Sorting Algorithms Explained

I have my analysis of algorithm exam in a few hours and I need you to teach me sorting algorithms.
I want you to teach them in a teacher like way, such that it gets deep into my mind.
Try to teach in simple yet effective way, also at any point if you feel that the concept is tricky, try to simplify it for me.

Take as many lines of explanation as needed to explain it effectively(no bar).

I will give you the name of the sorting algorithm.
I want you to follow a certain format while giving its explanation for each.

First you explanation with example(in depth step by step)
Then Algorithm
Then Pseudocode, give in normal text only (no code type output because it becomes difficult to view on phones)
Then Analysis (Best case, worst case, average case with examples)

Common sorting algorithms

1) Insertion sort
2) Selection sort

Insertion Sort

Explanation with Example (Step by Step)

Imagine you are sorting a deck of cards in your hand. You pick up one card at a time and place it in its correct position among the ones already sorted.

Let's take an example and sort the array:
[7, 3, 5, 2, 8]

1. **Start with the second element (index 1, value = 3)**
    - Compare 3 with 7. Since 3 is smaller, shift 7 to the right.
    - Insert 3 in the first position.
    - Array becomes: **[3, 7, 5, 2, 8]**
2. **Take the third element (index 2, value = 5)**
    - Compare 5 with 7. Since 5 is smaller, shift 7 to the right.
    - Compare 5 with 3. Since 5 is greater, place it after 3.
    - Array becomes: **[3, 5, 7, 2, 8]**
3. **Take the fourth element (index 3, value = 2)**

- Compare 2 with 7 → shift 7 to the right.
    - Compare 2 with 5 → shift 5 to the right.
    - Compare 2 with 3 → shift 3 to the right.
    - Place 2 in the first position.
    - Array becomes: **[2, 3, 5, 7, 8]**
4. **Take the fifth element (index 4, value = 8)**
    - Compare 8 with 7. Since 8 is greater, it remains in place.
    - Array remains: **[2, 3, 5, 7, 8]**

Final sorted array: [2, 3, 5, 7, 8]

Algorithm

1. Start from the second element.
2. Compare it with previous elements and shift larger elements to the right.
3. Insert the current element in its correct position.
4. Repeat the process for all elements in the array.

Pseudocode

```vbnet
InsertionSort(arr, n):
    For i from 1 to n-1:
        key = arr[i]
        j = i - 1
        While j >= 0 and arr[j] > key:
            arr[j+1] = arr[j]
            j = j - 1
        arr[j+1] = key
```

Analysis

- **Best Case (Already Sorted Array, e.g., [1,2,3,4,5]) → O(n)**
- **Worst Case (Reverse Sorted Array, e.g., [5,4,3,2,1]) → O(n^2)**
- **Average Case (Random Order, e.g., [3,1,4,2,5]) → O(n^2)**

Why?

- Best case happens when no shifting is required. Only comparisons take O(n).
- Worst case requires shifting each element for every new insertion, leading to O(n^2).

Selection Sort

Explanation with Example (Step by Step)

Selection Sort works like this:

- Find the smallest element and swap it with the first element.
- Find the second smallest element and swap it with the second element.

- Repeat until the whole array is sorted.

Let's take the array:
[7, 3, 5, 2, 8]

1. **First pass:**
   - Find the smallest number (2). Swap it with the first element.
   - Array becomes: **[2, 3, 5, 7, 8]**
2. **Second pass:**
   - The remaining array is [3, 5, 7, 8]. The smallest is 3, which is already in place.
   - Array remains: **[2, 3, 5, 7, 8]**
3. **Third pass:**
   - The remaining array is [5, 7, 8]. The smallest is 5, which is already in place.
   - Array remains: **[2, 3, 5, 7, 8]**
4. **Fourth pass:**
   - The remaining array is [7, 8]. The smallest is 7, which is already in place.
   - Array remains: **[2, 3, 5, 7, 8]**
5. **Final sorted array: [2, 3, 5, 7, 8]**

Algorithm

1. Find the minimum element in the unsorted part of the array.
2. Swap it with the first element of the unsorted part.
3. Move the boundary between sorted and unsorted parts one step right.
4. Repeat until the array is sorted.

Pseudocode

```vbnet
SelectionSort(arr, n):
    For i from 0 to n-1:
        min_index = i
        For j from i+1 to n:
            If arr[j] < arr[min_index]:
                min_index = j
        Swap arr[i] and arr[min_index]
```

Analysis

- **Best Case (Already Sorted Array, e.g., [1,2,3,4,5]) → O(n^2)**
- **Worst Case (Reverse Sorted Array, e.g., [5,4,3,2,1]) → O(n^2)**
- **Average Case (Random Order, e.g., [3,1,4,2,5]) → O(n^2)**

Why?

- Selection sort **always** runs in O(n^2) because it performs O(n^2) comparisons regardless of order.
- Even if the array is already sorted, it still scans for the minimum element in every iteration.

Key Differences Between Insertion Sort & Selection Sort

Feature	Insertion Sort	Selection Sort
Best case time	O(n)	$O(n^2)$
Worst case time	$O(n^2)$	$O(n^2)$
Average case time	$O(n^2)$	$O(n^2)$
Stability	Stable	Not stable
Space complexity	O(1)	O(1)
Efficiency	Efficient for small lists	Not efficient

Final Thoughts

- **Use Insertion Sort** if the array is nearly sorted (because of its O(n) best case).
- **Use Selection Sort** if swaps are expensive (because it makes at most n swaps).

Would you like me to continue with more sorting algorithms?

Sorting in Linear Time
1.Counting sort

Divide and conquer algorithms
1. Quick sort
2. Randomized Quick sort

Please try to keep it simple to understand as before

Sorting in Linear Time

Counting Sort

Explanation with Example (Step by Step)

Counting Sort is used when we know the range of values in the array. Instead of comparing elements, we count how many times each value appears and use that count to determine their position in the sorted array.

Let's take an example:
Input Array: [4, 2, 2, 8, 3, 3, 1]

Step 1: Find the Range

- The minimum value is **1**, and the maximum value is **8**.
- Create a `count` array of size `max + 1` (i.e., size 9, indexed 0-8).

Step 2: Count Occurrences

- Count how many times each number appears.
- `count` array before cumulative sum:

```makefile
Index:  0  1  2  3  4  5  6  7  8
Count:  0  1  2  2  1  0  0  0  1
```

Step 3: Cumulative Sum

- Modify the `count` array to store the position of each element in the sorted array.

```makefile
Index:  0  1  2  3  4  5  6  7  8
Count:  0  1  3  5  6  6  6  6  7
```

Step 4: Build the Sorted Array

- Traverse the input array from right to left and place each element at its correct position.
- **Final sorted array: [1, 2, 2, 3, 3, 4, 8]**

Algorithm

1. Find the maximum value in the array.
2. Create a `count` array of size (max + 1) and initialize it to 0.
3. Count occurrences of each element.
4. Compute cumulative sums in the `count` array.
5. Place elements in the correct position in the sorted output array.

Pseudocode

```sql
CountingSort(arr, n, max):
    Create count array of size (max + 1), initialized to 0
    For each element in arr:
        Increment count[element]

    Convert count array to cumulative sum

    Create output array of size n
    For each element in arr (traverse from right to left):
        Place it at count[element] - 1 position in output
        Decrement count[element]

    Copy output array back to arr
```

Analysis

- **Best Case:** O(n + k)
- **Worst Case:** O(n + k)
- **Average Case:** O(n + k)

    - Where `n` is the number of elements, and `k` is the range of values.
- **Space Complexity:** O(n + k) (extra space required for `count` array).

When to Use?

- Works best when `k` (range of numbers) is not too large compared to `n`.
- Not suitable for sorting large numbers with a huge range (e.g., millions).

Divide and Conquer Algorithms

Quick Sort

Explanation with Example (Step by Step)

Quick Sort works by selecting a **pivot** element, then rearranging the array so that:

- Elements smaller than the pivot go to the **left**.
- Elements larger than the pivot go to the **right**.

Example:
Input Array: **[8, 3, 7, 6, 2, 5]**
Let's take **pivot = 5** (last element).

Step 1: Partitioning

Rearrange elements:

- **Smaller numbers (≤5) move to the left:** [3, 2]
- **Pivot (5) stays in the middle**
- **Larger numbers (>5) move to the right:** [8, 7, 6]

New arrangement: **[3, 2, 5, 8, 7, 6]**

Step 2: Recursively Sort Left and Right Parts

Left Part [3, 2]:

- Pivot = 2
- Swap to get: **[2, 3]**

Right Part [8, 7, 6]:

- Pivot = 6
- Swap to get: **[6, 7, 8]**

Final sorted array: **[2, 3, 5, 6, 7, 8]**

Algorithm

1. Choose a pivot element.
2. Partition the array into two halves (smaller and larger than pivot).

3. Recursively sort both halves.
4. Combine sorted halves.

Pseudocode

```scss
QuickSort(arr, low, high):
    If low < high:
        pivot_index = Partition(arr, low, high)
        QuickSort(arr, low, pivot_index - 1)
        QuickSort(arr, pivot_index + 1, high)

Partition(arr, low, high):
    pivot = arr[high]
    i = low - 1
    For j from low to high - 1:
        If arr[j] < pivot:
            i = i + 1
            Swap arr[i] and arr[j]
    Swap arr[i+1] and arr[high]
    Return i + 1
```

Analysis

- **Best Case (Balanced Partitioning):** O(n log n)
- **Worst Case (Already Sorted or Reversed List):** O(n^2)
- **Average Case:** O(n log n)
- **Space Complexity:** O(log n) (recursive calls)

When to Use?

- Best for large datasets due to O(n log n) average performance.
- Avoid worst-case by using **randomized quick sort**.

Randomized Quick Sort

Explanation with Example (Step by Step)

Randomized Quick Sort improves Quick Sort by selecting a **random pivot** instead of the last element. This reduces the chance of worst-case performance.

Example:
Input Array: **[8, 3, 7, 6, 2, 5]**
Instead of choosing the last element as the pivot, pick a **random** one (e.g., `` `3` ``).

Partition and sort similarly:
Pivot = 3 → Left: [2], Right: [8, 7, 6, 5]
Recurse on both sides → Final sorted array: **[2, 3, 5, 6, 7, 8]**

Algorithm

1. Choose a **random pivot** instead of the last element.

2. Swap it with the last element and proceed as normal Quick Sort.

Pseudocode

```scss
RandomizedQuickSort(arr, low, high):
    If low < high:
        pivot_index = RandomPartition(arr, low, high)
        RandomizedQuickSort(arr, low, pivot_index - 1)
        RandomizedQuickSort(arr, pivot_index + 1, high)

RandomPartition(arr, low, high):
    random_index = Random(low, high)
    Swap arr[random_index] with arr[high]
    Return Partition(arr, low, high)
```

Analysis

- **Best Case:** O(n log n)
- **Worst Case:** O(n log n) *(with high probability due to randomness)*
- **Average Case:** O(n log n)
- **Space Complexity:** O(log n)

When to Use?

- Preferable when dealing with **adversarial inputs** (e.g., already sorted/reversed lists).
- Eliminates Quick Sort's worst-case issue.

Final Thoughts

- **Counting Sort** is best for sorting integers in a limited range **O(n + k)**.
- **Quick Sort** is best for large datasets but can suffer from O(n^2) worst case.
- **Randomized Quick Sort** prevents worst-case and is the preferred variant.

Would you like any more clarifications? 😊

3. Merge sort
4. Shell sort
5. Heapsort

Divide and Conquer Algorithms

Merge Sort

Explanation with Example (Step by Step)

Merge Sort is a **divide and conquer** algorithm that splits an array into halves, sorts them separately, and then merges them back together in sorted order.

Example:
Input Array: [6, 3, 8, 5, 2, 7, 4, 1]

Step 1: Divide the Array

Break it into two halves repeatedly until each subarray has one element.

```css
[6, 3, 8, 5]      [2, 7, 4, 1]
[6, 3]  [8, 5]    [2, 7]  [4, 1]
[6]  [3]  [8]  [5]  [2]  [7]  [4]  [1]
```

Step 2: Merge in Sorted Order

- Merge `` `[6]` `` and `` `[3]` `` → `` `[3, 6]` ``
- Merge `` `[8]` `` and `` `[5]` `` → `` `[5, 8]` ``
- Merge `` `[3, 6]` `` and `` `[5, 8]` `` → `` `[3, 5, 6, 8]` ``
- Merge `` `[2]` `` and `` `[7]` `` → `` `[2, 7]` ``
- Merge `` `[4]` `` and `` `[1]` `` → `` `[1, 4]` ``
- Merge `` `[2, 7]` `` and `` `[1, 4]` `` → `` `[1, 2, 4, 7]` ``
- Merge `` `[3, 5, 6, 8]` `` and `` `[1, 2, 4, 7]` `` → **Final sorted array:** `` `[1, 2, 3, 4, 5, 6, 7, 8]` ``

Algorithm

1. If the array has 1 element, return.
2. Find the middle index and divide the array into two halves.
3. Recursively sort both halves.
4. Merge the sorted halves into a single sorted array.

Pseudocode

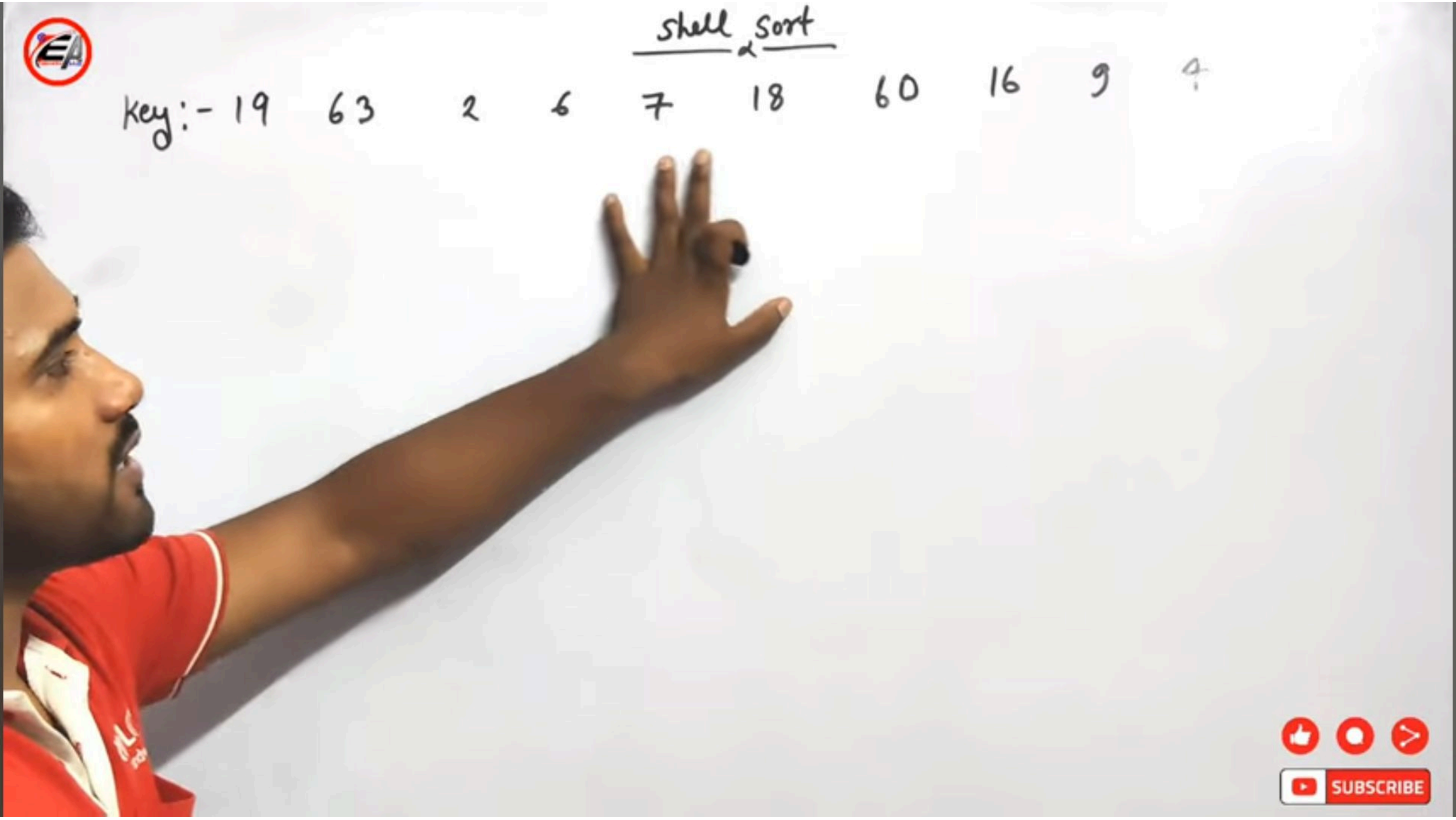
```sql
MergeSort(arr, left, right):
    If left < right:
        mid = (left + right) / 2
        MergeSort(arr, left, mid)
        MergeSort(arr, mid + 1, right)
        Merge(arr, left, mid, right)

Merge(arr, left, mid, right):
    Create two temporary arrays L and R
    Copy left half into L, right half into R
    Merge L and R back into arr in sorted order
```

