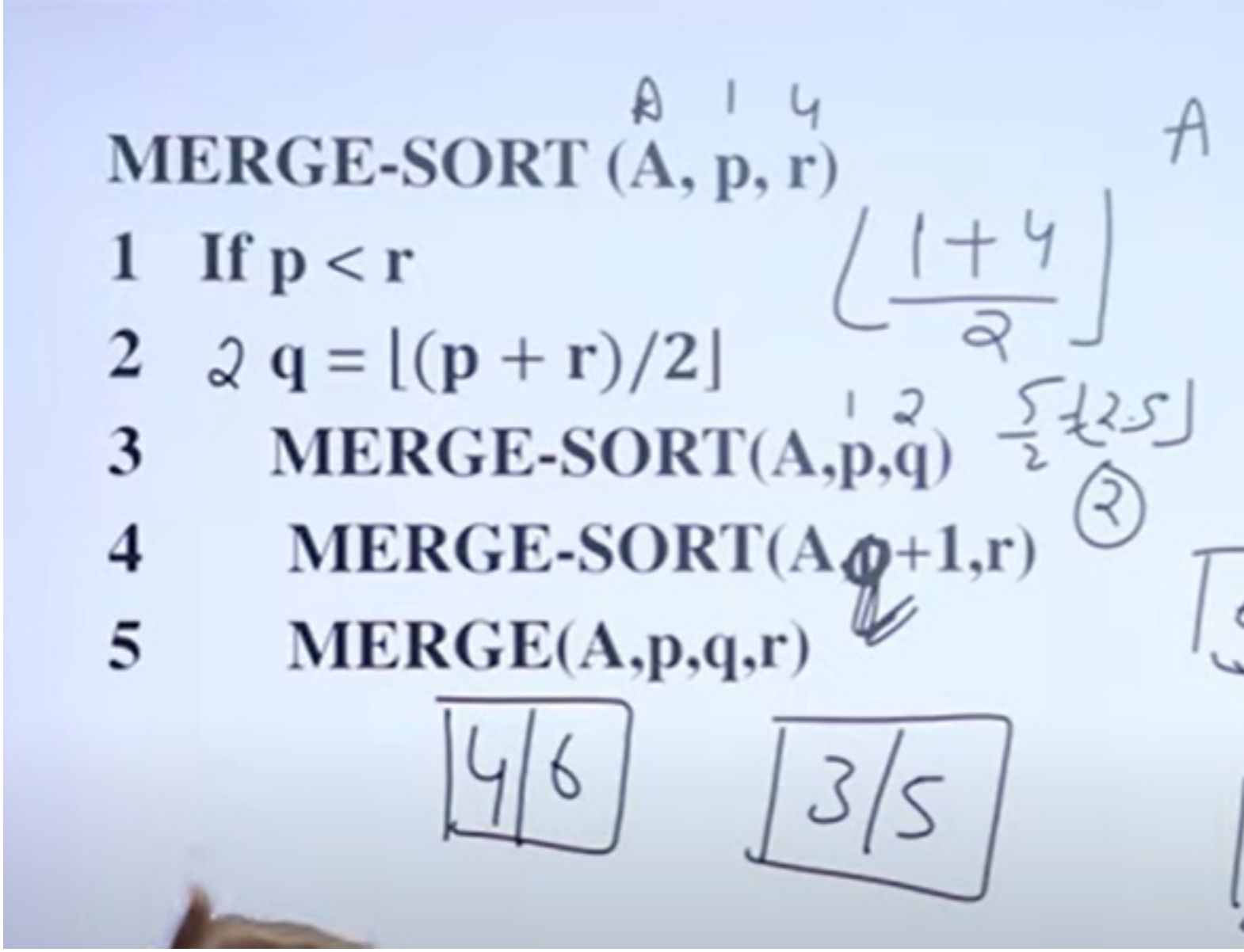
Step 1 – Make the right-most index value pivot
Step 2 – partition the array using pivot value
Step 3 – quicksort left partition recursively
Step 4 – quicksort right partition recursively

