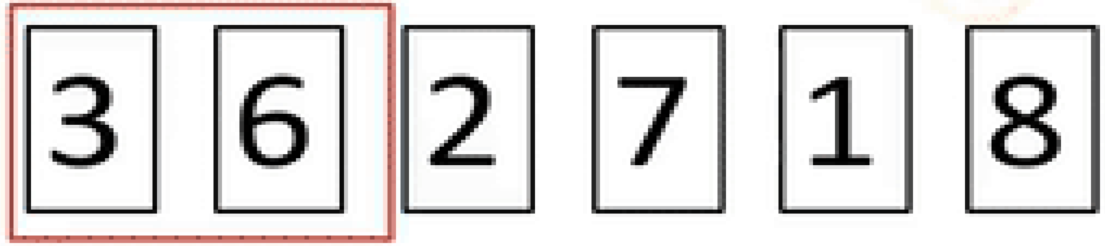
Sorting

```
2. External Sorting:
-The data that is to be sorted cannot be accomadate in the memory at the
same time and require some additional auxillary memory, then it is called
external sorting.

arr = {1,2,6,9,3,5,8};

Stable and Not stable sorting:
------------------------------
arr = {1,2,6,9,3,5,8,6,9,3,5,8};

arr = {1,2,3,3,5,5,6,6,8,8,9,9};

or
arr = {1,2,6,9,3,5,8,6,9,3,5,8};

arr = {1,2,3,3,5,5,6,6,8,8,9,9};
```

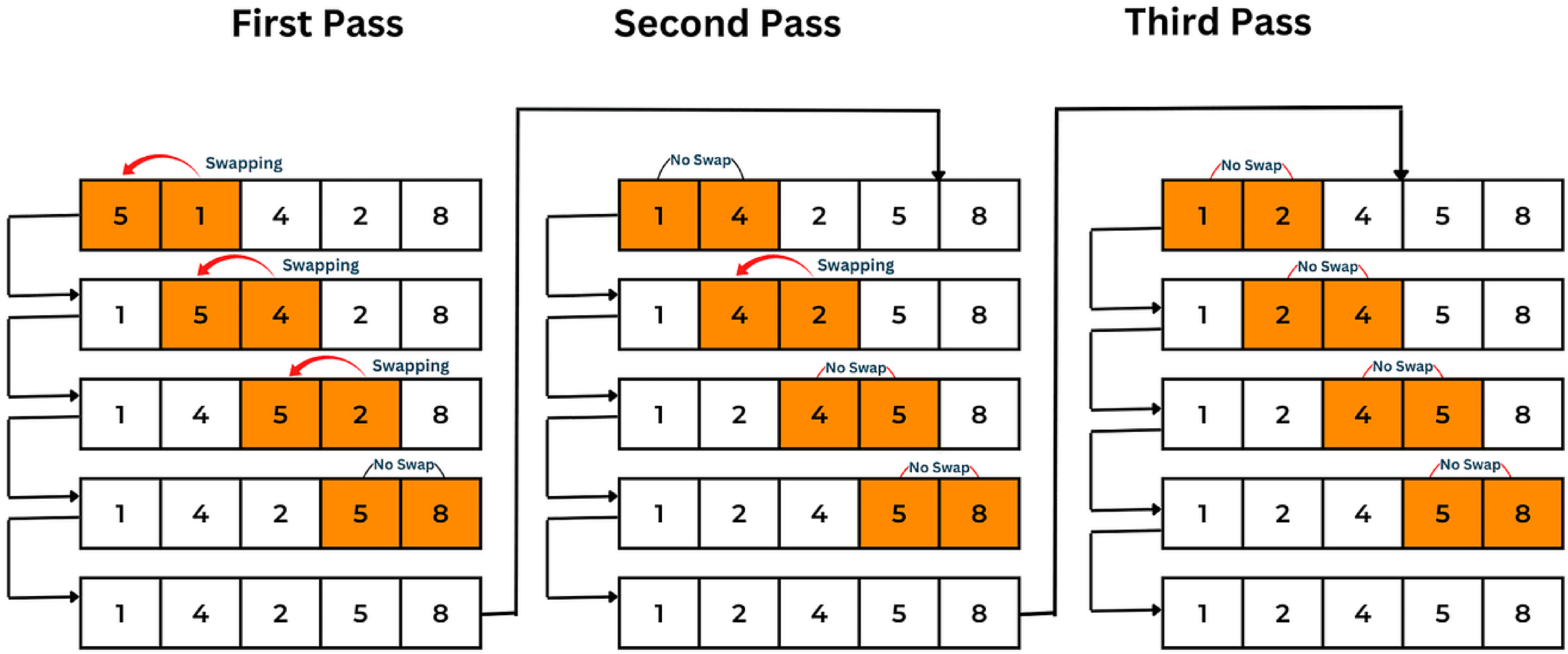
Stable : Sorting

Not stable : sorting

Bubble Sort:



BUBBLE SORTING



Algorithm 1: Bubble sort

Data: Input array *A[]*

Result: Sorted *A[]*

```
int i, j, k;
N = length(A);
for j = 1 to N do
    for i = 0 to N-1 do
        if A[i] > A[i+1] then
            temp = A[i];
            A[i] = A[i+1];
            A[i+1] = temp;
        end
    end
end
```

```java
class Bsort{
    void bsort(int arr[])
    {
        int n = arr.length;
        for(int i =0 ;i<n-1;i++){
            for(int j=0;j<n-i-1;j++){
                if(arr[j] > arr[j+1])
                {
                    int temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
                }
            }


        }
    }

    void display(int arr[]){
        int n = arr.length;
        for(int i=0;i<n;i++){
```