Analysis

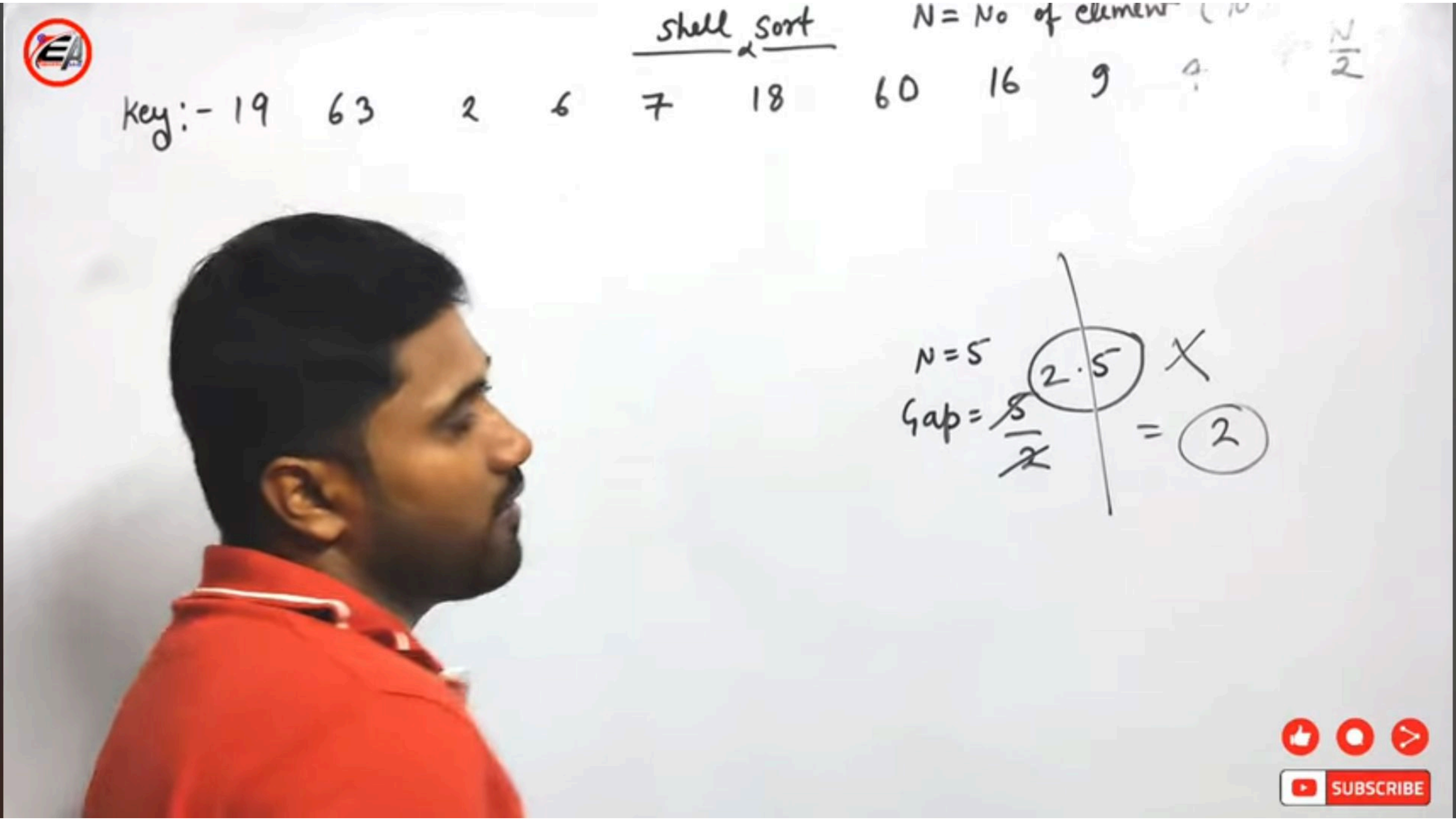
- **Best Case:** O(n log n)

Shell sort



Shell sort

Key:- 19 63 2 6 7 18 60 16 9 4



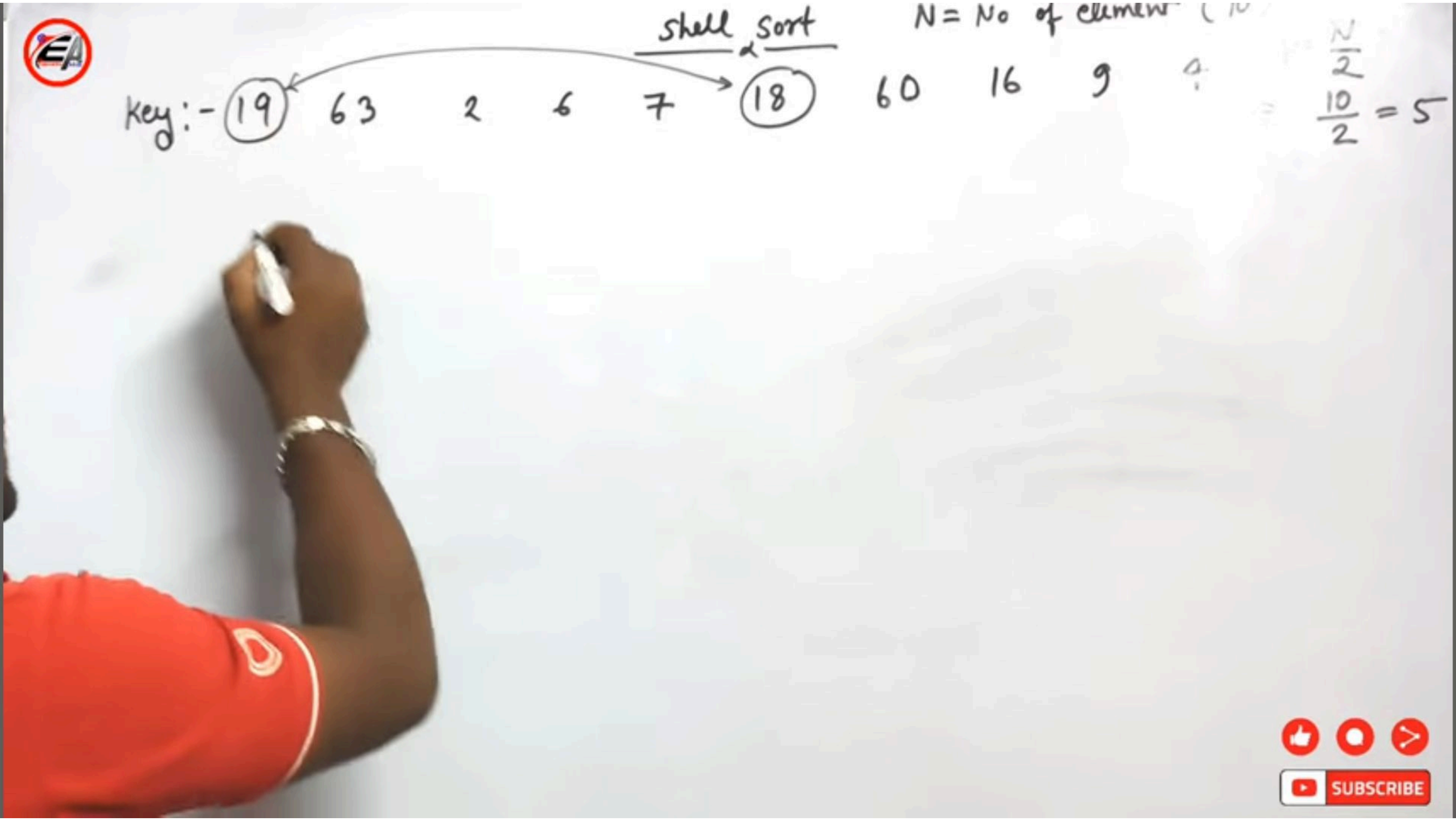
Shell sort   N = No of element (N

$\frac{N}{2}$

Key:- 19   63   2   6   7   18   60   16   9   4

N = 5

$\text{Gap} = \frac{\cancel{5}}{\cancel{2}}$ 2.5   X   = 2

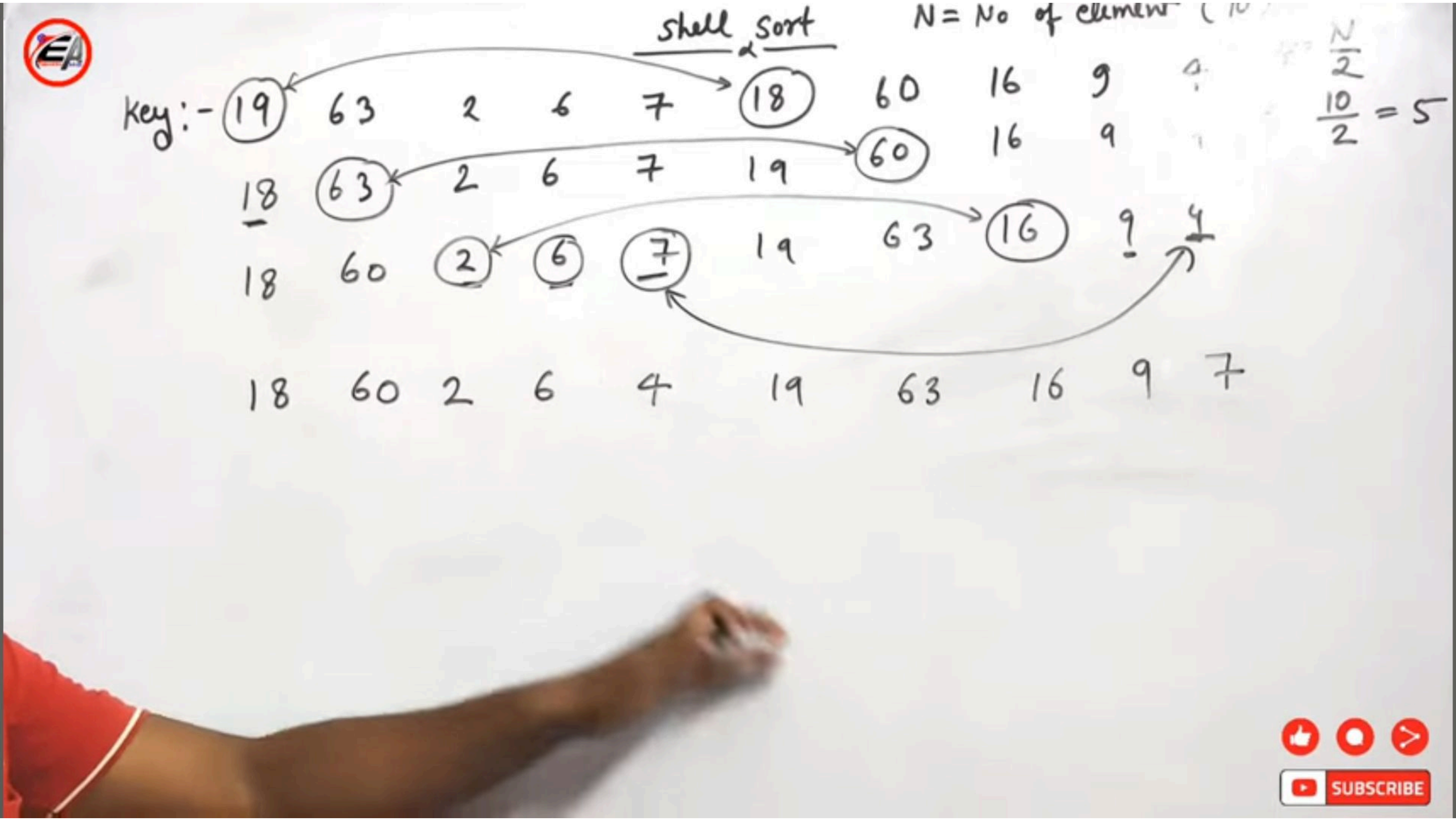


Shell Sort

N = No of elemen[illegible] (10)

$$\frac{N}{2} = \frac{10}{2} = 5$$

$$\begin{tikzcd}[column sep=small] \text{Key:-} & \boxed{19} \arrow[rrrrr, bend left, <->] & 63 & 2 & 6 & 7 & \boxed{18} & 60 & 16 & 9 & 4 \end{tikzcd}$$



Shell sort

N = No of element (10)

$\frac{N}{2}$

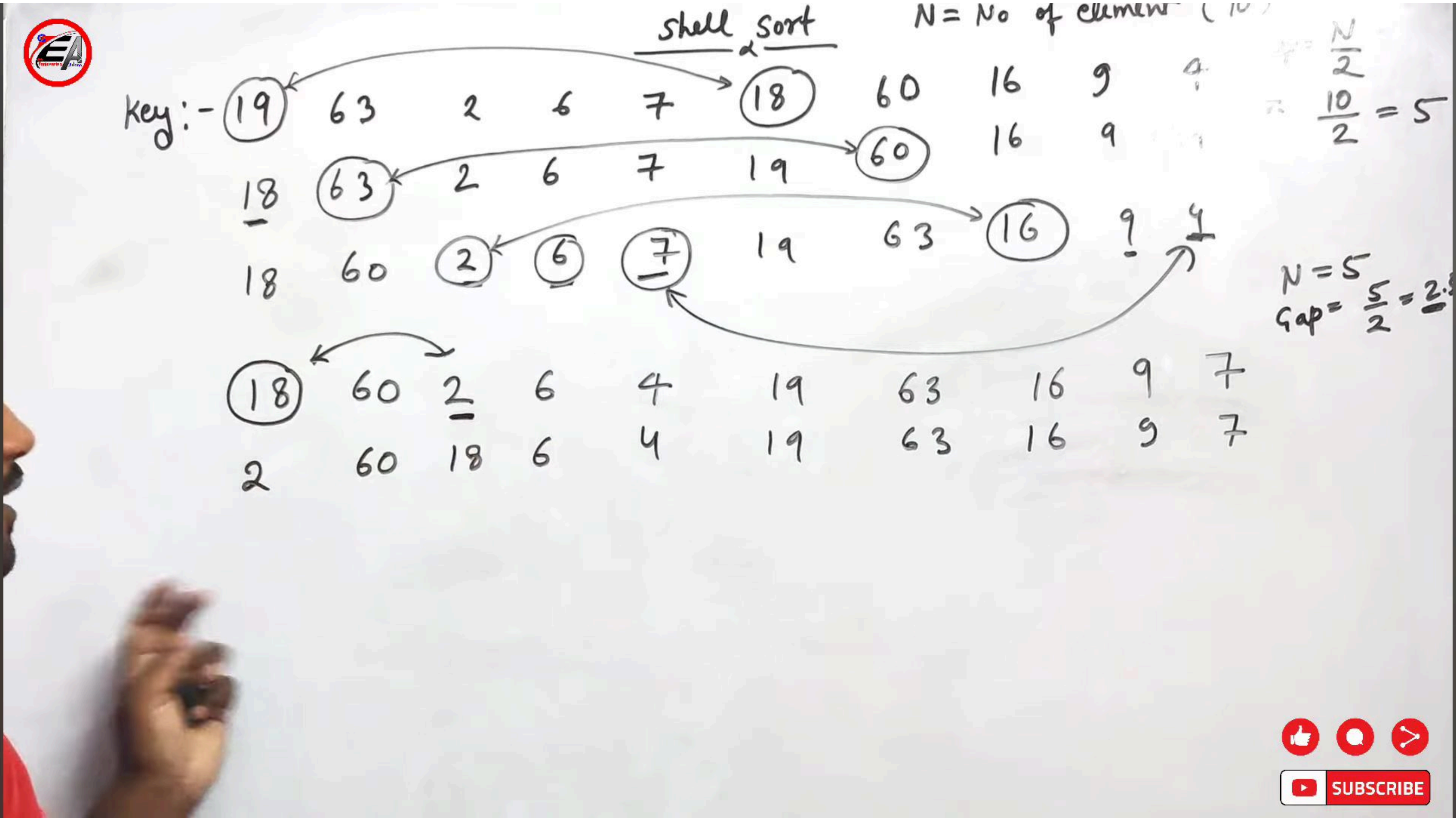
$\frac{10}{2} = 5$

Key:- 19 63 2 6 7 18 60 16 9 4

18 63 2 6 7 19 60 16 9 4

18 60 2 6 7 19 63 16 9 4

18 60 2 6 4 19 63 16 9 7



Shell sort

N = No of element (10)