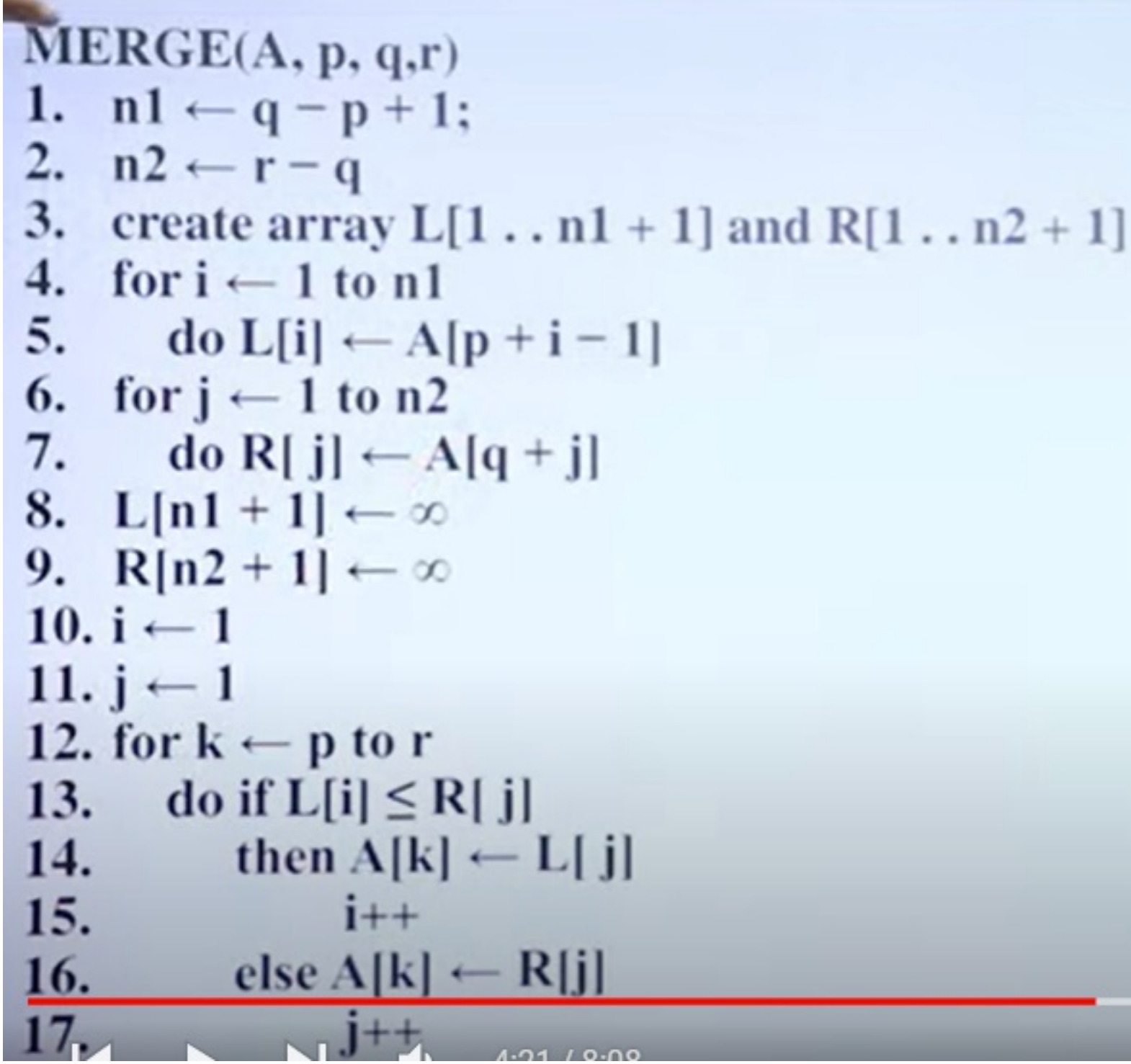
Quick Sort Pseudocode

To get more into it, let see the pseudocode for quick sort algorithm –

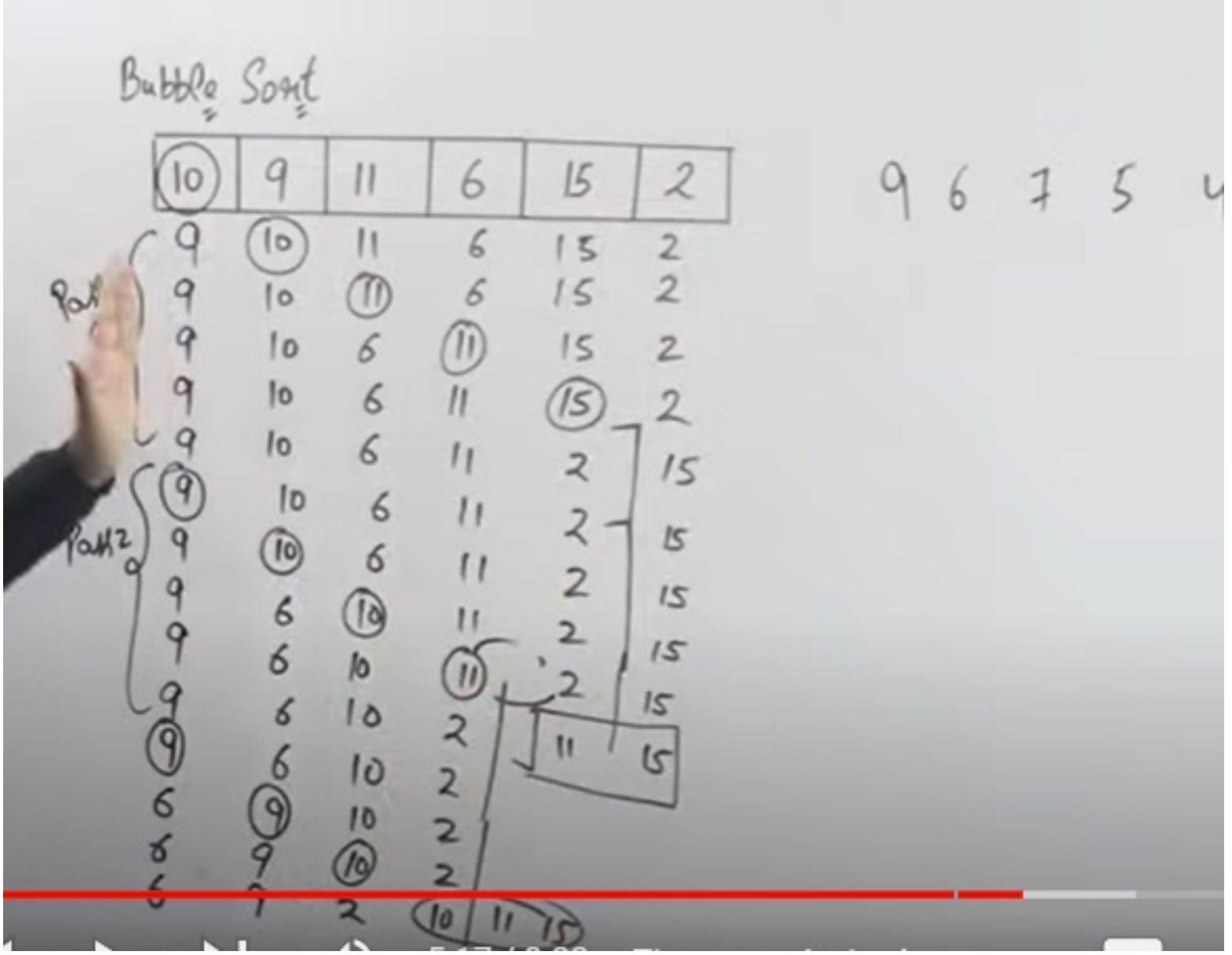
```
procedure quickSort(left, right)

   if right-left <= 0
      return
   else
      pivot = A[right]
      partition = partitionFunc(left, right, pivot)
      quickSort(left,partition-1)
      quickSort(partition+1,right)
   end if

end procedure
```