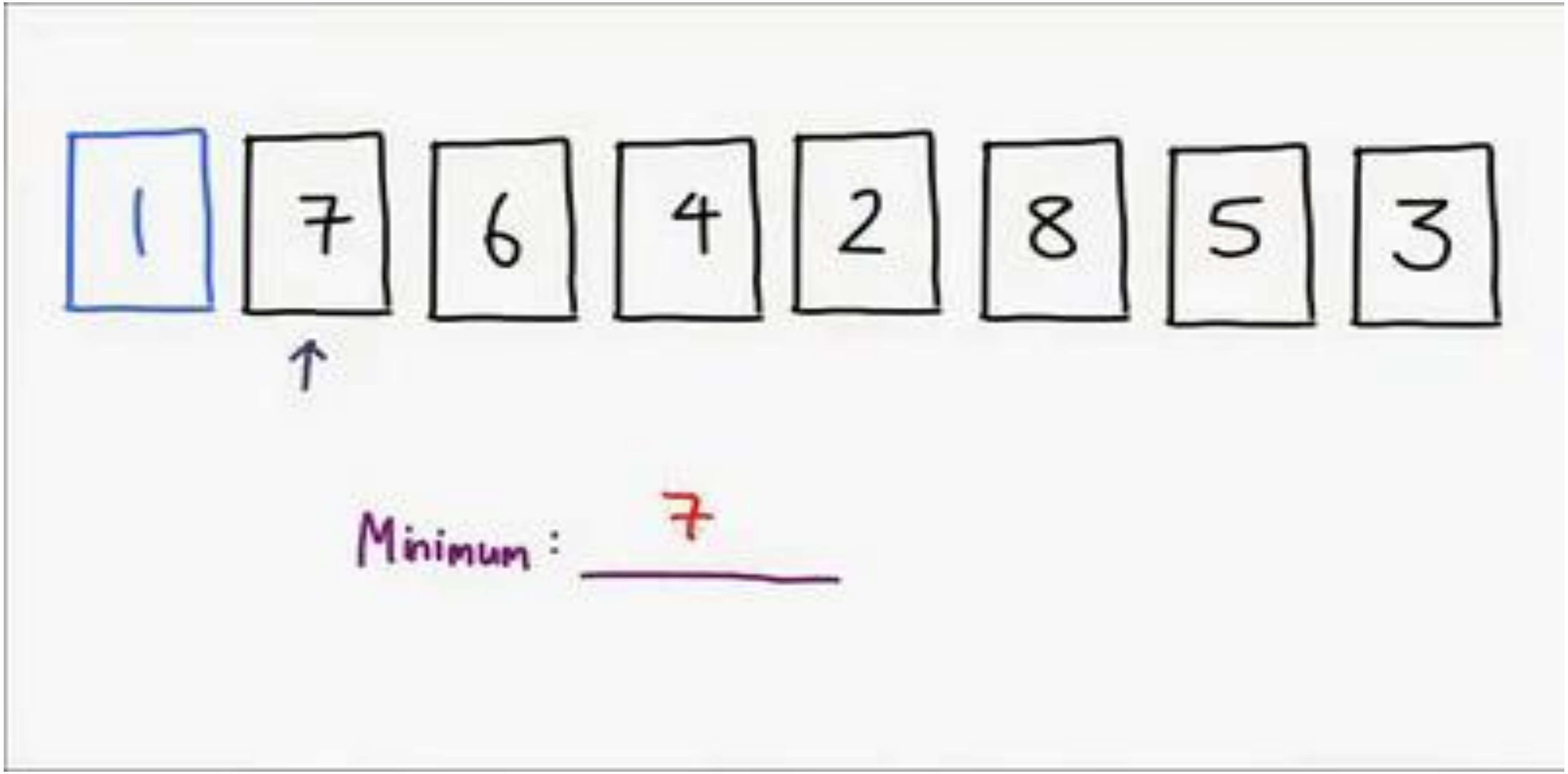
length : 638   lines : 47   Ln : 17   Col : 13   Pos : 257   Windows