Gap = $\frac{N}{2}$ = $\frac{10}{2}$ = 5

Key:- 19 63 2 6 7 18 60 16 9 4

18 63 2 6 7 19 60 16 9 4

18 60 2 6 7 19 63 16 9 4

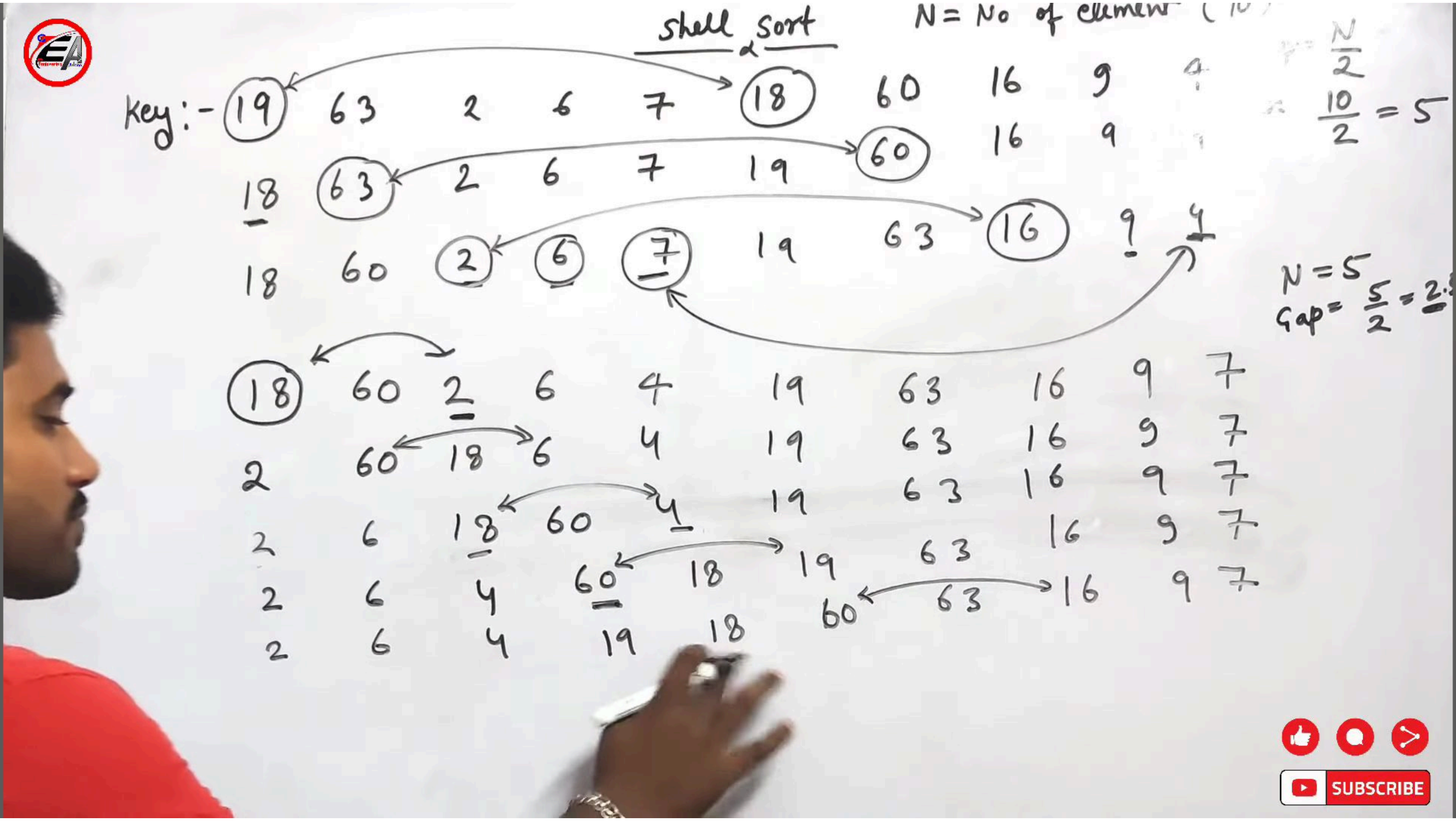
N = 5

Gap = $\frac{5}{2}$ = 2.5

18 60 2 6 4 19 63 16 9 7

2 60 18 6 4 19 63 16 9 7

SUBSCRIBE



Shell sort

N = No of element (10)

$= \frac{N}{2}$

$\frac{10}{2} = 5$

Key:- 19 63 2 6 7 18 60 16 9 4

18 63 2 6 7 19 60 16 9 4

18 60 2 6 7 19 63 16 9 4

N = 5

Gap = $\frac{5}{2}$ = 2.

18 60 2 6 4 19 63 16 9 7

2 60 18 6 4 19 63 16 9 7

2 6 18 60 4 19 63 16 9 7

2 6 4 60 18 19 63 16 9 7

2 6 4 19 18 60 63 16 9 7

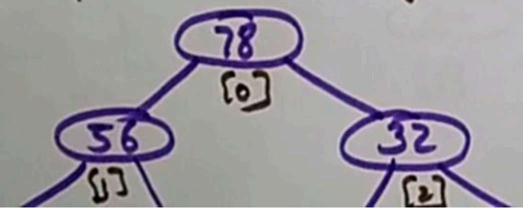
Heap sort

→ **Heap sort**

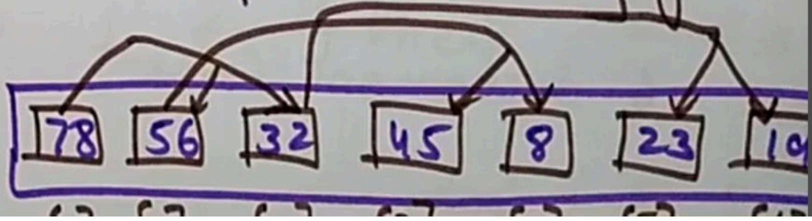
⇒ The heap sort is an improved version of the selection sort in which the larget element (the root) is selected and Exchanged with the last element in the unsorted list.

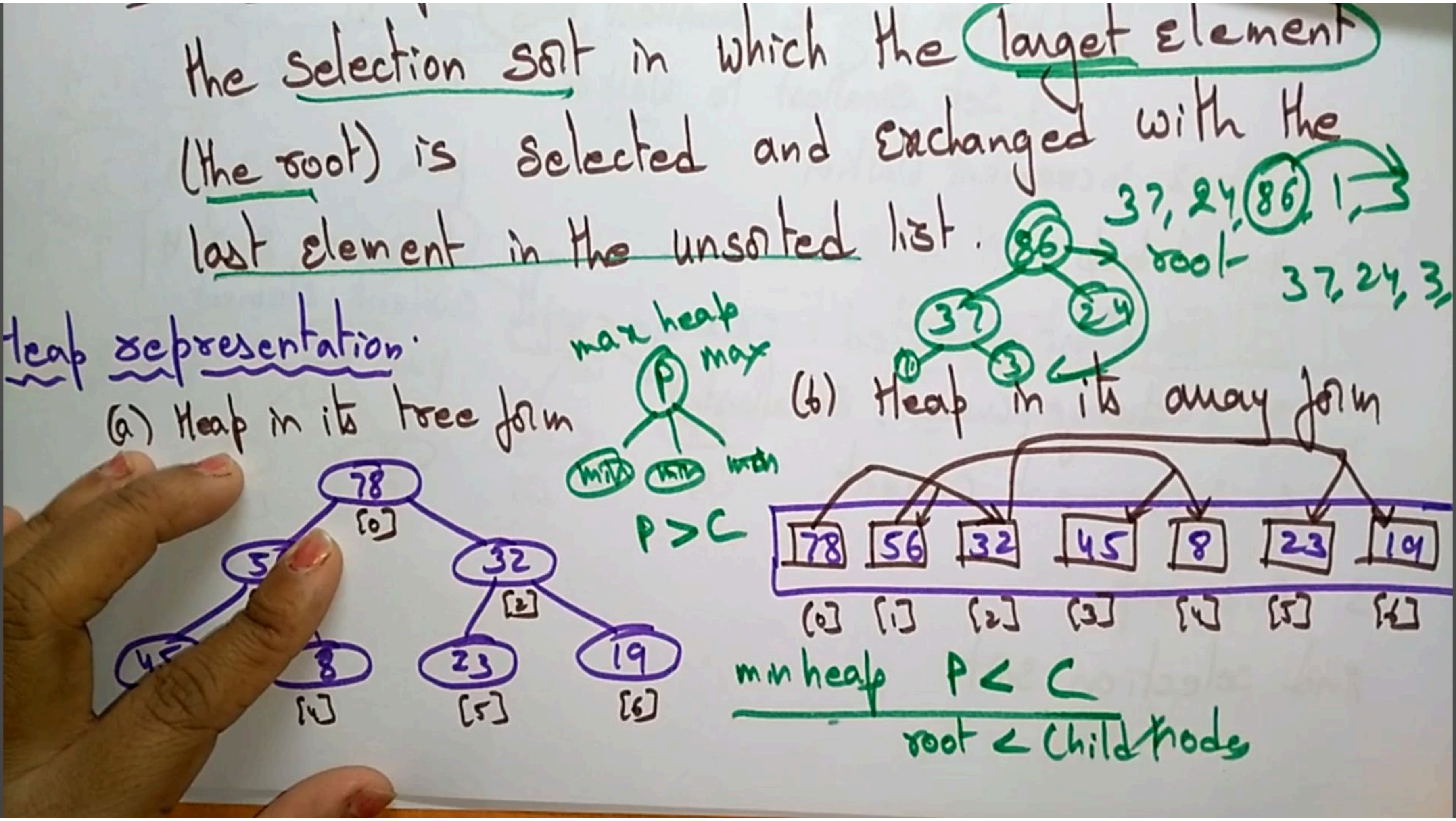
Heap representation.

(a) Heap in its tree form



(b) Heap in its array form





The selection sort in which the larget element (the root) is selected and Exchanged with the last element in the unsorted list.

86
37
24
1
3
root
37, 24, 86, 1, 3
37, 24, 3,

Heap representation.

(a) Heap in its tree form

78
[0]
5
32
[2]
45
8
[4]
23
[5]
19
[6]
max heap
p
max

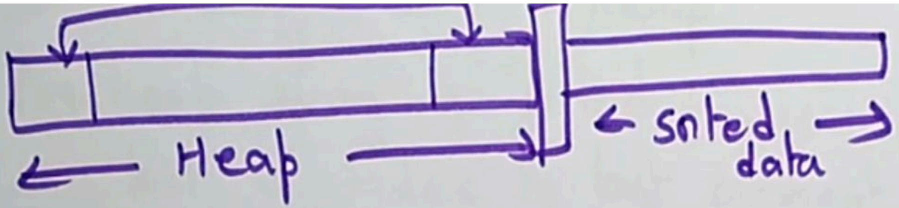
P > C

(b) Heap in its array form

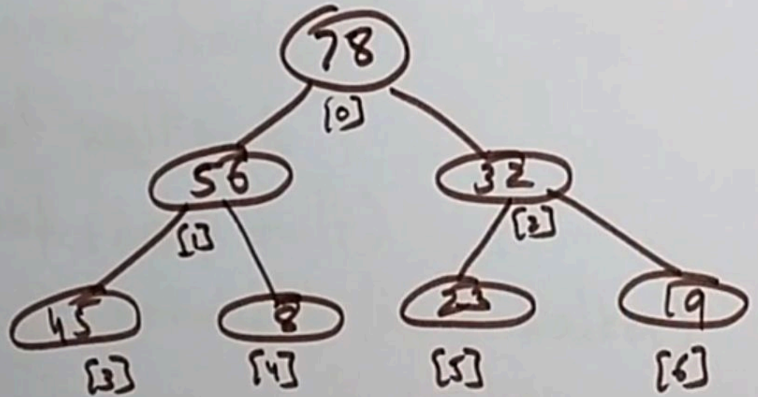
78	56	32	45	8	23	19
[0]	[1]	[2]	[3]	[4]	[5]	[6]

min heap P < C

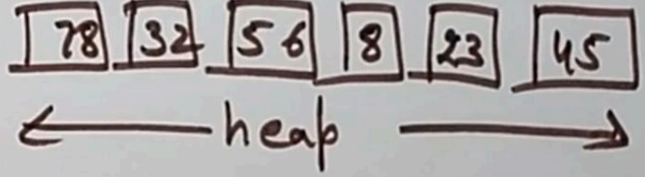
root < Child node

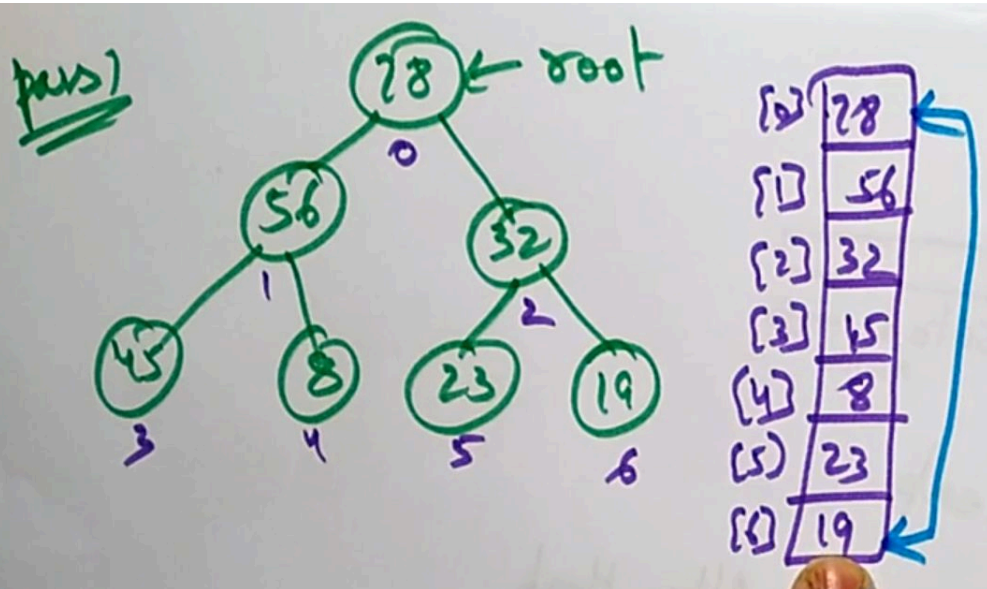


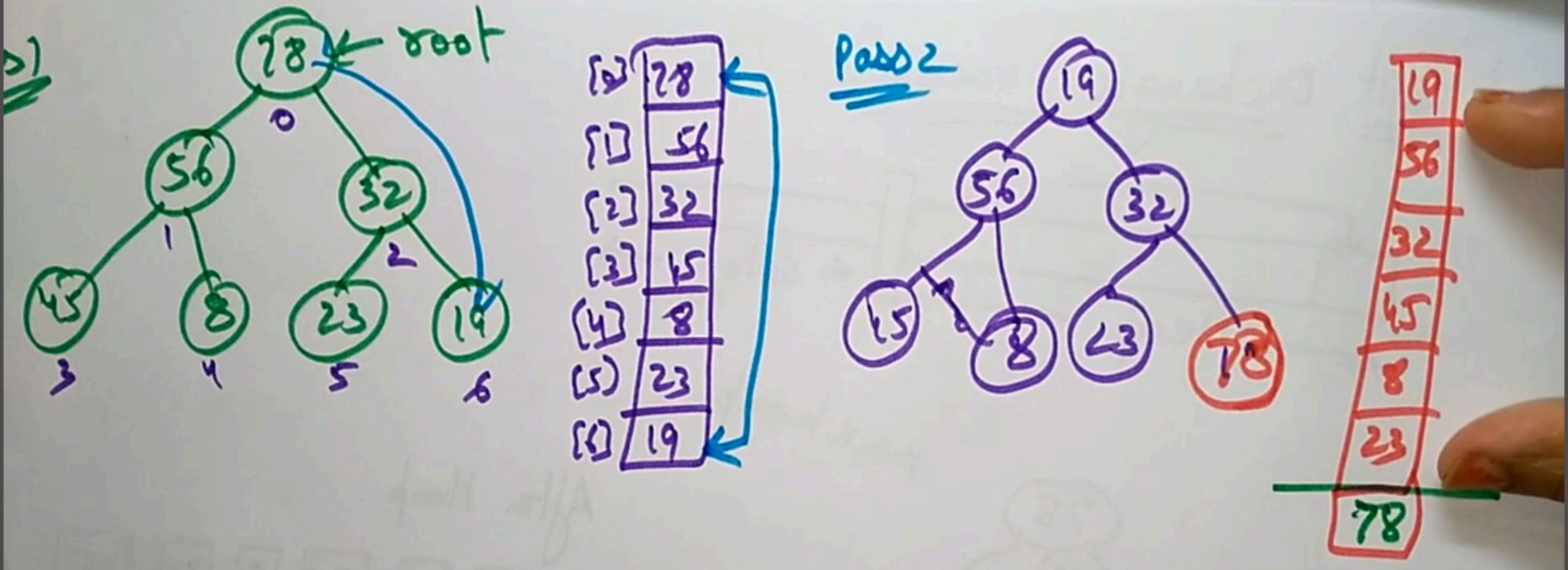
xample 1.