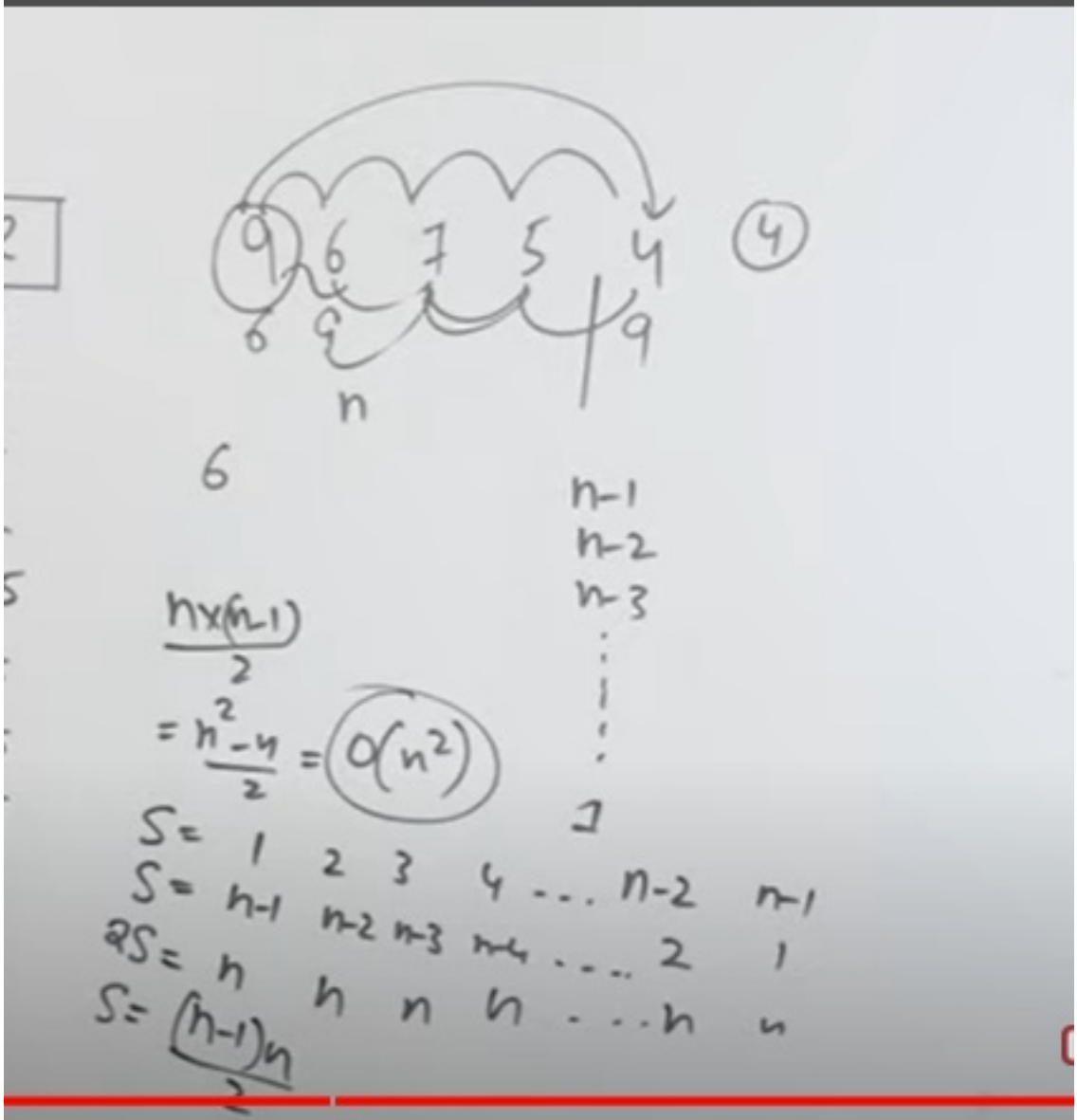

1 2 3 4 5 6 7 8
6 4 2 1 9 8 3 5
6 4 2 1
9 8 3 5
6 4
2 1
9 8
3 5
6
4
2
1
9
8
3
5
Divide


MERGE-SORT (A, p, r)
1 If p < r
2 q = ⌊(p + r)/2⌋
3 MERGE-SORT(A,p,q)
4 MERGE-SORT(A,q+1,r)
5 MERGE(A,p,q,r)


MERGE(A, p, q,r)
1. n1 ← q − p + 1;
2. n2 ← r − q
3. create array L[1 . . n1 + 1] and R[1 . . n2 + 1]
4. for i ← 1 to n1
5. do L[i] ← A[p + i − 1]
6. for j ← 1 to n2
7. do R[j] ← A[q + j]
8. L[n1 + 1] ← ∞
9. R[n2 + 1] ← ∞
10. i ← 1
11. j ← 1
12. for k ← p to r
13. do if L[i] ≤ R[j]
14. then A[k] ← L[j]
15. i++
16. else A[k] ← R[j]
17. j++

Bubble sort


Bubble Sort
10 9 11 6 15 2
9 6 7 5
Pass1
Pass2



In Bubble Sort algorithm,

- *traverse from left and compare adjacent elements a one is placed at right side.*
- *In this way, the largest element is moved to the righ first.*
- *This process is then continued to find the second la place it and so on until the data is sorted.*

- **Total no. of passes:** n-1
- **Total no. of comparisons:** n*(n-1)/2

Complexity Analysis of Bubble Sort:

Time Complexity: $O(N^2)$

Auxiliary Space: O(1)

Case	Time Complexity
Best Case	O(n)
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

Algorithm for optimized bubble sort

```
bubbleSort(array)
n = length(array)
repeat
  swapped = false
  for i = 1 to n - 1
        if array[i - 1] > array[i], then
        swap(array[i - 1], array[i])
        swapped = true
        end if
   end for
   n = n - 1
 until not swapped
end bubbleSort
```