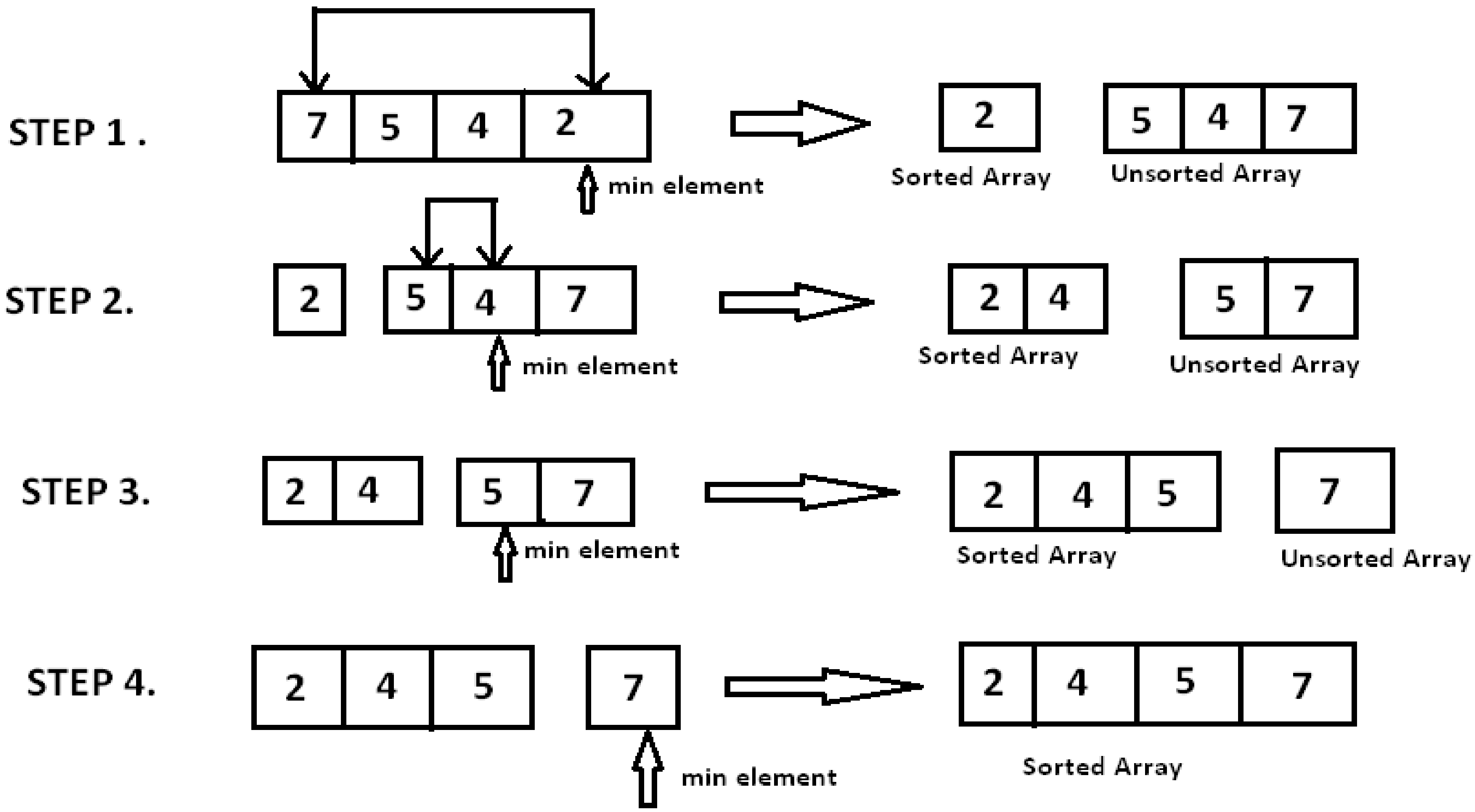
Best case: 11,22,33,44,...99

Average case:
Worst case
O(n^2)

No of comparisions: n-1

Space complexity: O(n)


1
7
6
4
2
8
5
3
Minimum : 7

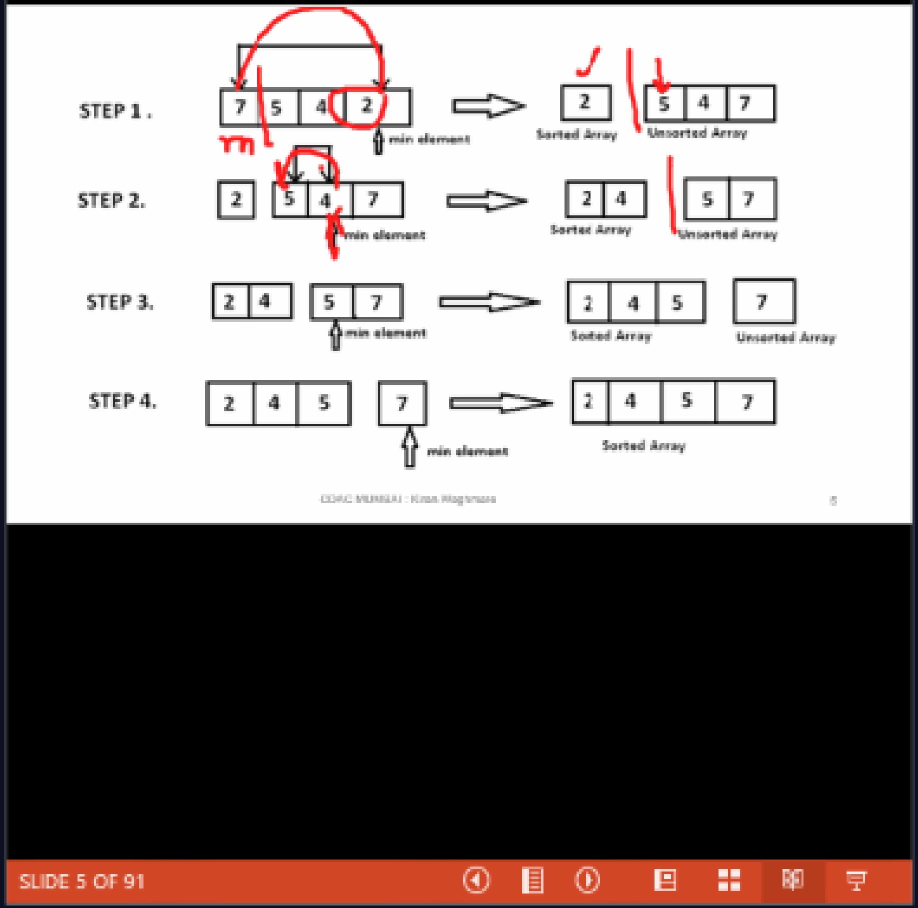

STEP 1 .
7 5 4 2
min element
2
Sorted Array
5 4 7
Unsorted Array
STEP 2.
2
5 4 7
min element
2 4
Sorted Array
5 7
Unsorted Array
STEP 3.
2 4
5 7
min element
2 4 5
Sorted Array
7
Unsorted Array
STEP 4.
2 4 5
7
min element
2 4 5 7
Sorted Array