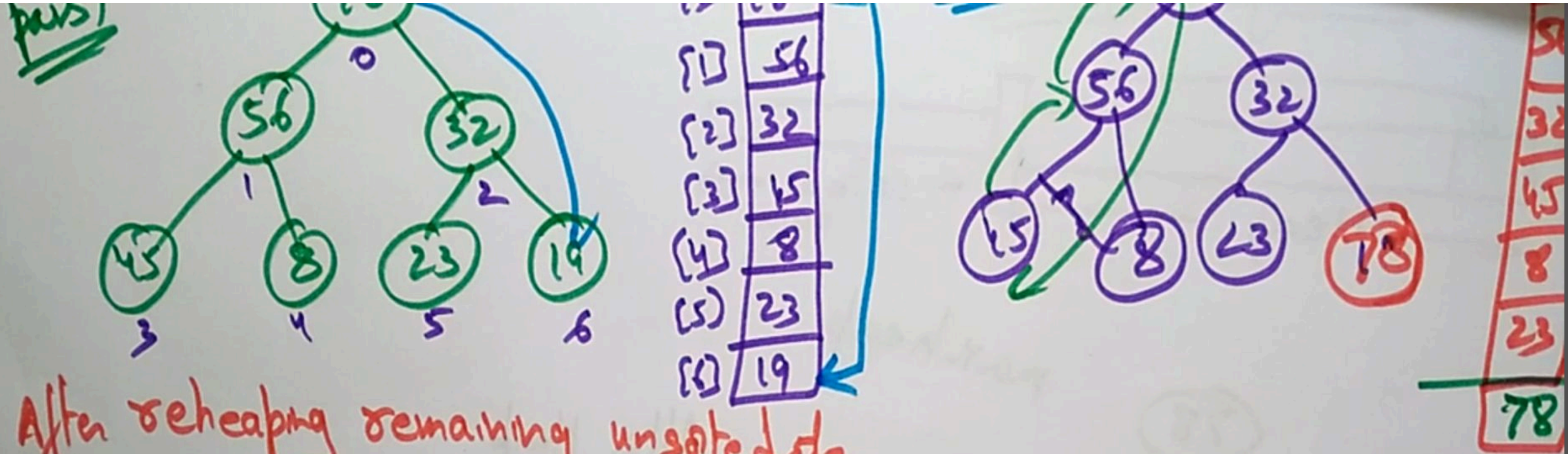
After Heap



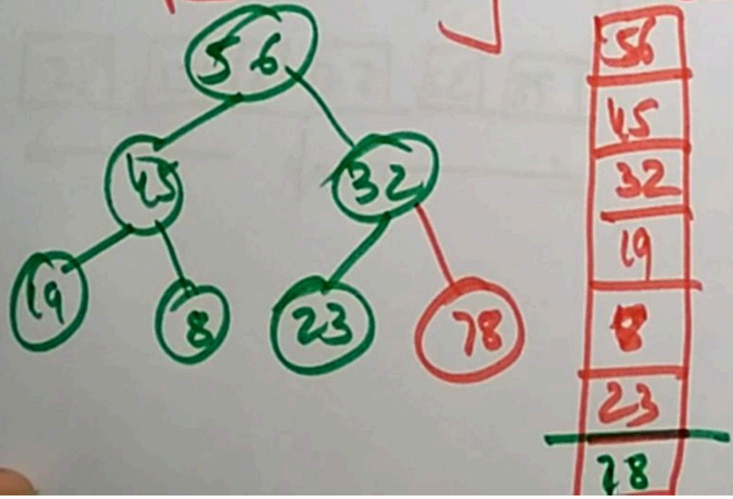

pass1
28
root
0
56
32
1
2
45
8
23
19
3
4
5
6
[0] 28
[1] 56
[2] 32
[3] 45
[4] 8
[5] 23
[6] 19

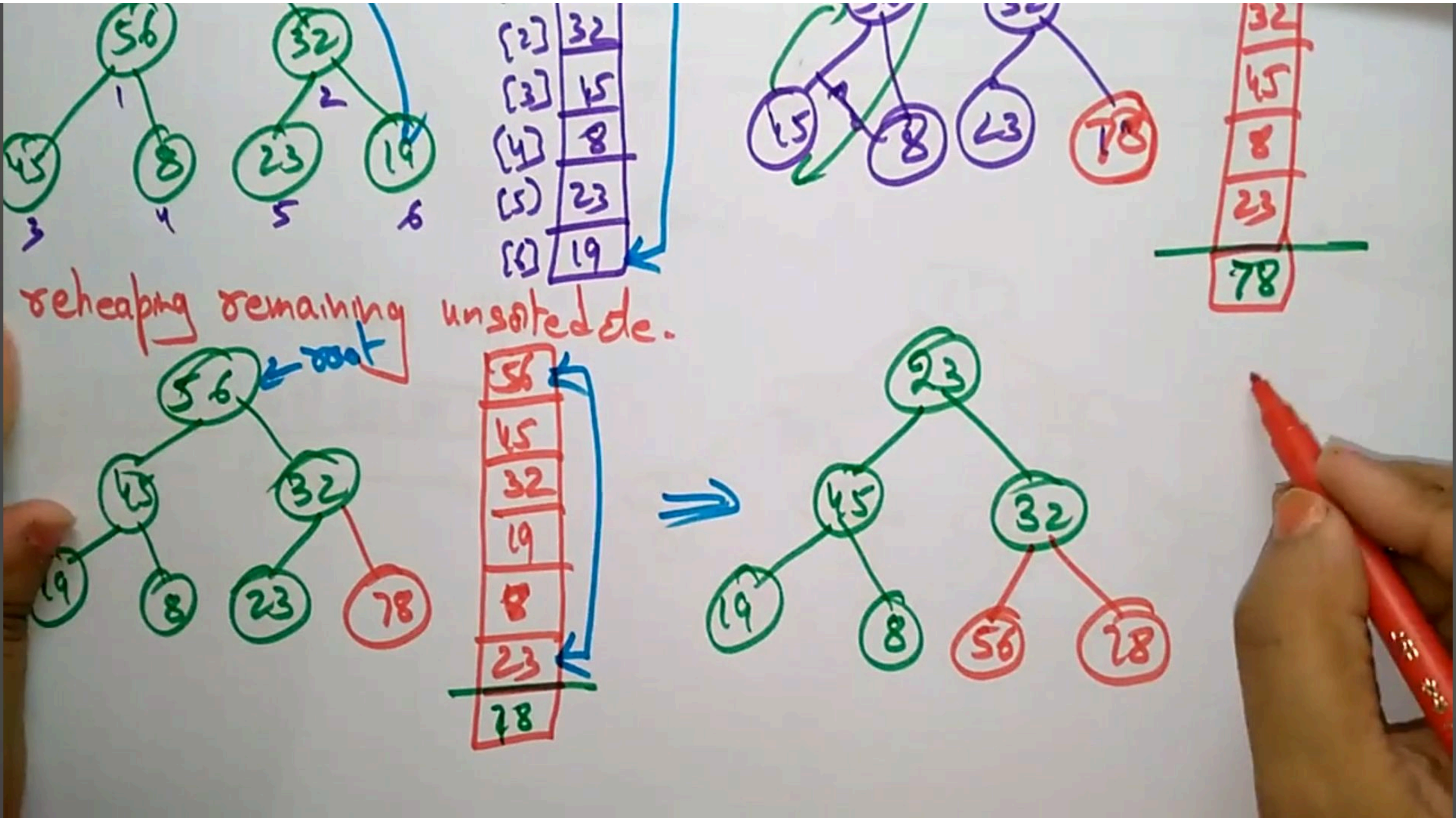


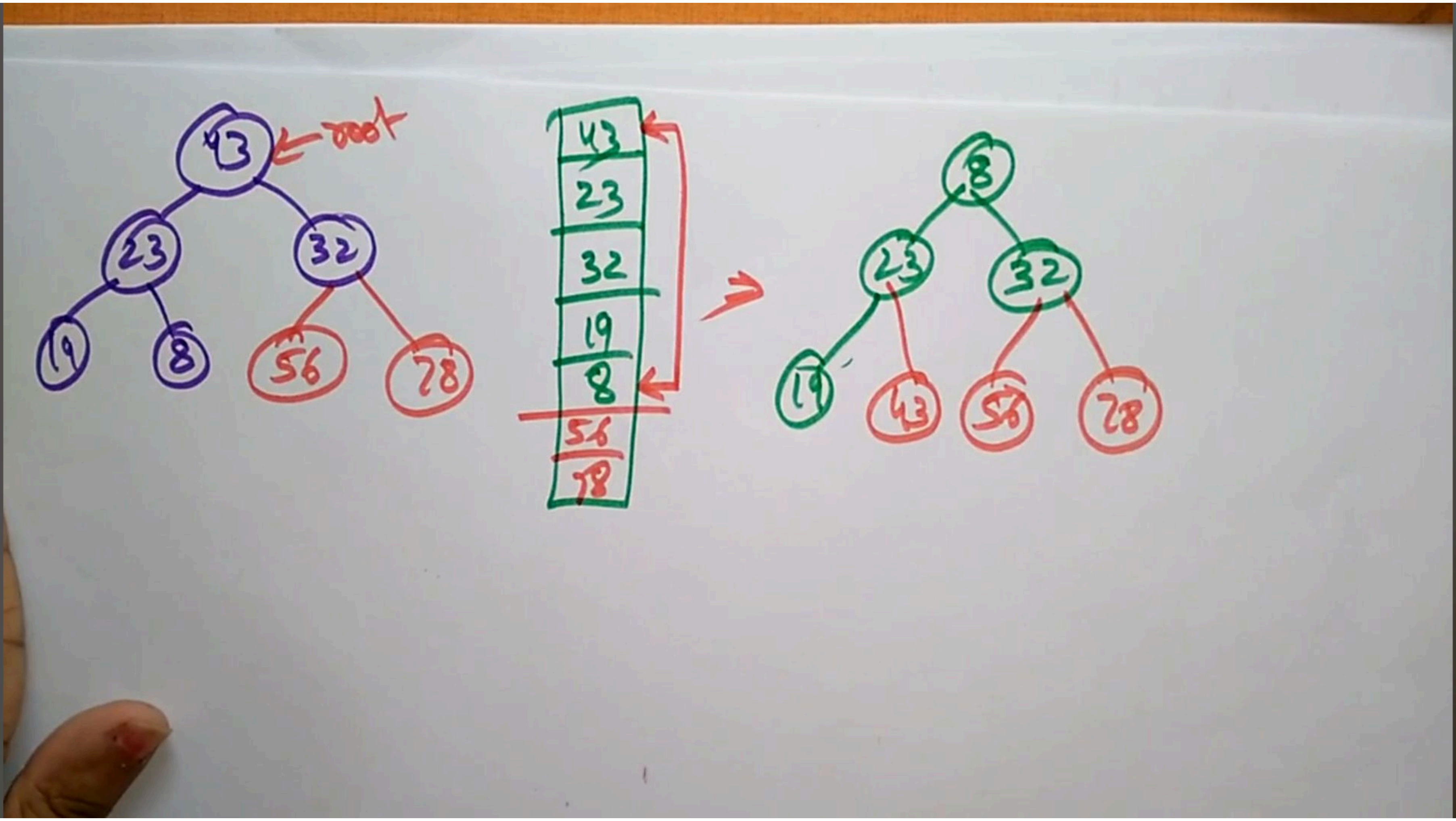
root
78
56
32
45
8
23
19
0
1
2
3
4
5
6
[0] 78
[1] 56
[2] 32
[3] 45
[4] 8
[5] 23
[6] 19
Pass 2
19
56
32
45
8
23
78
19
56
32
45
8
23
78

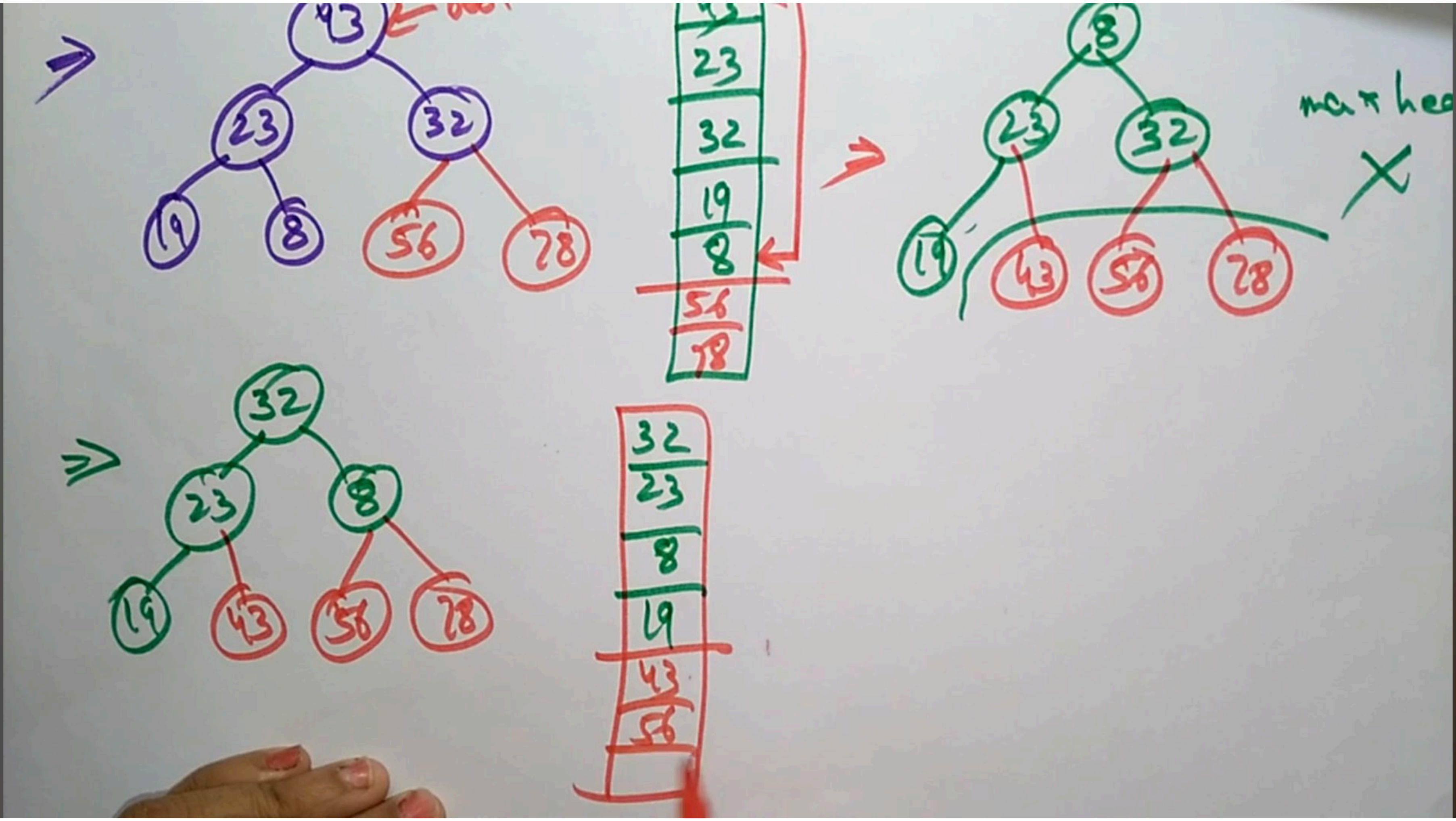


After reheaping remaining unsorted ele.