Insertion sort

INSERTION-SORT(A)

```
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1..j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

10 20 30

20 40 60 60

10 20 40 60

10 20 40 50 60

1

Best Case (Ascending Order)

Compari

10 20 30 40 50 60

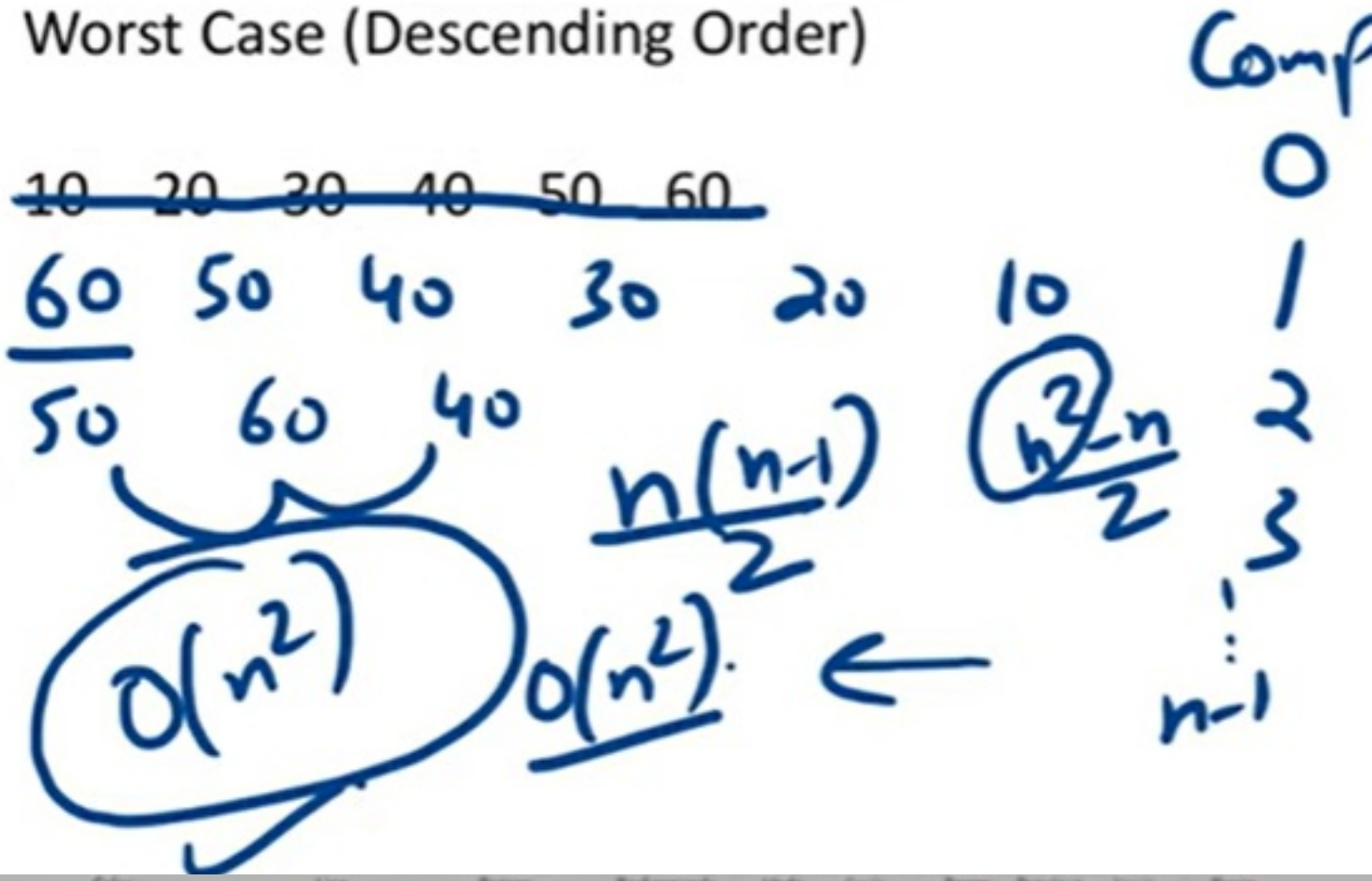
0

O(n)

(n-1)

!
!
!

Worst Case (Descending Order)



Worst-case performance	$O(n^2)$ comparisons and swaps
Best-case performance	$O(n)$ comparisons, $O(1)$ swaps
Average performance	$O(n^2)$ comparisons and swaps

- Insertion Sort is Stable
- Insertion Sort is In place

selection sort

SELECTION_SORT (A)

```
for i ← 1 to n-1 do
    min j ← i;
    min x ← A[i]
    for j ← i + 1 to n do
        If A[j] < min x then
            min j ← j
            min x ← A[j]
    A[min j] ← A [i]
    A[i] ← min x
```

```
for i = 1 to n - 1
/* set current element as
   min = i

   /* check the element to

   for j = i+1 to n
      if list[j] < list[mi
         min = j;
      end if
   end for