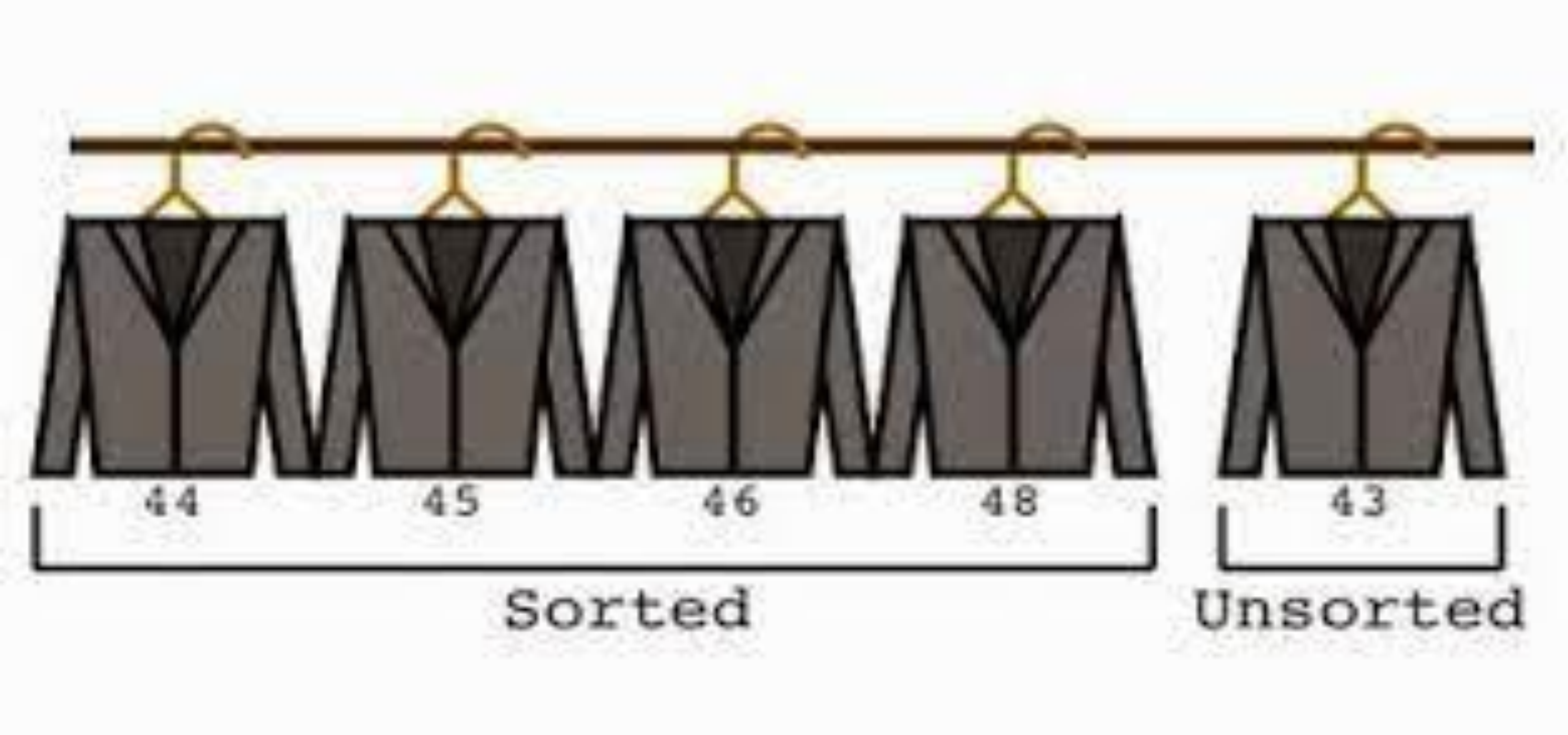
Algorithm:

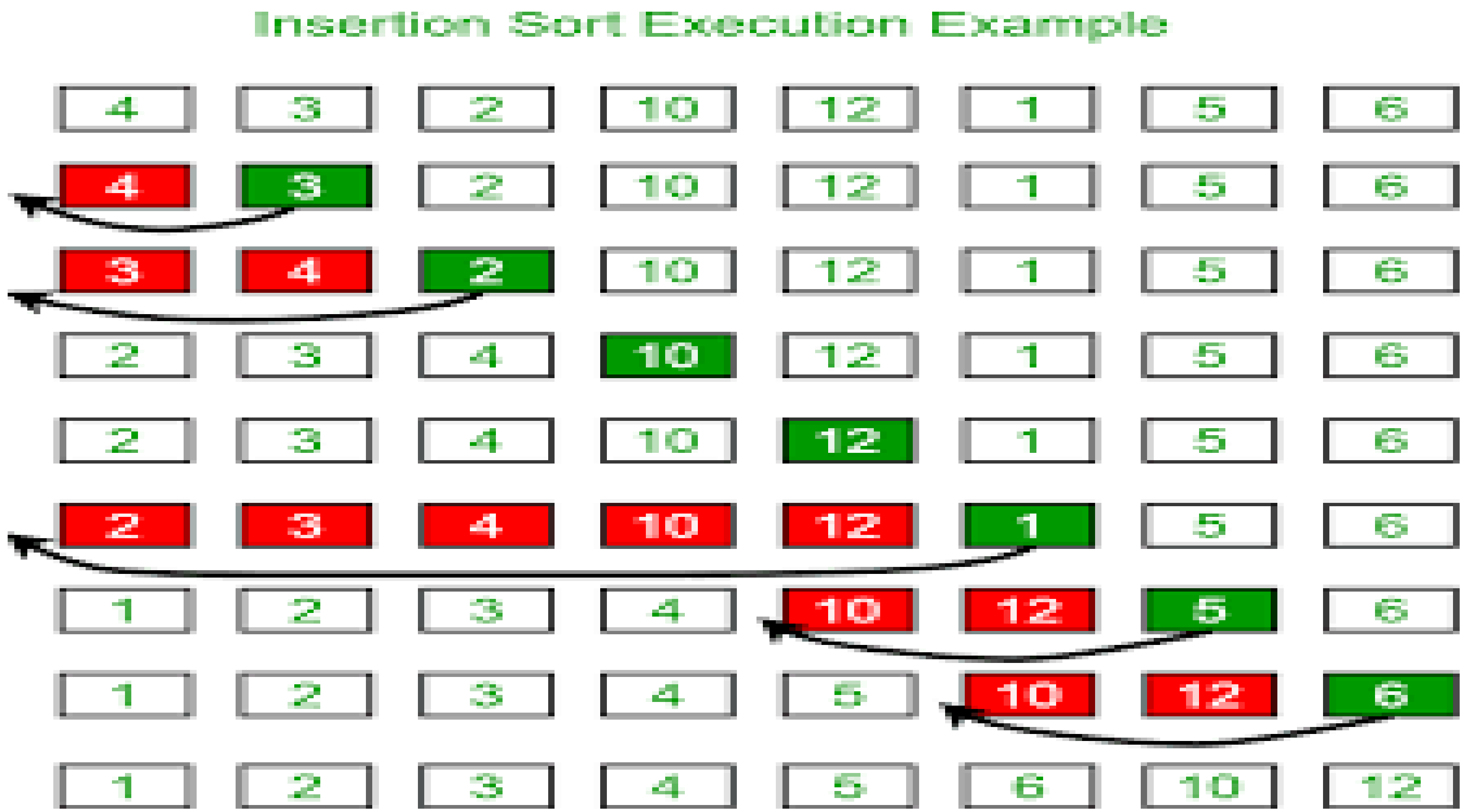
```
SelectionSort(A)
{
        for( i = 0;i < n ;i++)
        {
                least=A[i];
                p=i;
                for ( j = i + 1;j < n ;j++)
                {
                        if (A[j] < A[i])
                        least= A[j]; p=j;
                }
        }
        swap(A[i],A[p]);
}
```