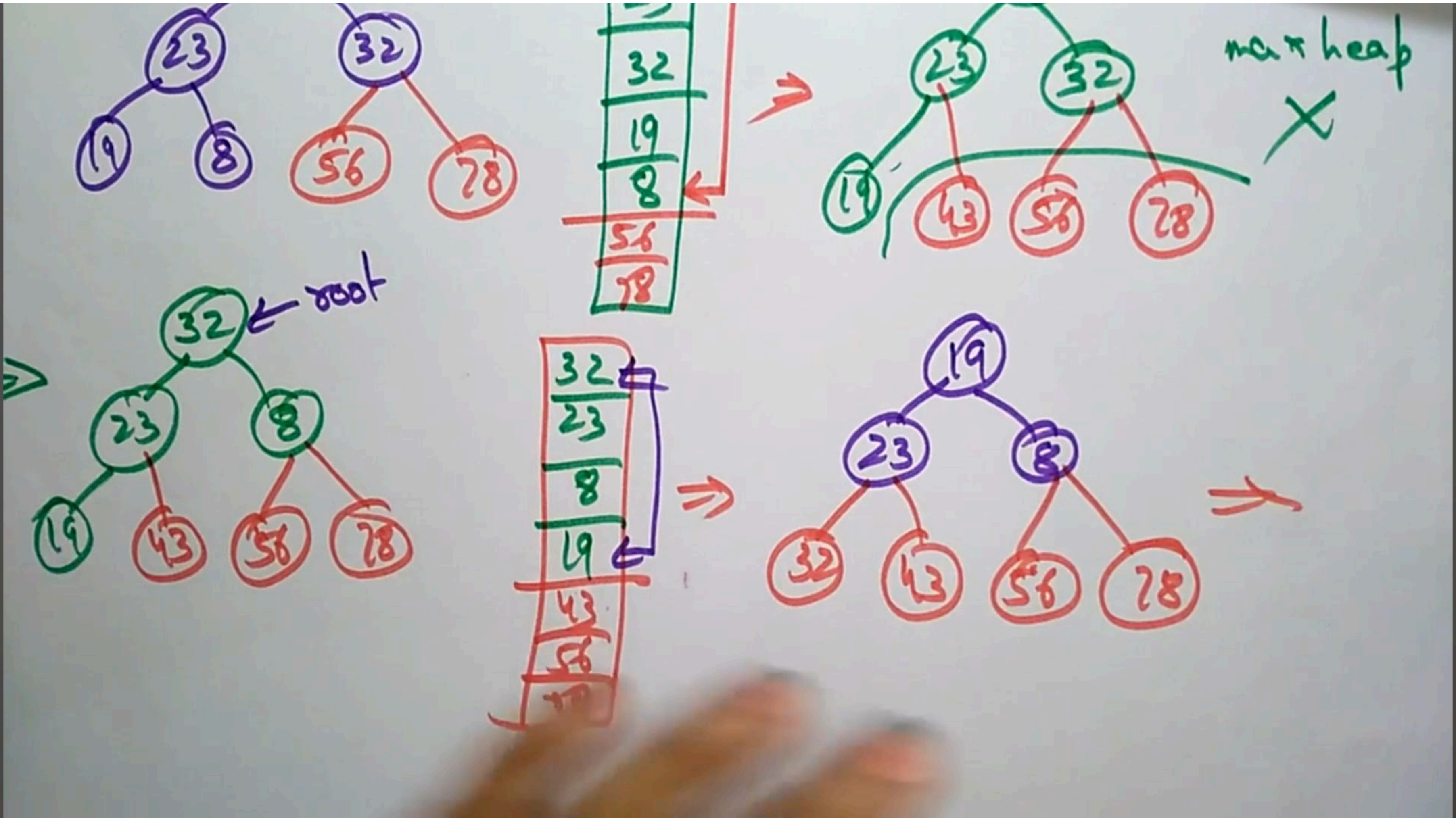


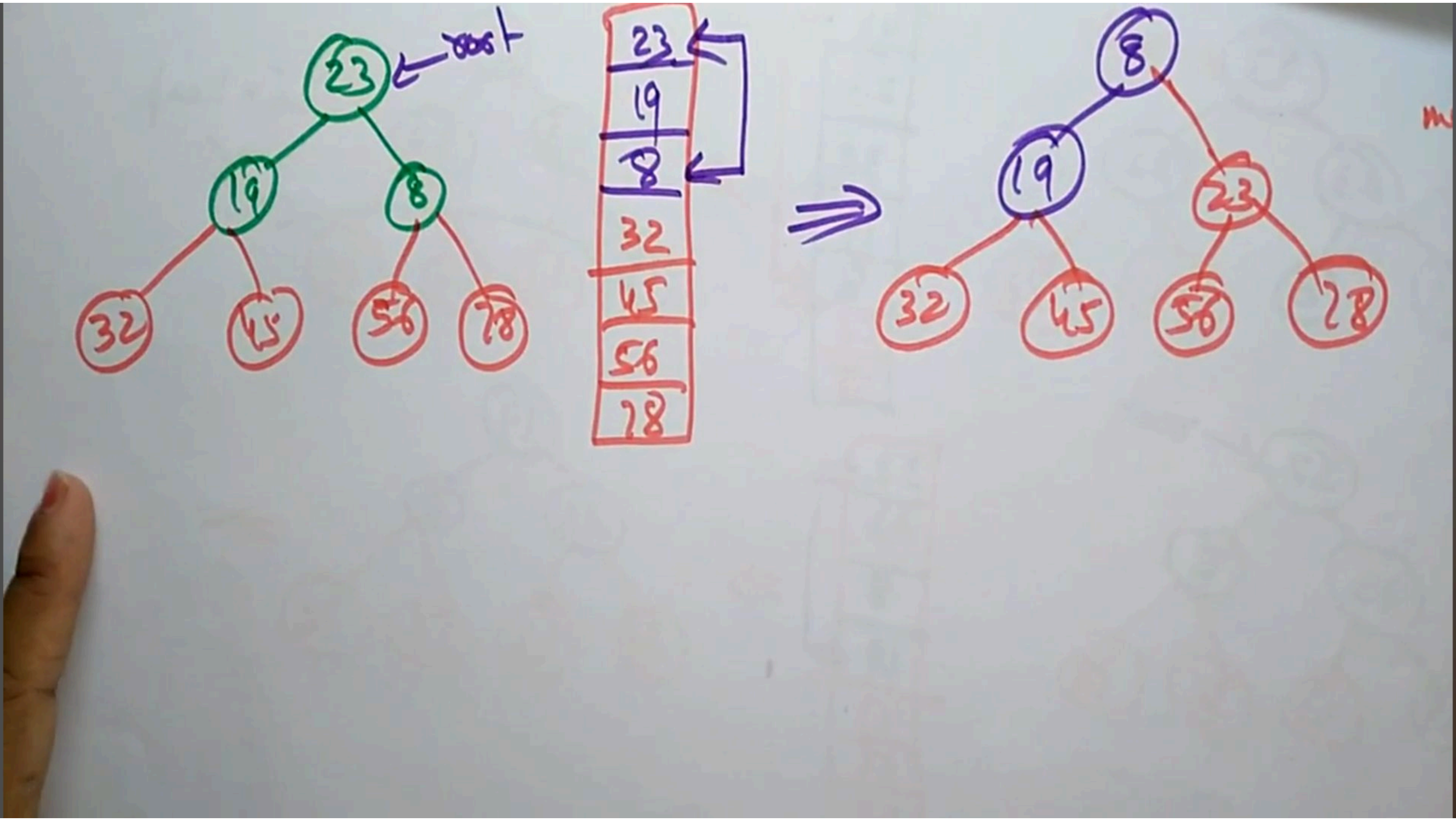

reheaping remaining unsorted ele.

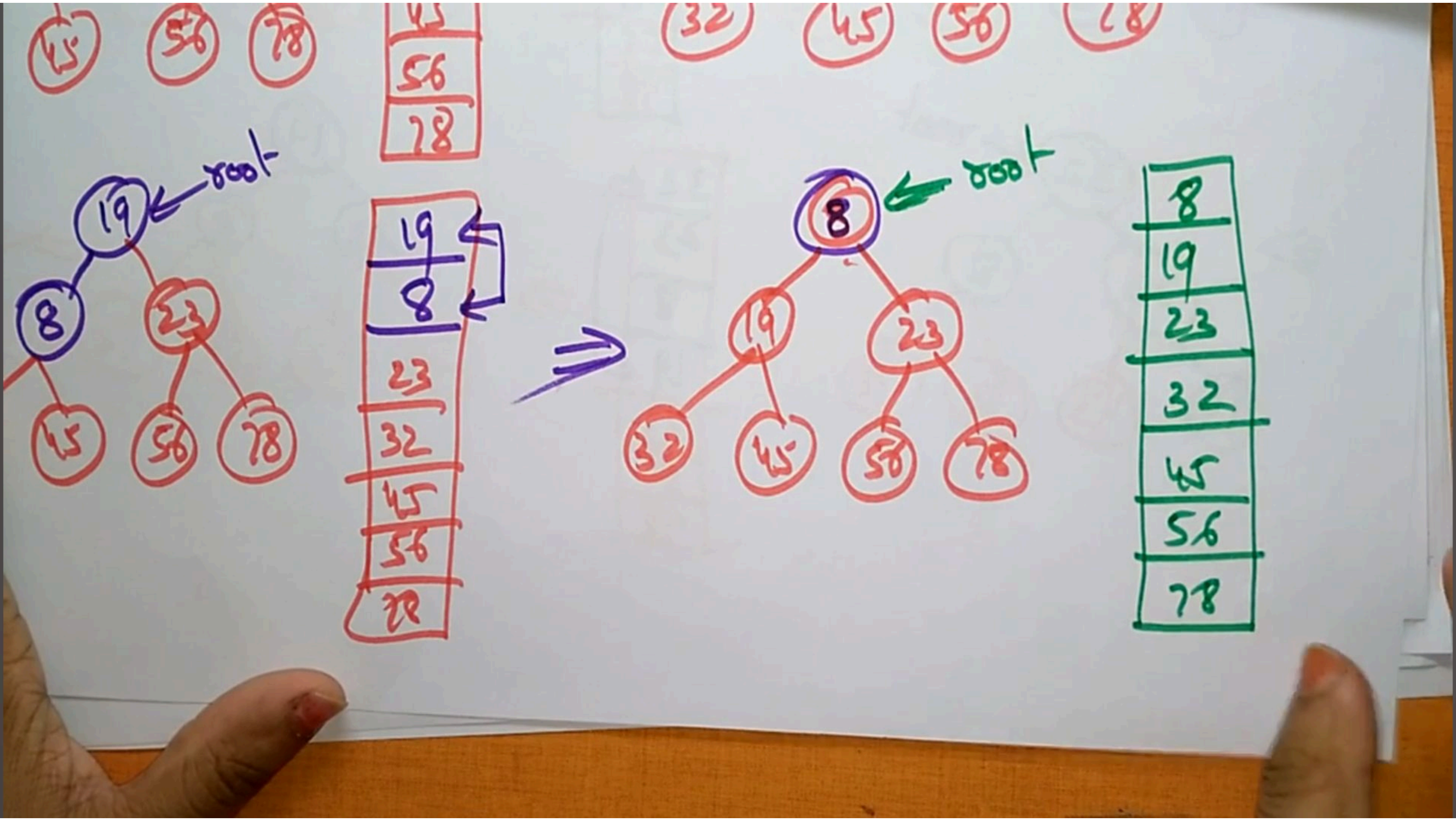

43
root
23
32
19
8
56
78
43
23
32
19
8
56
78
8
23
32
19
43
56
78



43
23
32
19
8
56
28
max hea
32
23
8
19
43
56
28

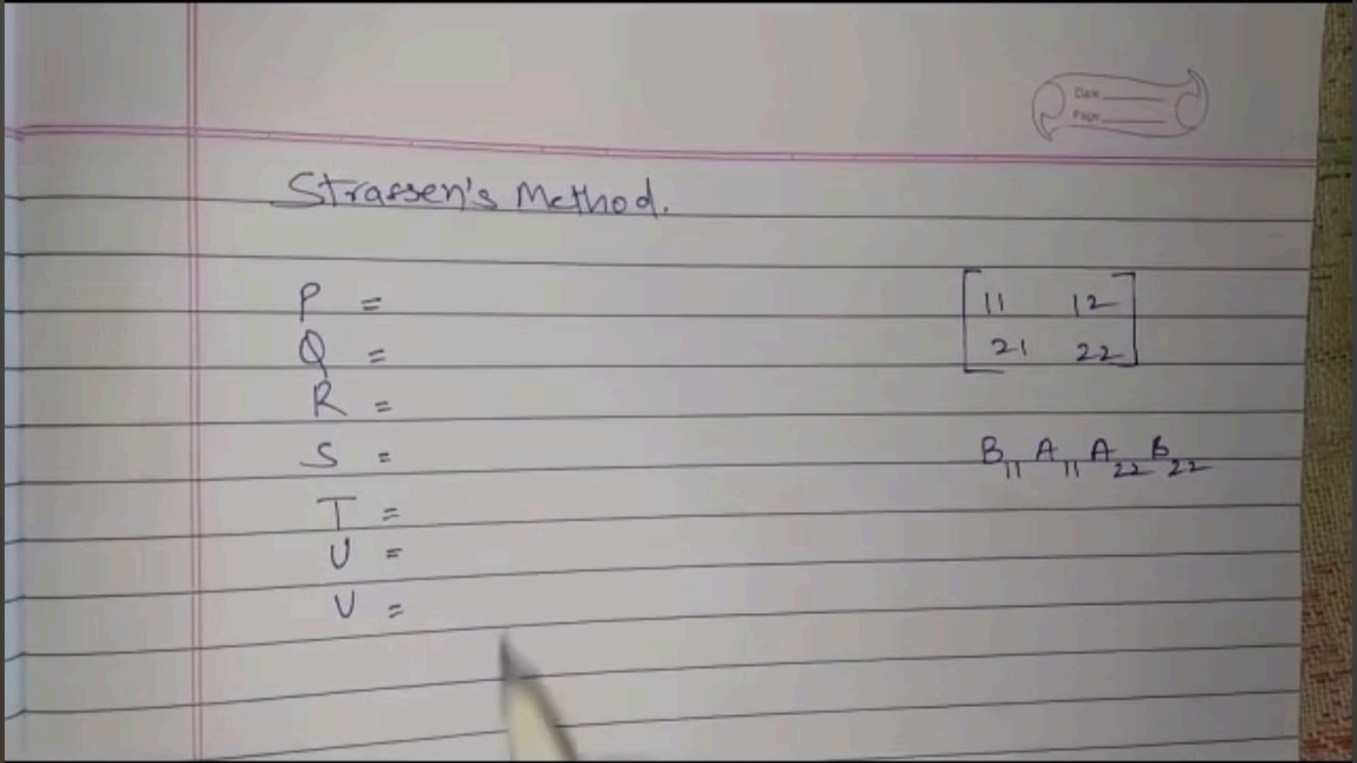

root
max heap
32
23
19
8
56
78
43


root
23
19
8
32
45
56
28
23
19
8
32
45
56
28
8
19
23
32
45
56
28



45
56
78
56
78
root
19
8
23
45
56
78
19
8
23
32
45
56
78
32
45
56
78
root
8
19
23
32
45
56
78
8
19
23
32
45
56
78