   /* swap the minimum ele
   if indexMin != i  then
      swap list[min] and li
   end if
end for
```

40	20	60	10	50	30

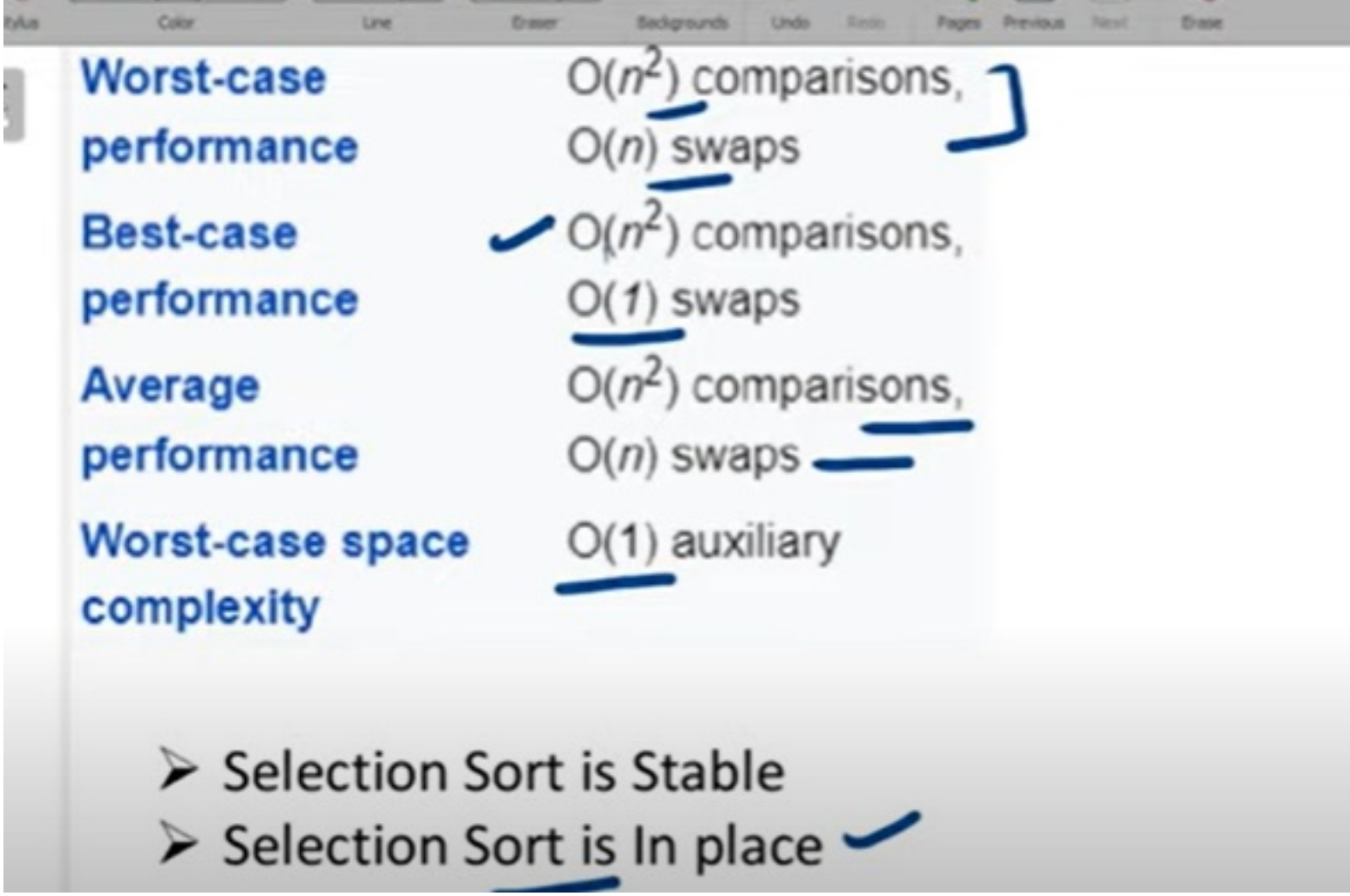
```
for i = 1 to n - 1
/* set current element as minim
```

Best Case (Ascending Order) $O(n^2)$

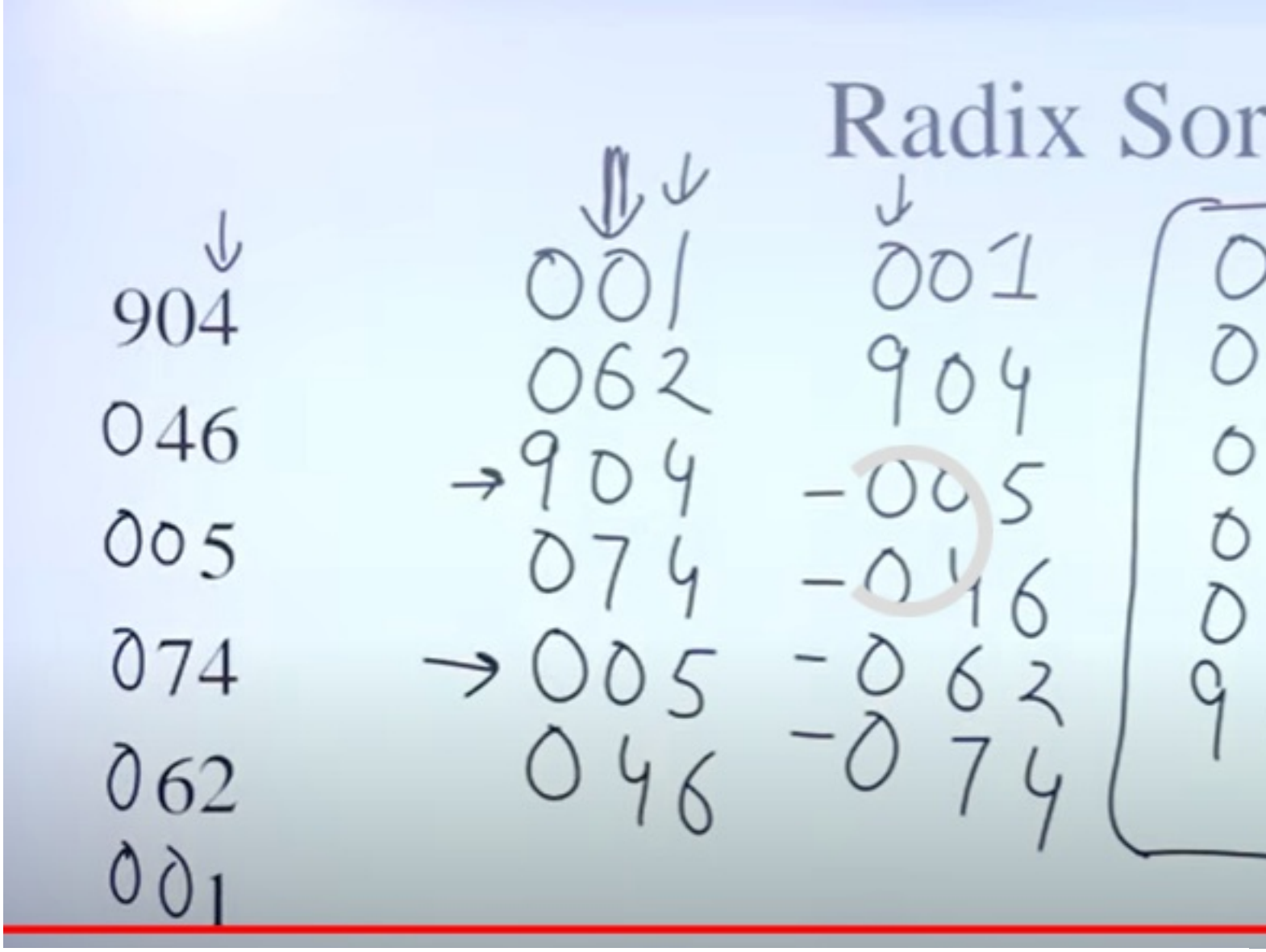
10 20 30 40 50 60

min=10

n-1
n-2
n-3
.
.
.
0


Worst-case performance