Pooja Nalawade raised hand View

```java
}

void ssort(int arr[]){
    int n = arr.length;
    for(int i=0;i<n-1;i++){
        int min = i;
        for(int j=i+1;j<n;j++){
            if(arr[j] < arr[min])
                min = j;

        }
        int temp = arr[min];
        arr[min] = arr[i];
        arr[i] = temp;
    }
}

void display(int arr[]){
    int n = arr.length;
    for(int i=0;i<n;i++){
        System.out.print(arr[i]+" ");
    }
```




44
45
46
48
43
Sorted
Unsorted

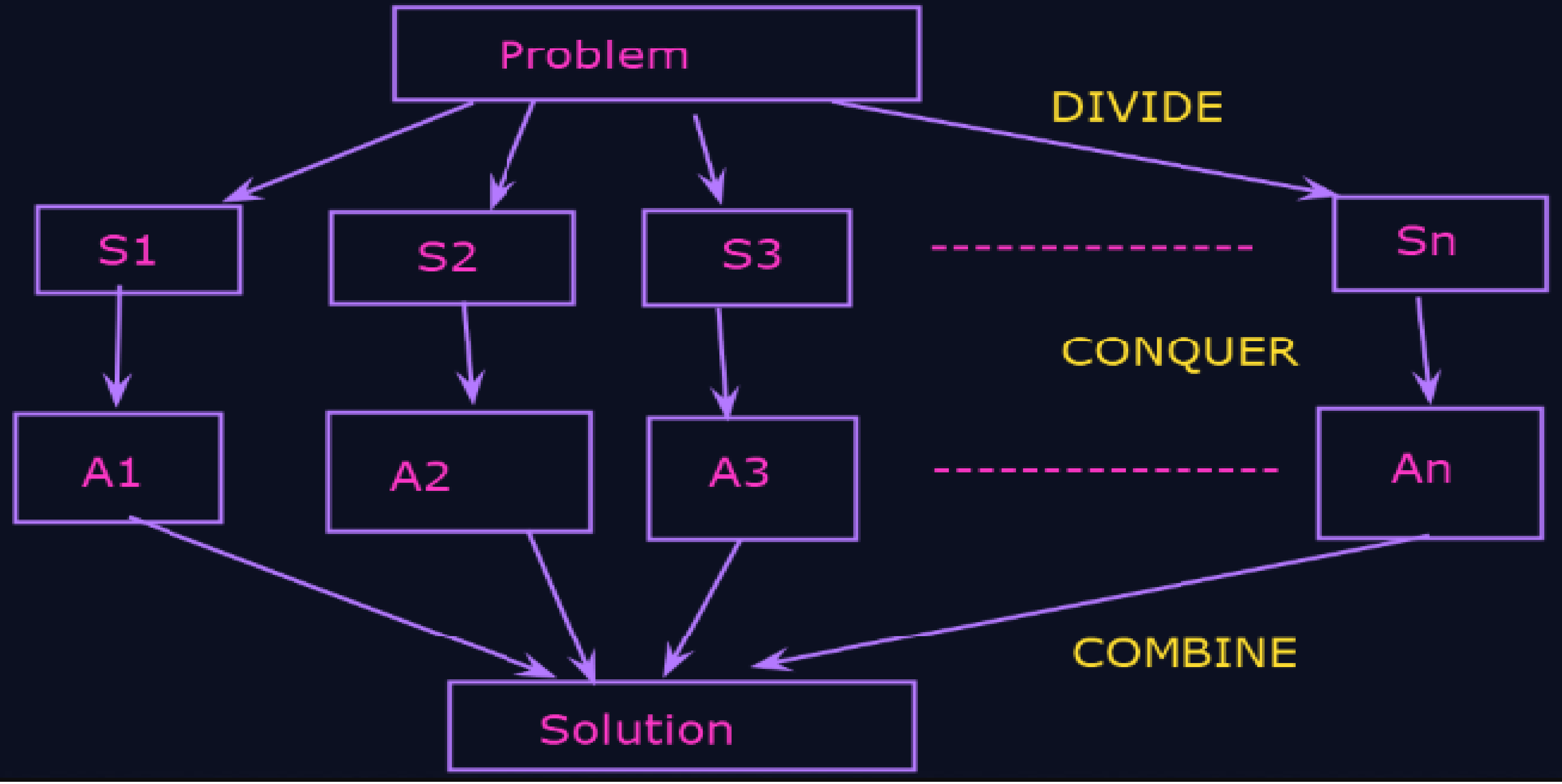

Insertion Sort Execution Example
4 3 2 10 12 1 5 6
4 3 2 10 12 1 5 6
3 4 2 10 12 1 5 6
2 3 4 10 12 1 5 6
2 3 4 10 12 1 5 6
2 3 4 10 12 1 5 6
1 2 3 4 10 12 5 6
1 2 3 4 5 10 12 6
1 2 3 4 5 6 10 12