Strassen's matrix multiplication

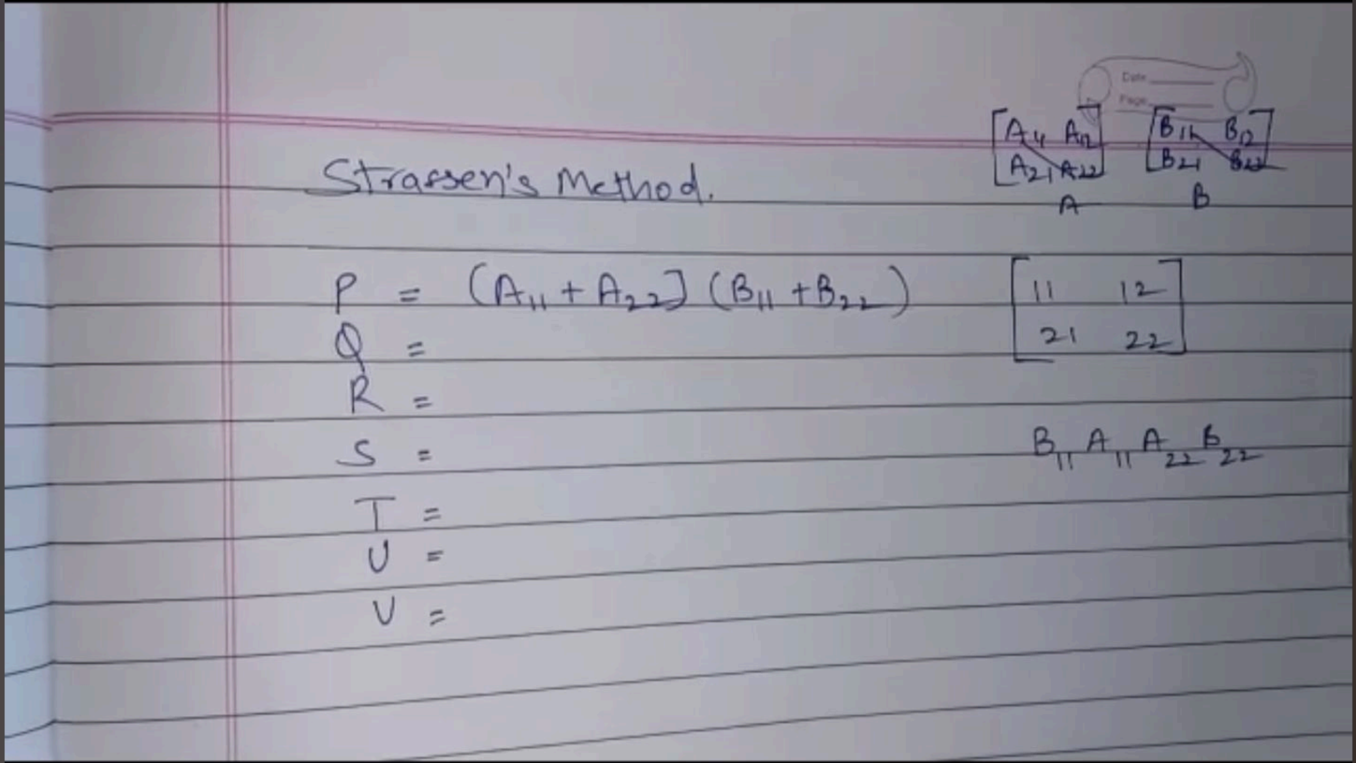


Strassen's Method.

P =
Q =
R =
S =
T =
U =
V =

$$\begin{bmatrix} 11 & 12 \\ 21 & 22 \end{bmatrix}$$

B_{11} A_{11} A_{22} B_{22}



Strassen's Method.

$$\underset{A}{\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}} \quad \underset{B}{\begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}}$$

$P = (A_{11} + A_{22}] (B_{11} + B_{22})$

$Q =$

$R =$

$S =$

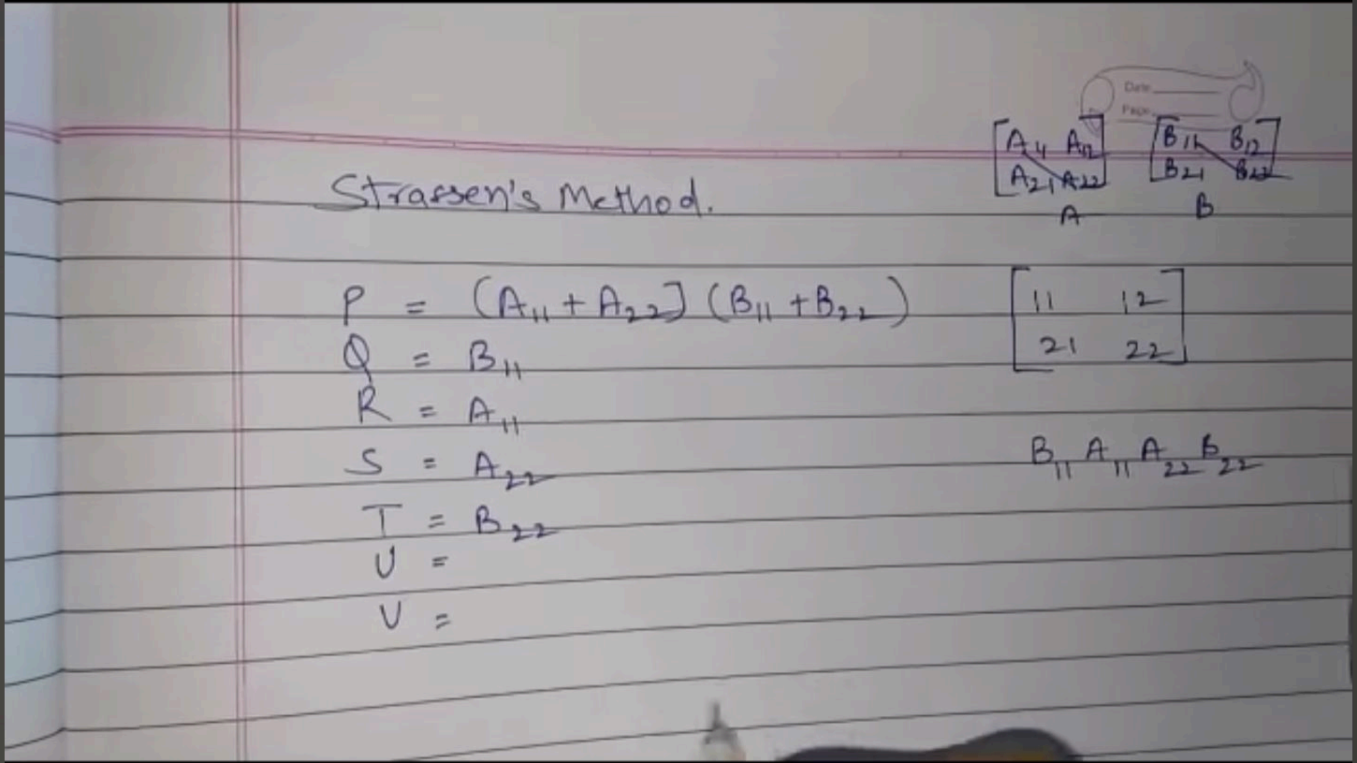
$T =$

$U =$

$V =$

$$\begin{bmatrix} 11 & 12 \\ 21 & 22 \end{bmatrix}$$

$B_{11} \; A_{11} \; A_{22} \; B_{22}$



Strassen's Method.

$$\underset{A}{\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}} \quad \underset{B}{\begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}}$$

$$\begin{bmatrix} 11 & 12 \\ 21 & 22 \end{bmatrix}$$

$B_{11}\ A_{11}\ A_{22}\ B_{22}$

$P = (A_{11} + A_{22}] (B_{11} + B_{22})$

$Q = B_{11}$

$R = A_{11}$

$S = A_{22}$

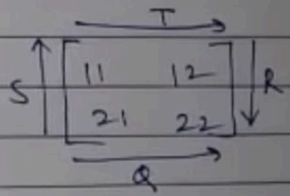
$T = B_{22}$

$U =$

$V =$

$$\underset{A}{\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}} \quad \underset{B}{\begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}}$$

Strassen's Method.



$B_{11}\ A_{11}\ A_{22}\ B_{22}$

$P = (A_{11} + A_{22})(B_{11} + B_{22})$

$Q = B_{11}$

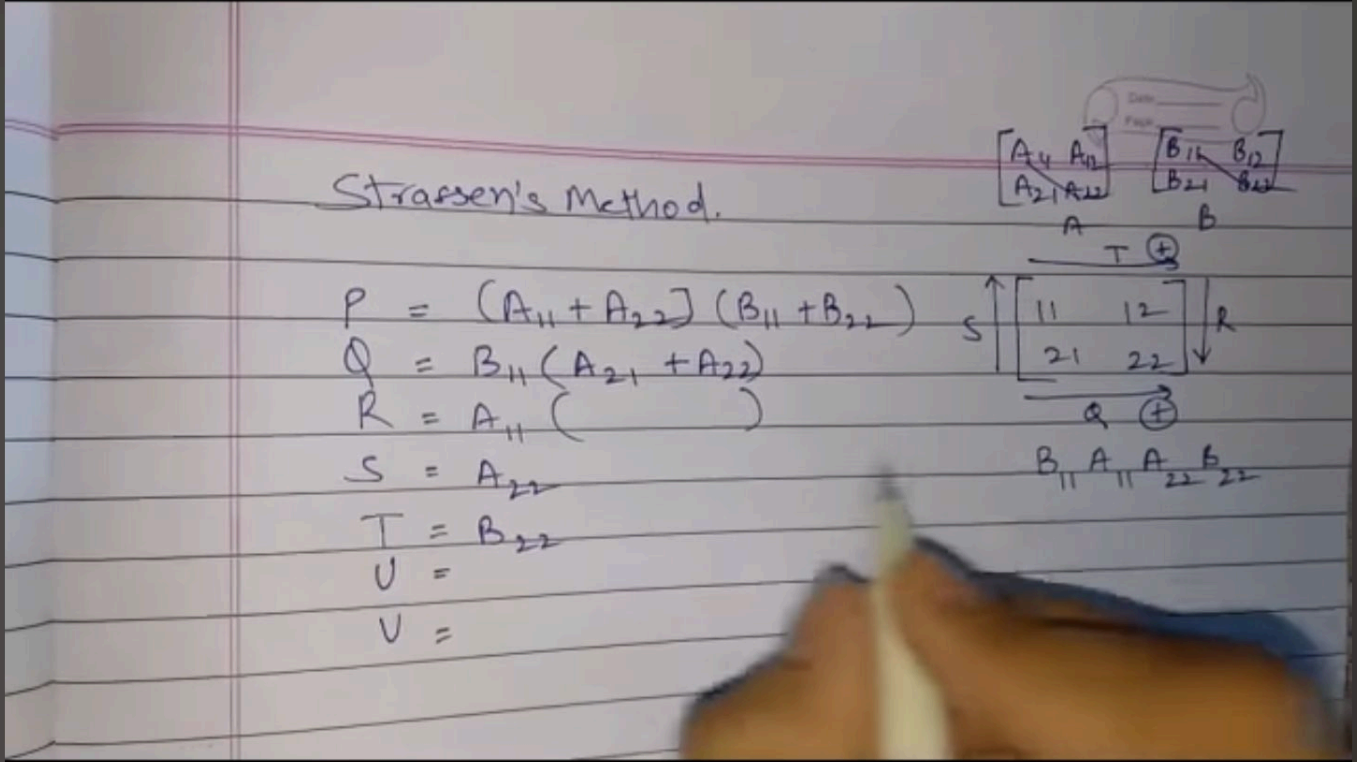
$R = A_{11}$

$S = A_{22}$

$T = B_{22}$

$U =$

$V =$



Strassen's Method.

$P = (A_{11} + A_{22}] (B_{11} + B_{22})$

$Q = B_{11} (A_{21} + A_{22})$

$R = A_{11} (\quad\quad)$

$S = A_{22}$

$T = B_{22}$

$U =$

$V =$

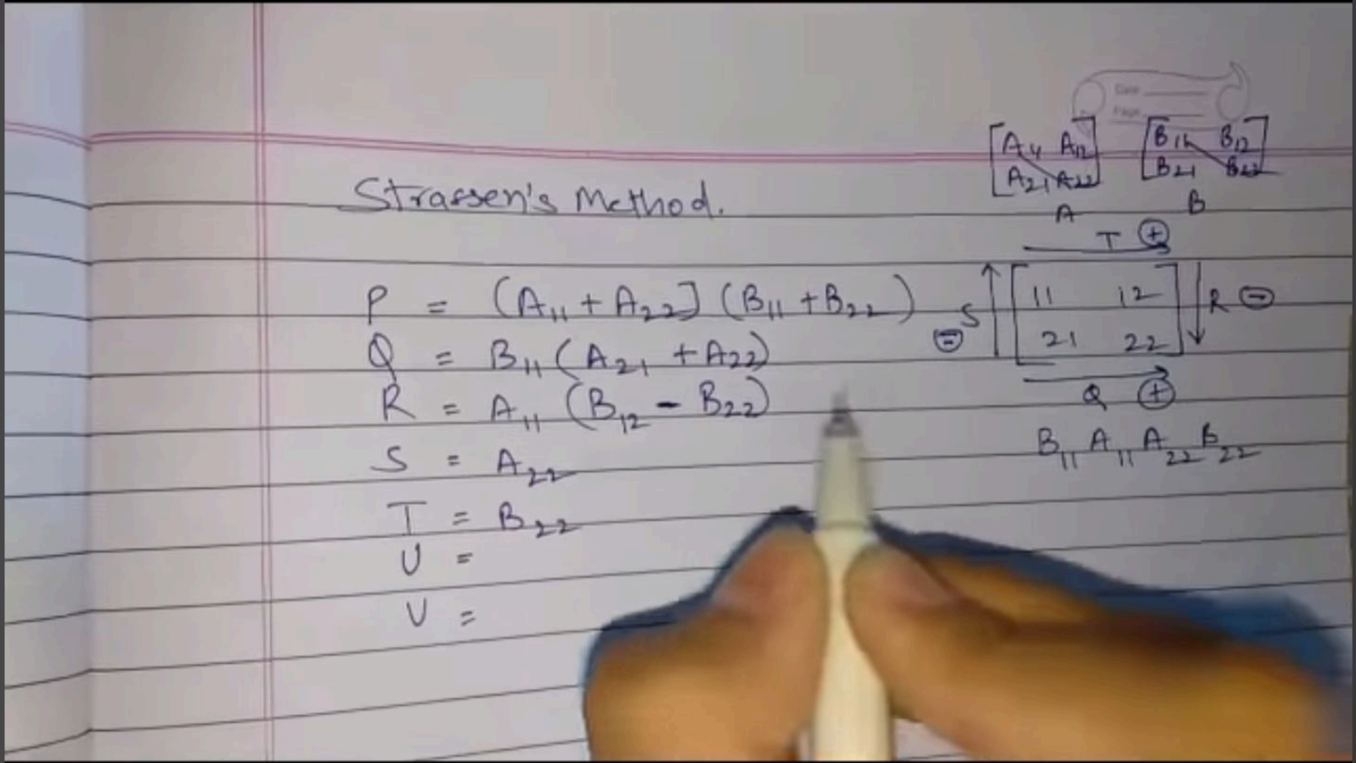
$$\underset{A}{\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}} \quad \underset{B}{\begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}}$$

$$\overset{T \; \oplus}{\longrightarrow}$$

$$S \uparrow \begin{bmatrix} 11 & 12 \\ 21 & 22 \end{bmatrix} \downarrow R$$

$$\underset{Q \; \oplus}{\longrightarrow}$$

$B_{11} \; A_{11} \; A_{22} \; B_{22}$



Strassen's Method.