O(n²) comparisons, O(n) swaps
Best-case performance
O(n²) comparisons, O(1) swaps
Average performance
O(n²) comparisons, O(n) swaps
Worst-case space complexity
O(1) auxiliary
Selection Sort is Stable
Selection Sort is In place

Radix Sort



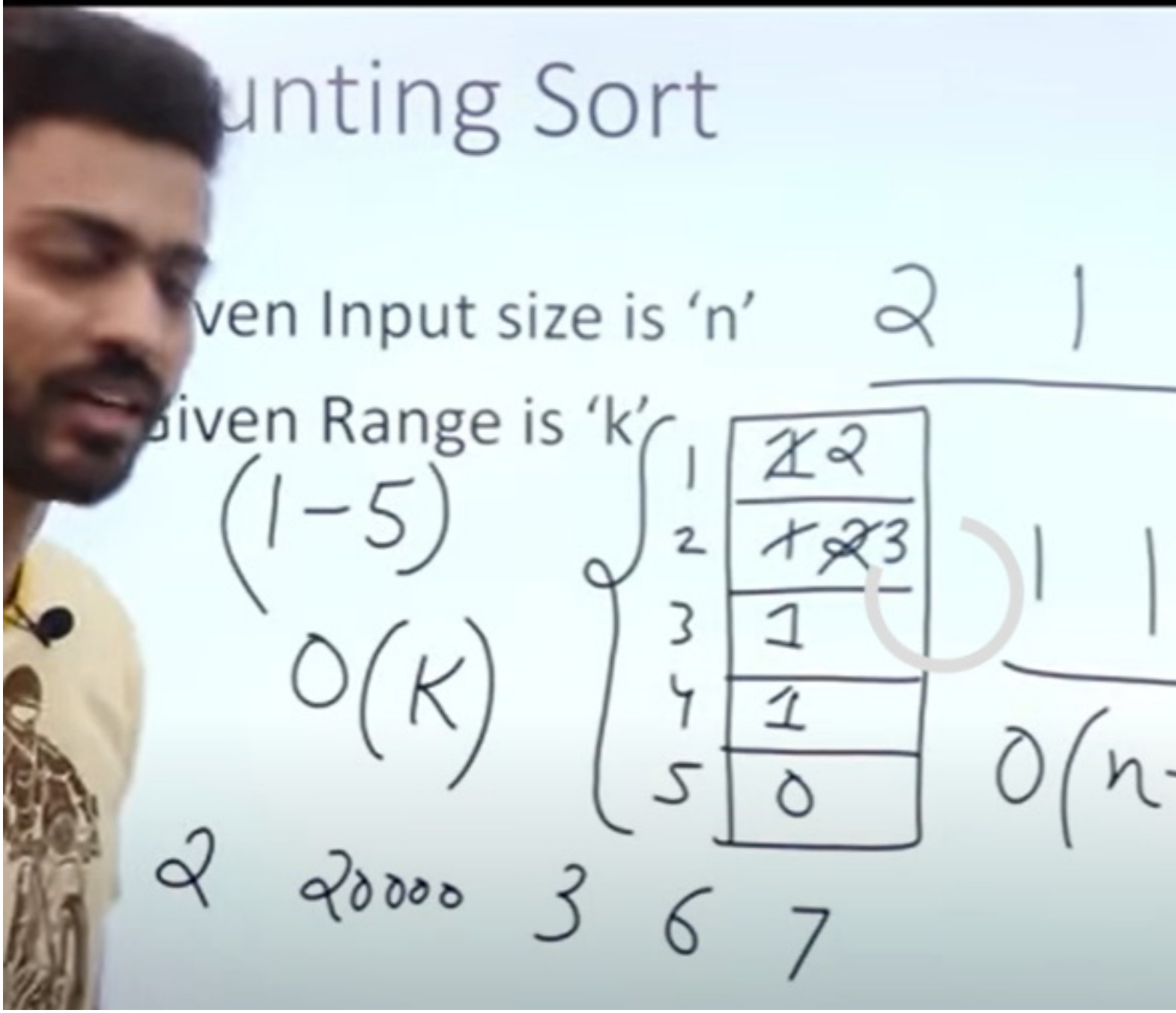
1. Time Complexity

Case	Time Complexity
Best Case	Ω(n+k)
Average Case	θ(nk)
Worst Case	O(nk)

Algorithm