$P = (A_{11} + A_{22})(B_{11} + B_{22})$

$Q = B_{11}(A_{21} + A_{22})$

$R = A_{11}(B_{12} - B_{22})$

$S = A_{22}$

$T = B_{22}$

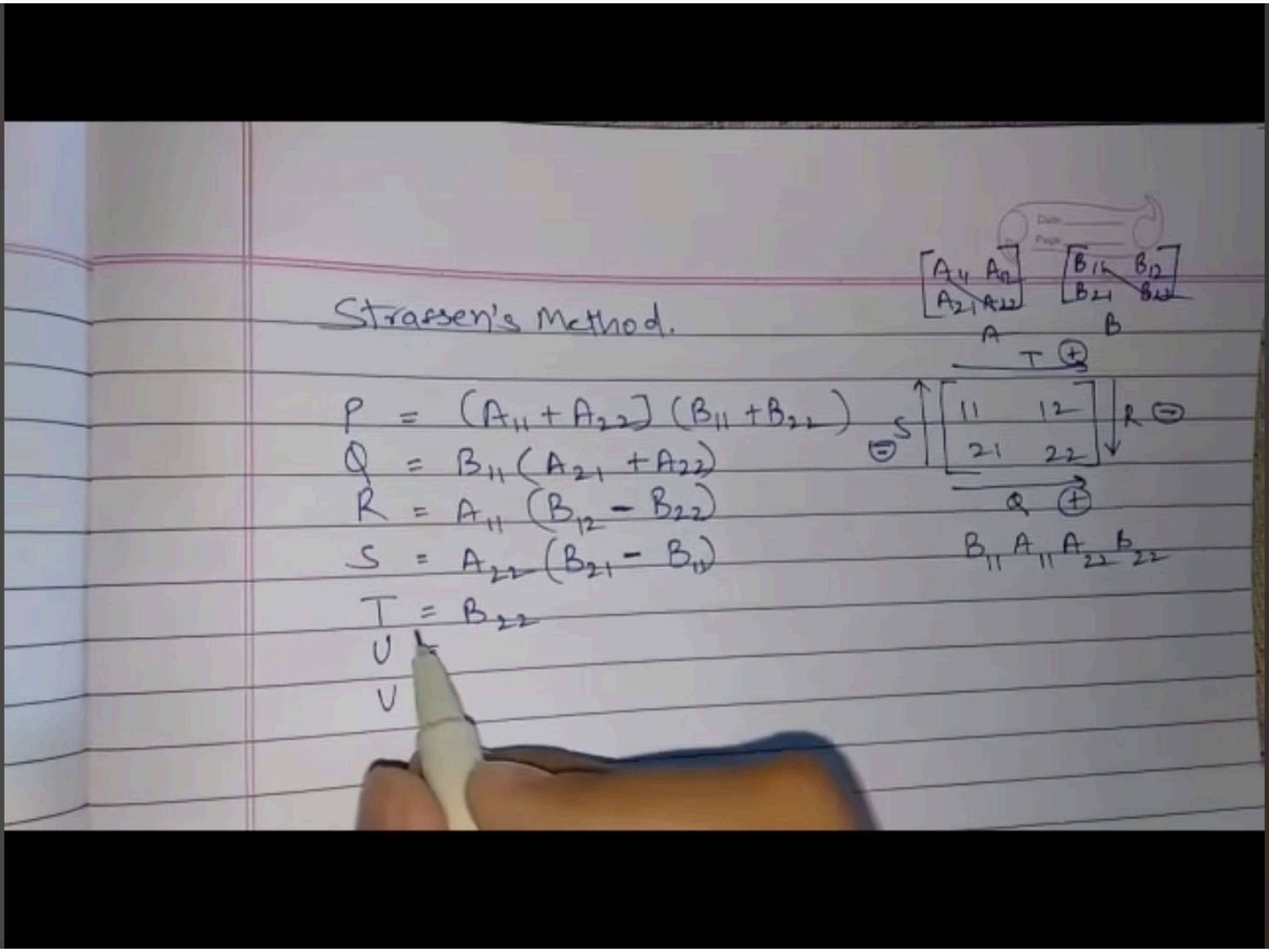
$U =$

$V =$

$\underset{A}{\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}}$ $\underset{B}{\begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}}$

T ⊕
S ⊖
R ⊖
11 12
21 22
Q ⊕

$B_{11}\ A_{11}\ A_{22}\ B_{22}$



Strassen's Method.

$P = (A_{11} + A_{22})(B_{11} + B_{22})$

$Q = B_{11}(A_{21} + A_{22})$

$R = A_{11}(B_{12} - B_{22})$

$S = A_{22}(B_{21} - B_{11})$

$T = B_{22}$

U

V

$$\underset{A}{\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}} \quad \underset{B}{\begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}}$$

$$\begin{array}{ccc} & T \oplus & \\ \ominus\ S \uparrow & \begin{bmatrix} 11 & 12 \\ 21 & 22 \end{bmatrix} & \downarrow R \ominus \\ & Q \oplus & \end{array}$$

$B_{11}\ A_{11}\ A_{22}\ B_{22}$

Strassen's Method.

$P = (A_{11} + A_{22}] (B_{11} + B_{22})$

$Q = B_{11} (A_{21} + A_{22})$

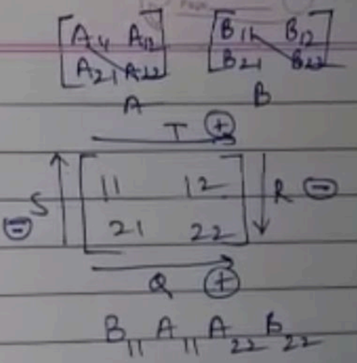
$R = A_{11} (B_{12} - B_{22})$

$S = A_{22} (B_{21} - B_{11})$

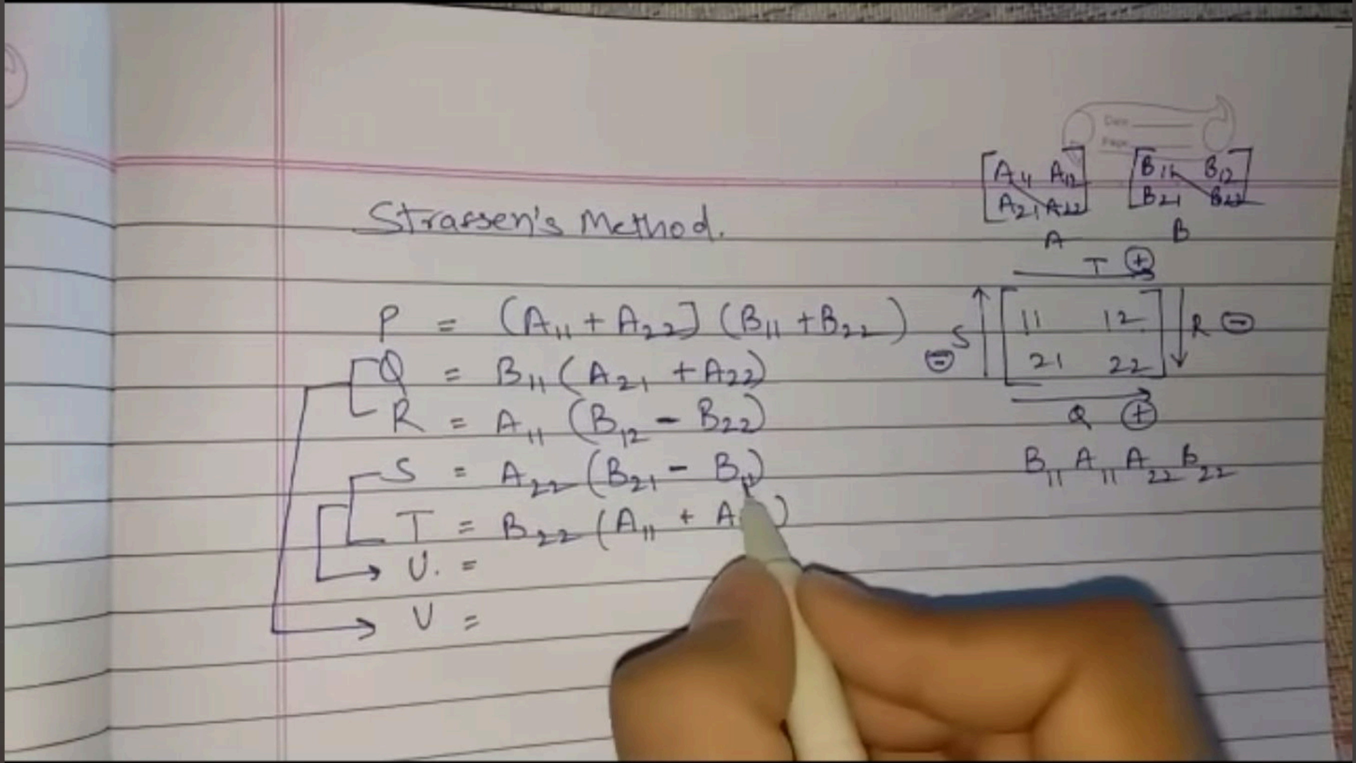
$T = B_{22} (A_{11} + A_{12})$

$U =$

$V =$



$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$ A
$\begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$ B
T (+)
S (−)
R (−)
11 12
21 22
Q (+)
B_{11} A_{11} A_{22} B_{22}



Strassen's Method.

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \quad \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

A B

T ⊕

$$\begin{array}{ccc} & 11 & 12 \\ S\ \ominus & & & R\ \ominus \\ & 21 & 22 \end{array}$$

Q ⊕

$B_{11}\ A_{11}\ A_{22}\ B_{22}$

$P = (A_{11} + A_{22})(B_{11} + B_{22})$

$Q = B_{11}(A_{21} + A_{22})$

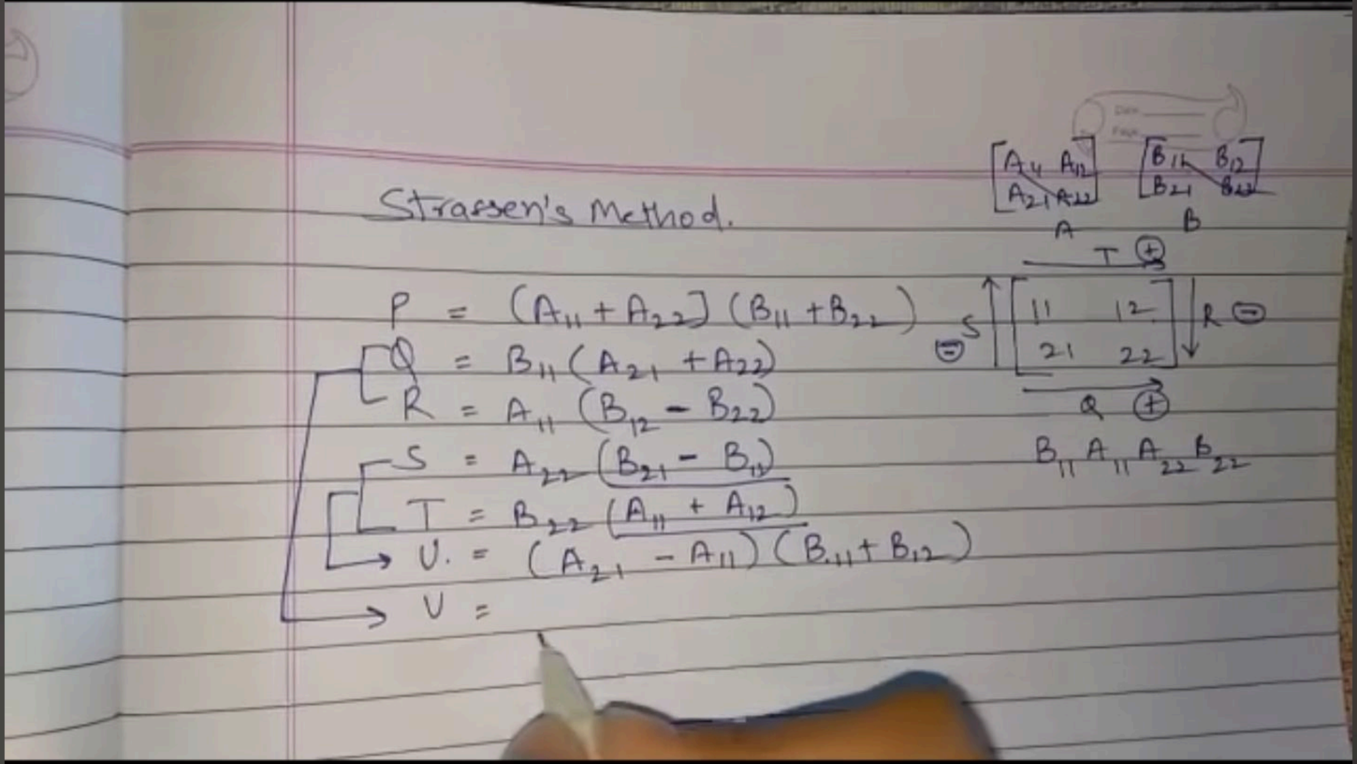
$R = A_{11}(B_{12} - B_{22})$

$S = A_{22}(B_{21} - B_{11})$

$T = B_{22}(A_{11} + A\ \ )$

$U. =$

$V =$



Strassen's Method.

$$\underset{A}{\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}} \quad \underset{B}{\begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}}$$

$P = (A_{11} + A_{22})(B_{11} + B_{22})$

$Q = B_{11}(A_{21} + A_{22})$

$R = A_{11}(B_{12} - B_{22})$

$S = A_{22}(B_{21} - B_{11})$

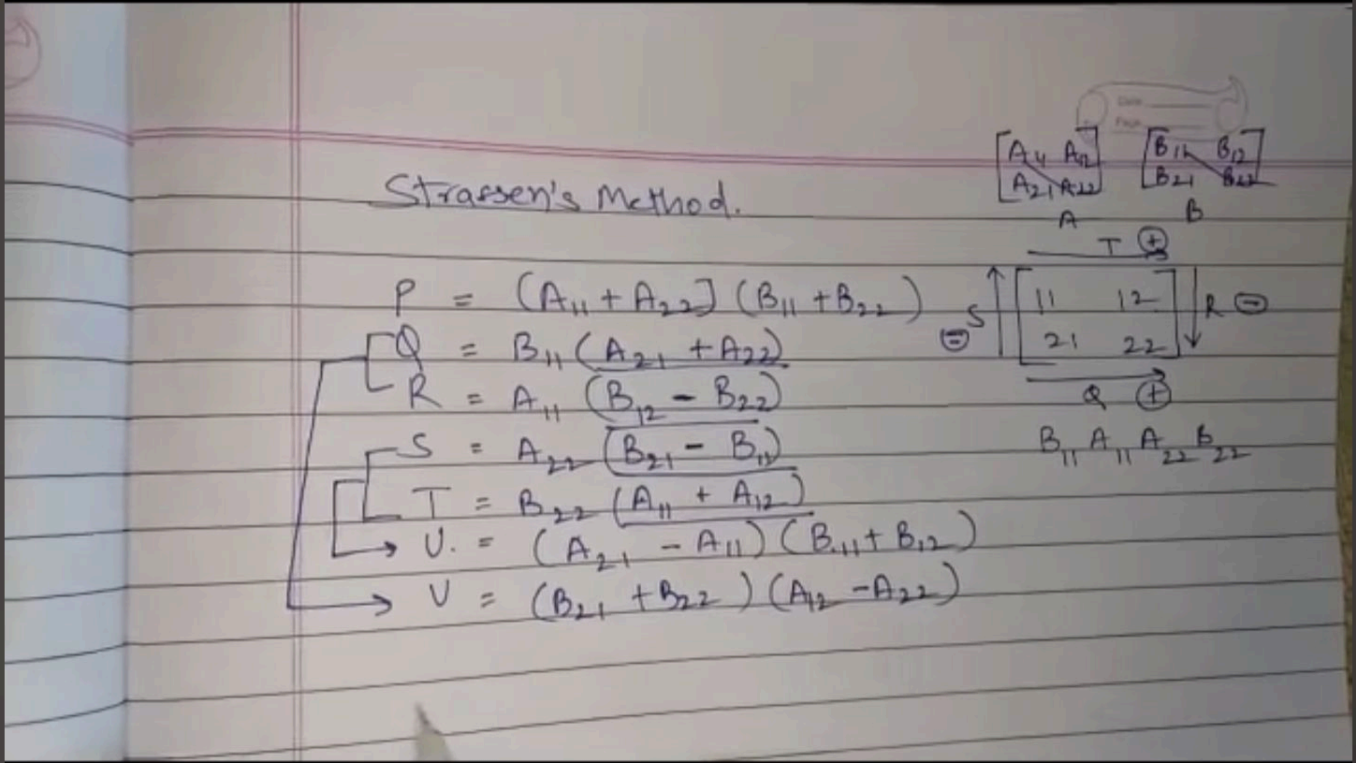
$T = B_{22}(A_{11} + A_{12})$

$U = (A_{21} - A_{11})(B_{11} + B_{12})$

$V =$

$$\begin{array}{c} T \oplus \\ \ominus\, S \begin{bmatrix} 11 & 12 \\ 21 & 22 \end{bmatrix} R \ominus \\ Q \oplus \end{array}$$

$B_{11}\ A_{11}\ A_{22}\ B_{22}$



Strassen's Method.