```
radixSort(arr)
max = largest element in the given array
d = number of digits in the largest element (or, max)
Now, create d buckets of size 0 - 9
for i -> 0 to d
sort the array elements using counting sort (or any stable sort) a
the ith place
```

Counting Sort



If range is given then counting sort is best

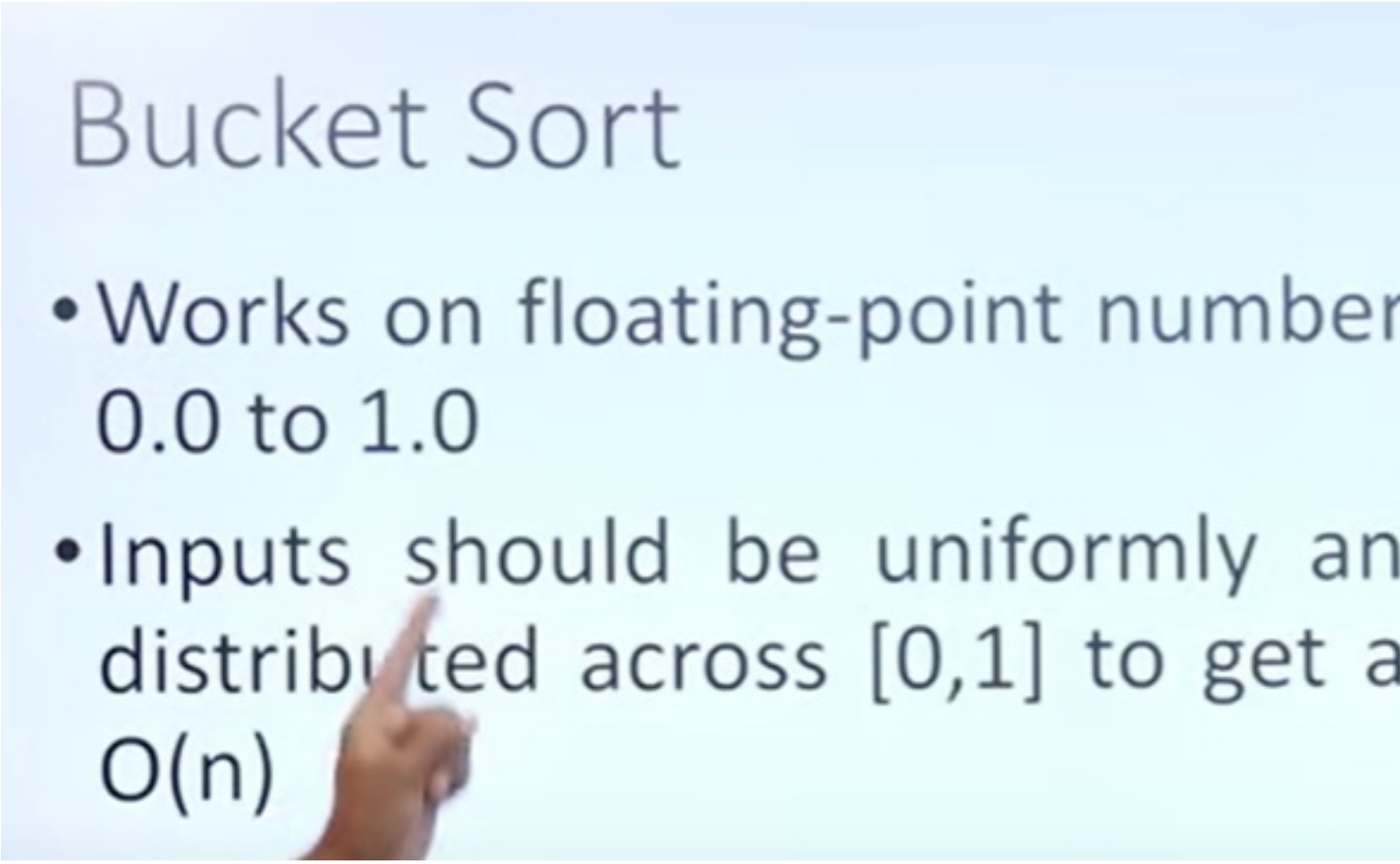
Its not good when range is higher like 2000 in given example

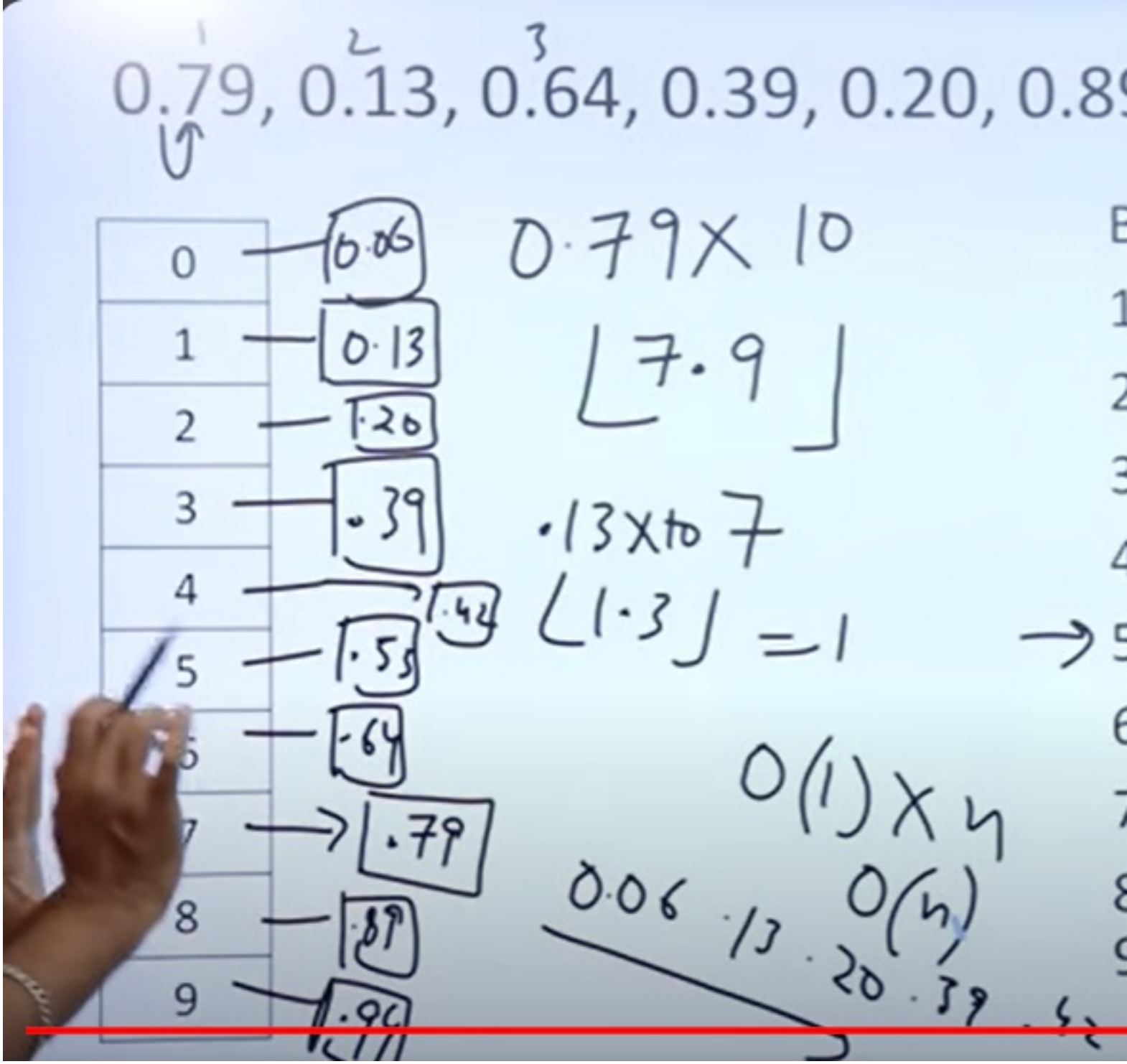
Algorithm

```
countingSort(array, n) // 'n' is the size of array
max = find maximum element in the given array
create count array with size maximum + 1
Initialize count array with all 0's
for i = 0 to n
find the count of every unique element and
store that count at ith position in the count array
for j = 1 to max
Now, find the cumulative sum and store it in count array
for i = n to 1
Restore the array elements
Decrease the count of every restored element by 1
end countingSort
```

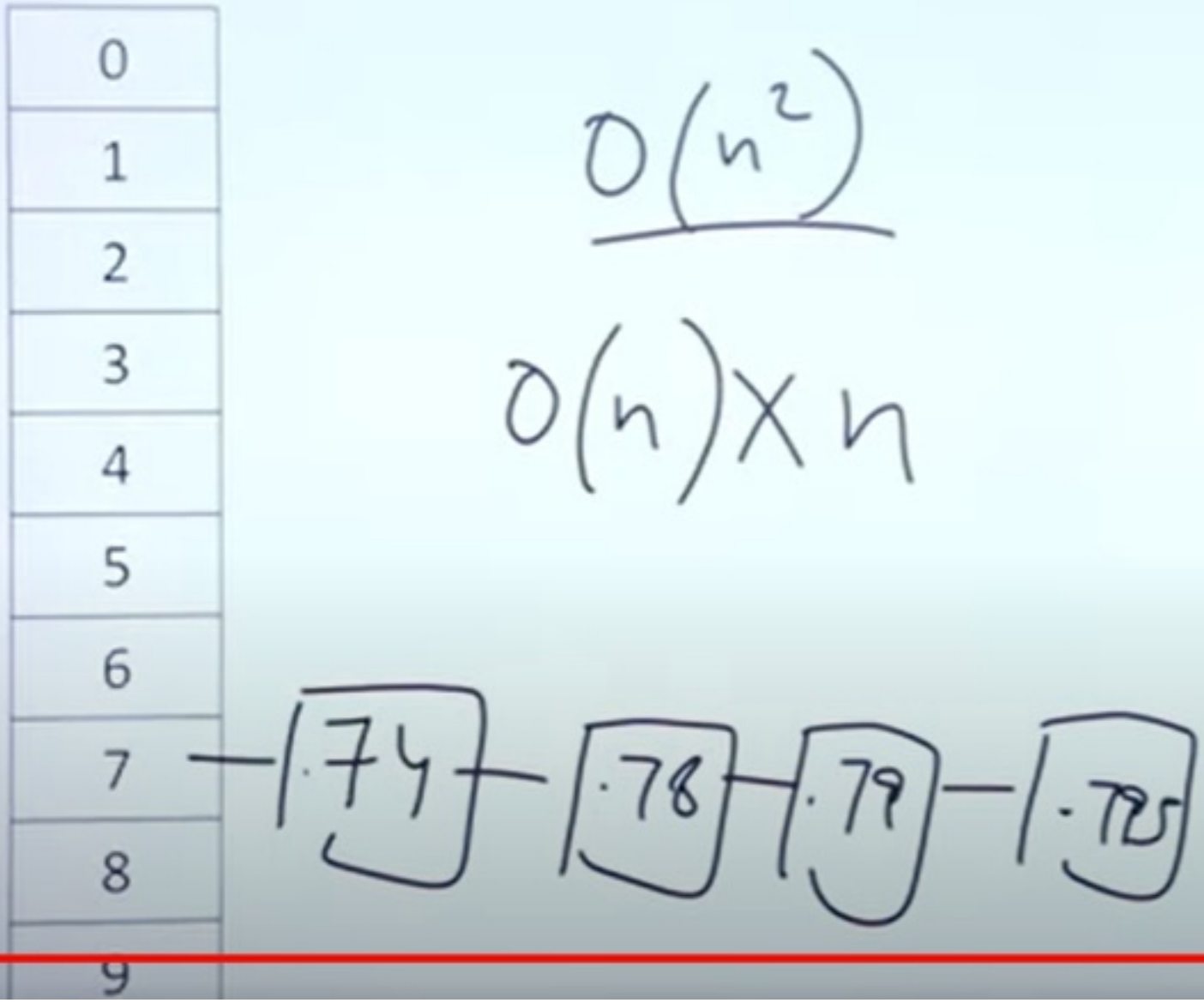
Case	Time
Best Case	O(n + k)
Average Case	O(n + k)
Worst Case	O(n + k)

BUCKET SORT


Bucket Sort
• Works on floating-point number
0.0 to 1.0
• Inputs should be uniformly an
distributed across [0,1] to get a
O(n)


0.79, 0.13, 0.64, 0.39, 0.20, 0.8
0
1
2
3
4
5
6
7
8
9
0.06
0.13
.20
.39
.43
.55
.64
.79
.89
.90
0.79 X 10
⌊7.9⌋
.13 X 10 7
⌊1.3⌋ = 1
O(1) X 4
O(4)
0.06 .13 .20 .39

If the elements are like this: 0.79, 0.78



$O(n^2)$

$O(n) \times n$

The advantages of bucket sort are -

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

The limitations of bucket sort are -

- It may or may not be a stable sorting algorithm.
- It is not useful if we have a large array because it increase
- It is not an in-place sorting algorithm, because some extra

Case	Time
Best Case	O(n + k)
Average Case	O(n + k)
Worst Case	$O(n^2)$

Space Complexity
Stable