$P = (A_{11} + A_{22})(B_{11} + B_{22})$

$Q = B_{11}(A_{21} + A_{22})$

$R = A_{11}(B_{12} - B_{22})$

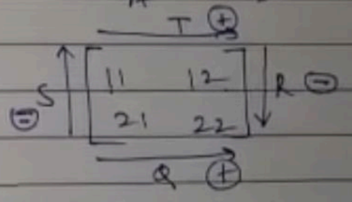
$S = A_{22}(B_{21} - B_{11})$

$T = B_{22}(A_{11} + A_{12})$

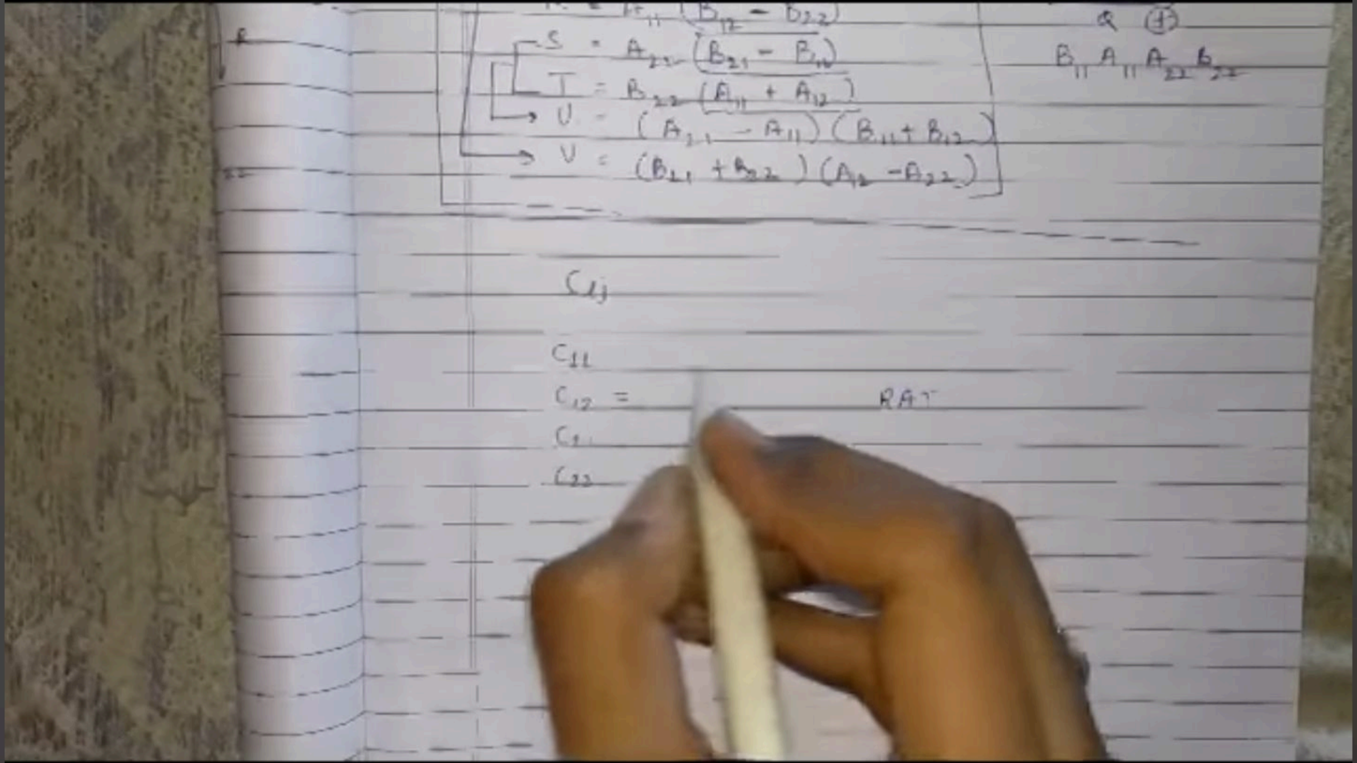
$U = (A_{21} - A_{11})(B_{11} + B_{12})$

$V = (B_{21} + B_{22})(A_{12} - A_{22})$

$$\begin{array}{cc} \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} & \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \\ A & B \end{array}$$



$B_{11}\ A_{11}\ A_{22}\ B_{22}$



$R = A_{11}(B_{12} - B_{22})$

$S = A_{22}(B_{21} - B_{11})$

$T = B_{22}(A_{11} + A_{12})$

$U = (A_{21} - A_{11})(B_{11} + B_{12})$

$V = (B_{21} + B_{22})(A_{12} - A_{22})$

Q $\boxed{+}$

$B_{11}\ A_{11}\ A_{22}\ B_{22}$

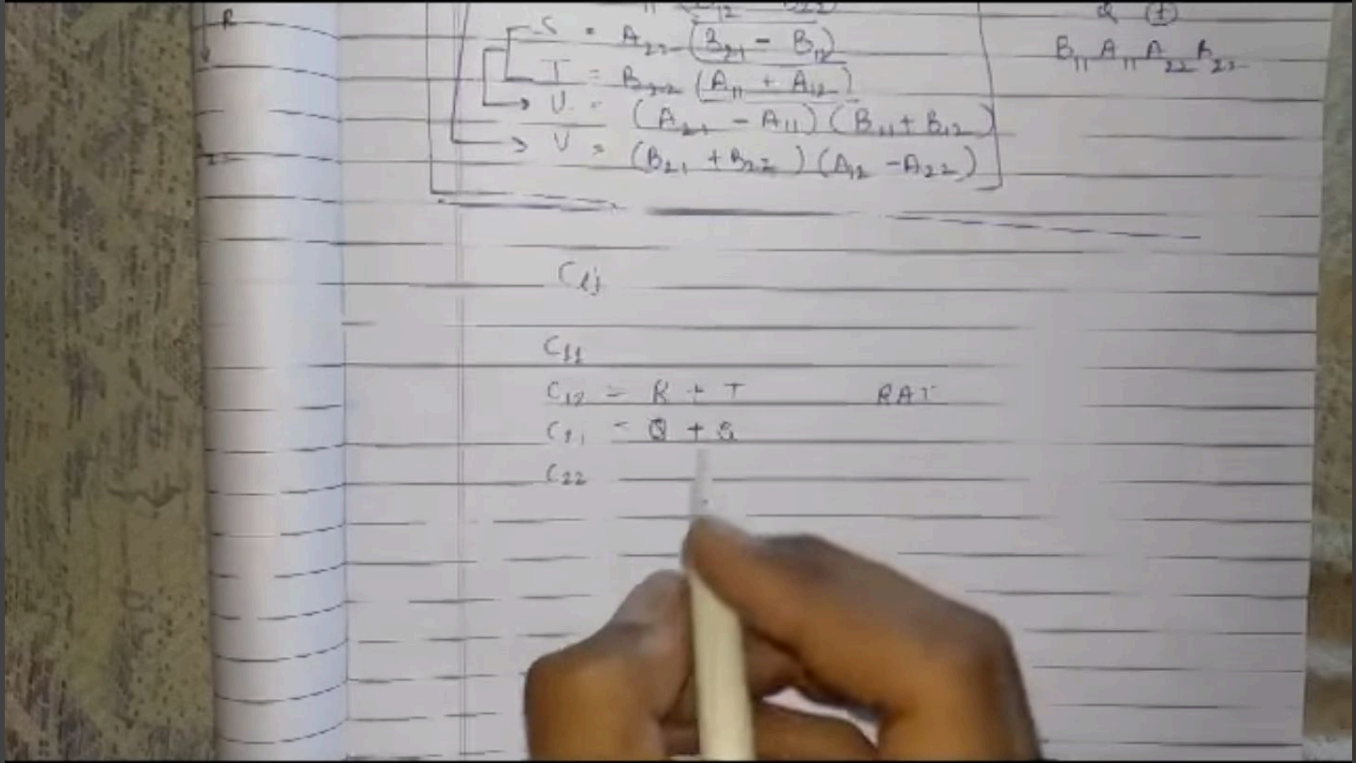
C_{ij}

C_{11}

$C_{12} =$ RAT

C_{21}

C_{22}



$S = A_{22}(B_{21} - B_{12})$

$T = B_{22}(A_{11} + A_{12})$

$U = (A_{21} - A_{11})(B_{11} + B_{12})$

$V = (B_{21} + B_{22})(A_{12} - A_{22})$

Q (+)

$B_{11}\ A_{11}\ A_{22}\ A_{22}$

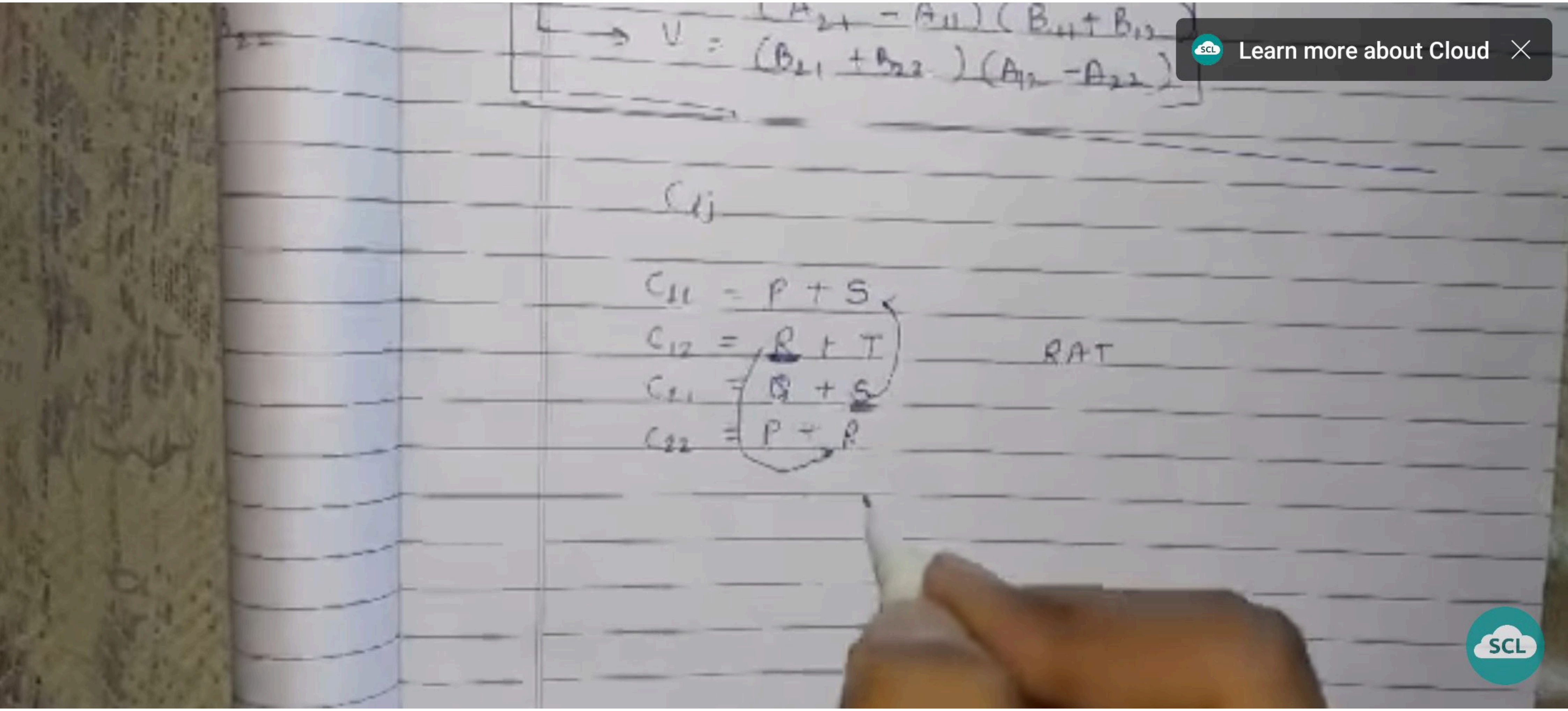
C_{ij}

C_{11}

$C_{12} = R + T$ RAT

$C_{21} = Q + S$

C_{22}



$(A_{21} - A_{11})(B_{11} + B_{12})$

$\rightarrow V = (B_{21} + B_{22})(A_{12} - A_{22})$

C_{ij}

$C_{11} = P + S$

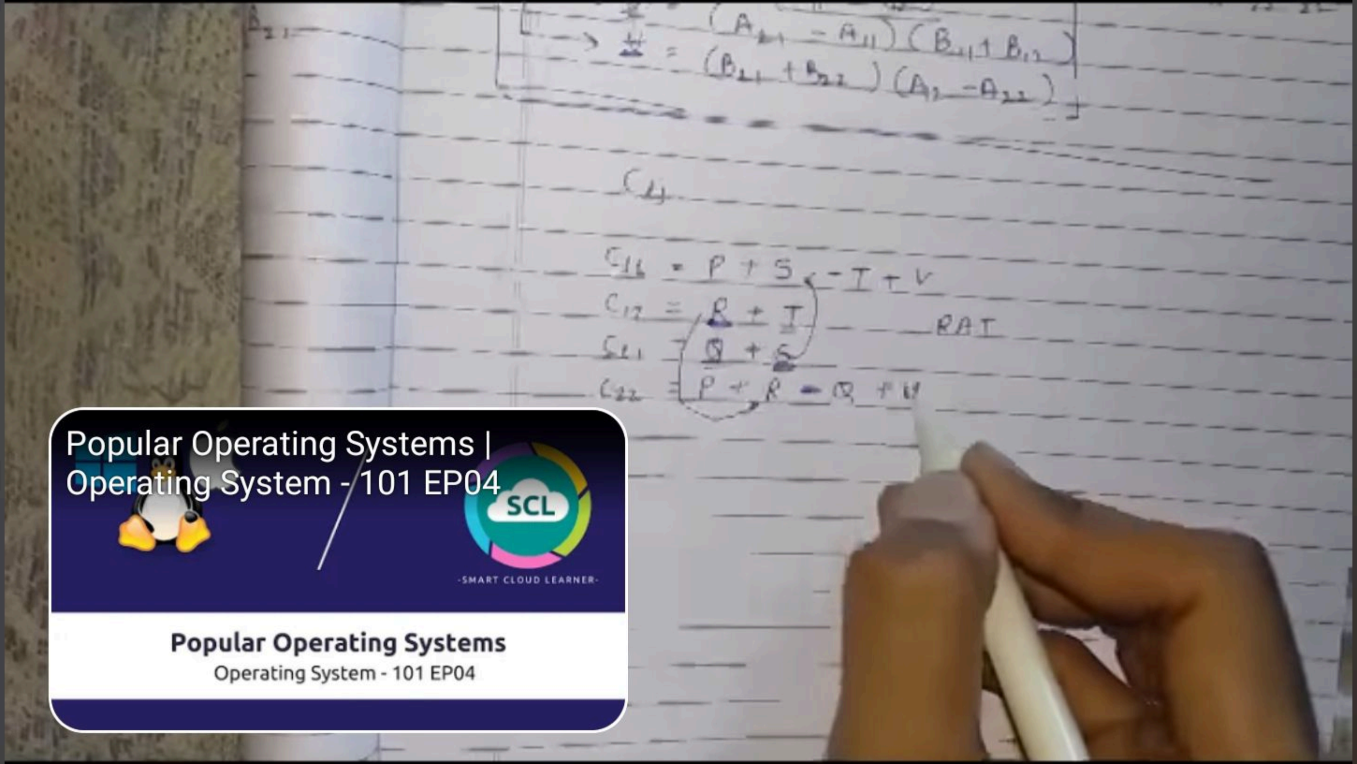
$C_{12} = R + T$

$C_{21} = Q + S$

$C_{22} = P + R$

RAT

Learn more about Cloud



$(A_{21} - A_{11})(B_{11} + B_{12})$

$\rightarrow V = (B_{21} + B_{22})(A_{12} - A_{22})$

C_{ij}

$C_{11} = P + S - T + V$

$C_{12} = R + T$

$C_{21} = Q + S$

$C_{22} = P + R - Q + U$

RAT

Popular Operating Systems | Operating System - 101 EP04

SCL

-SMART CLOUD LEARNER-

Popular Operating Systems

Operating System - 101 EP04