INSERTION-SORT(*A*)	cost	times
for j ← 2 to n	c_1	n
do key ← A[j]	c_2	$n-1$
▷Insert A[j] into the sorted sequence A[1 . . j -1]	0	$n-1$
i ← j - 1	c_4	$n-1$
while i > 0 and A[i] > key	c_5	$\sum_{j=2}^{n} t_j$
do A[i + 1] ← A[i]	c_6	$\sum_{j=2}^{n} (t_j - 1)$
i ← i – 1	c_7	$\sum_{j=2}^{n} (t_j - 1)$
A[i + 1] ← key	c_8	$n-1$

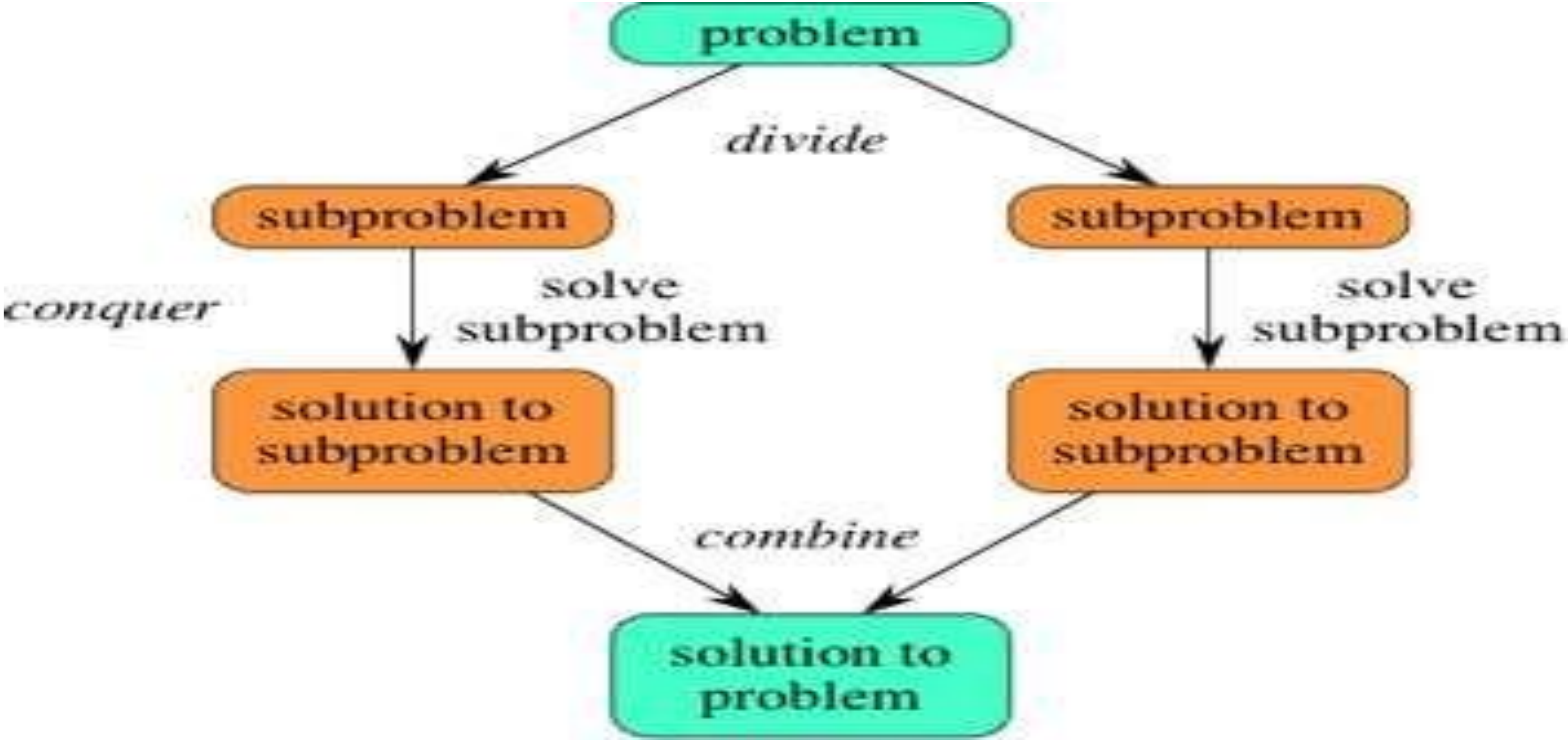
t_j: # of times the while statement is executed at iteration j

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Divide and conquer:



Divide-and-conquer



Merge Sort:

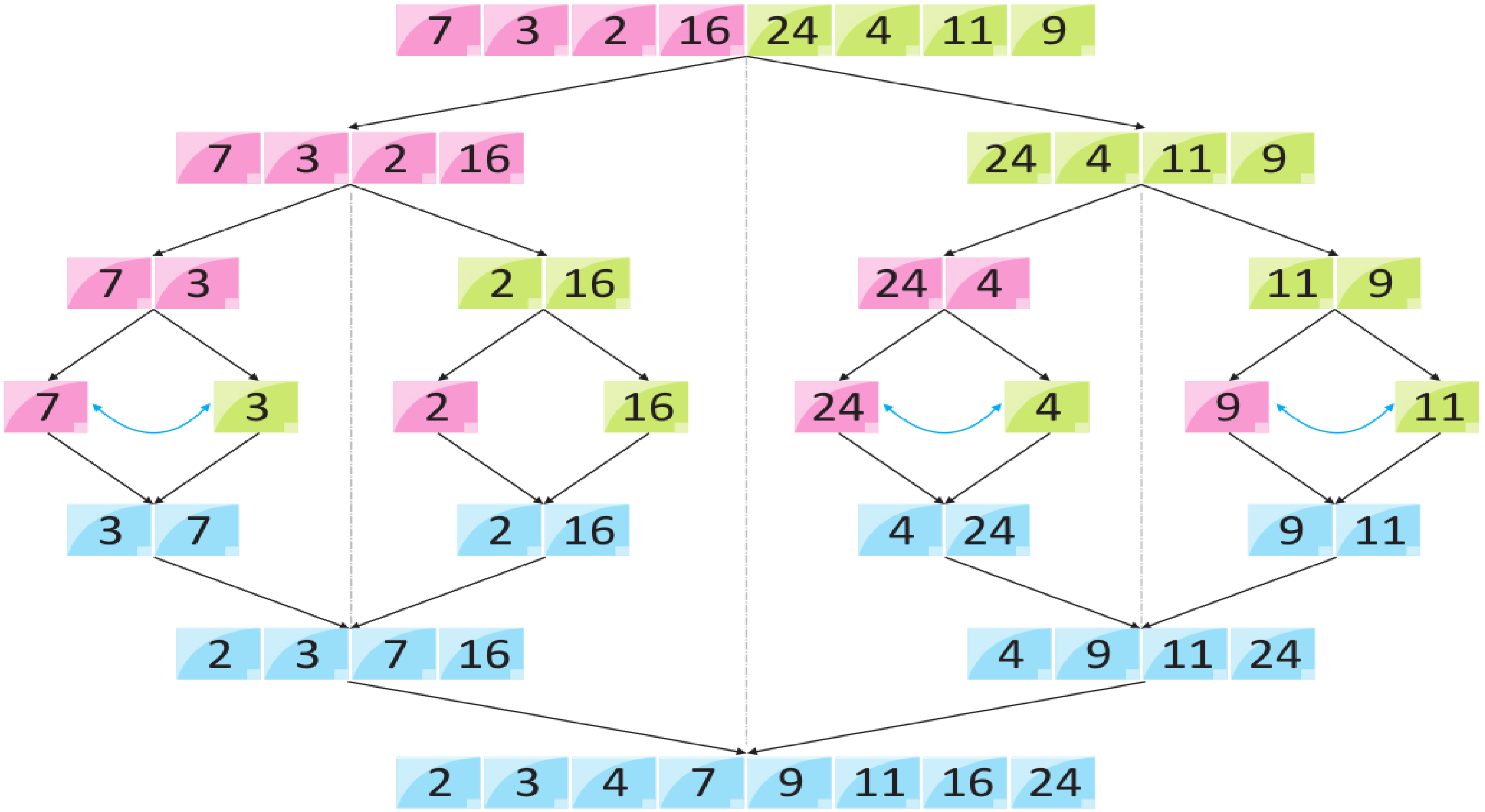
$\boxed{3\ 1\ 6\ 8\ 4\ 5\ 7\ 2}$

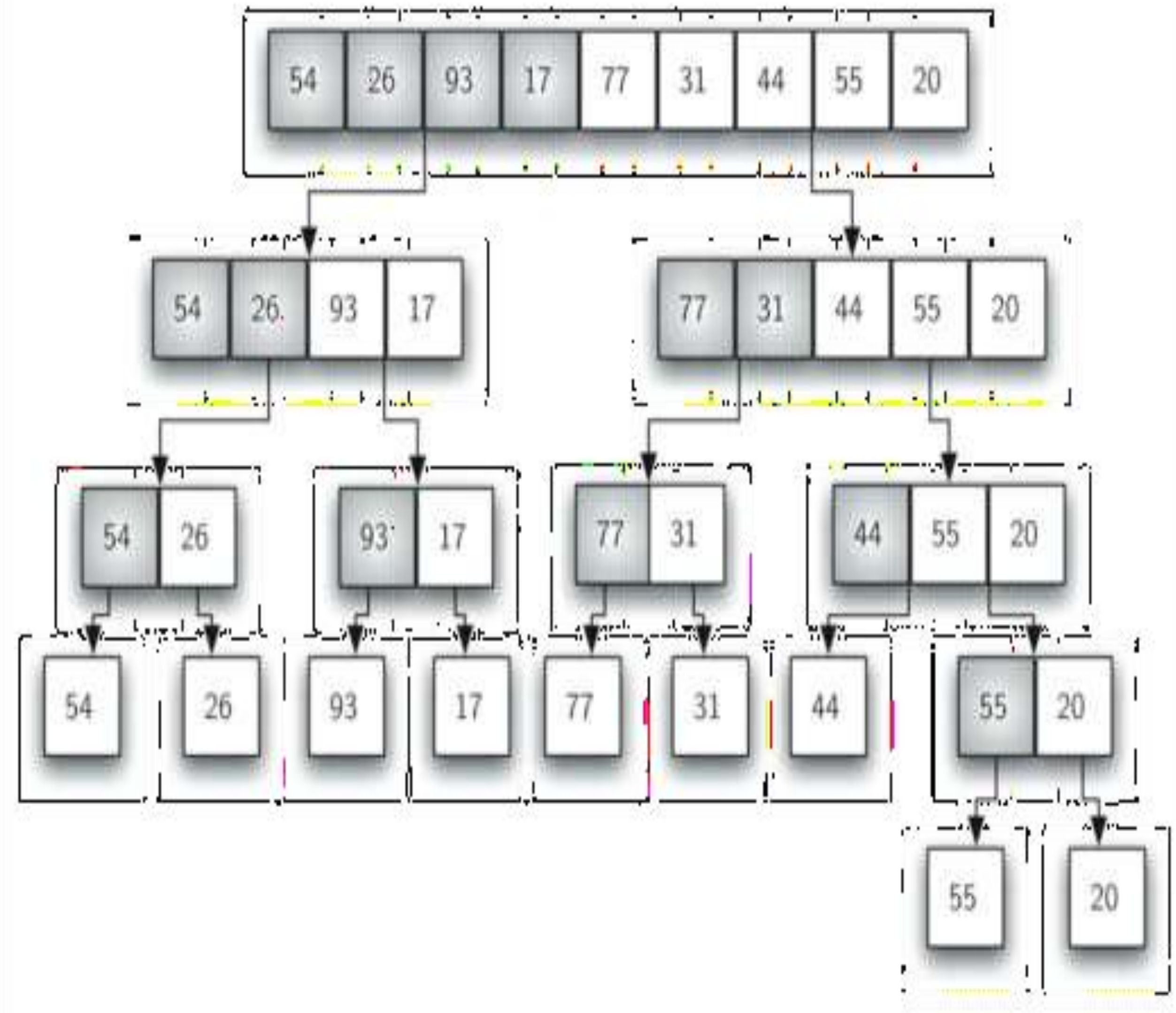
method call

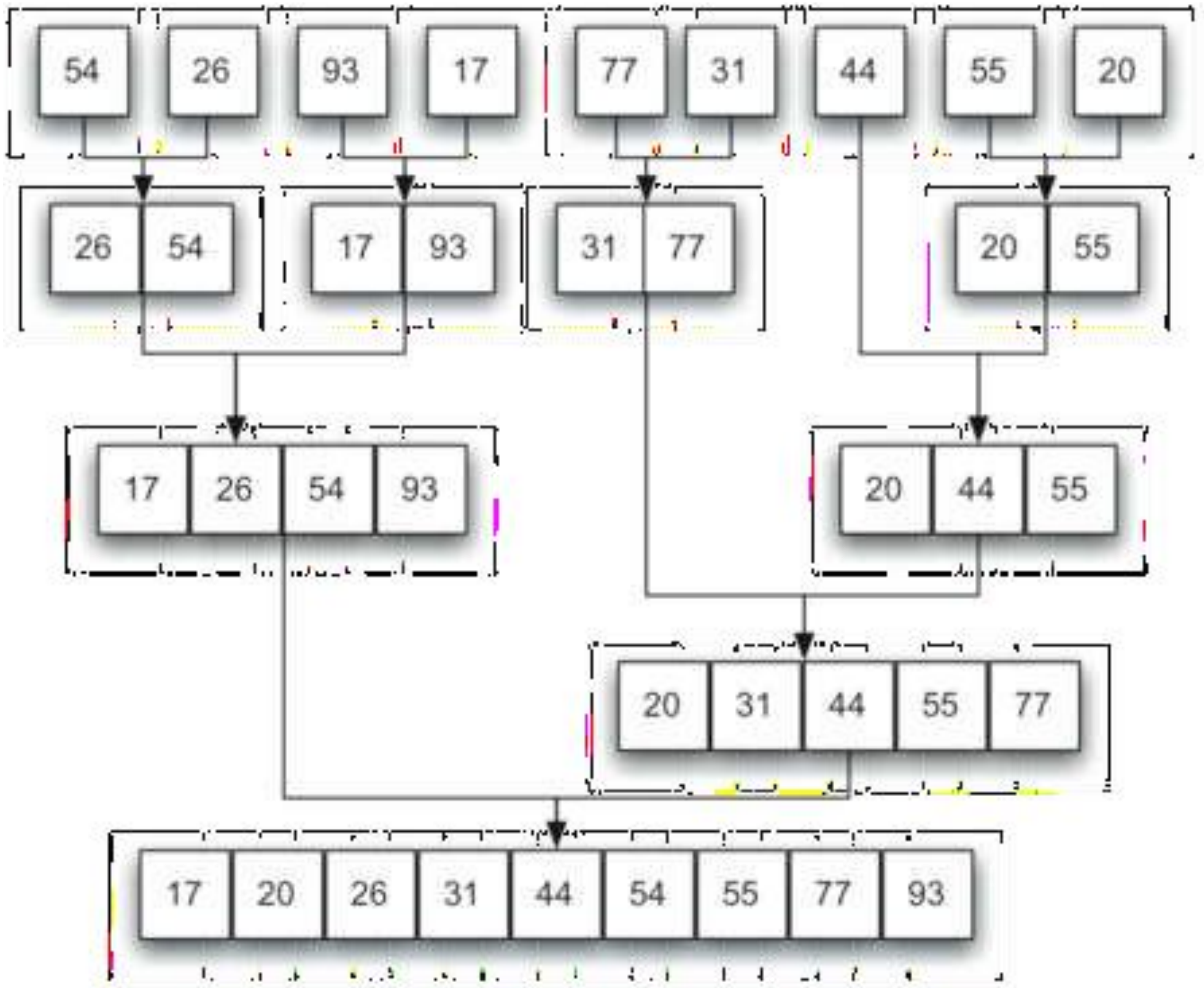
return value

merge

$\Downarrow$


7 3 2 16 24 4 11 9
7 3 2 16
24 4 11 9
7 3
2 16
24 4
11 9
7
3
2
16
24
4
9
11
3 7
2 16
4 24
9 11
2 3 7 16
4 9 11 24
2 3 4 7 9 11 16 24


54 26 93 17 77 31 44 55 20
54 26 93 17
77 31 44 55 20
54 26
93 17
77 31
44 55 20
54
26
93
17
77
31
44
55 20
55
20


54 26 93 17 77 31 44 55 20
26 54
17 93
31 77
20 55
17 26 54 93
20 44 55
20 31 44 55 77
17 20 26 31 44 54 55 77 93

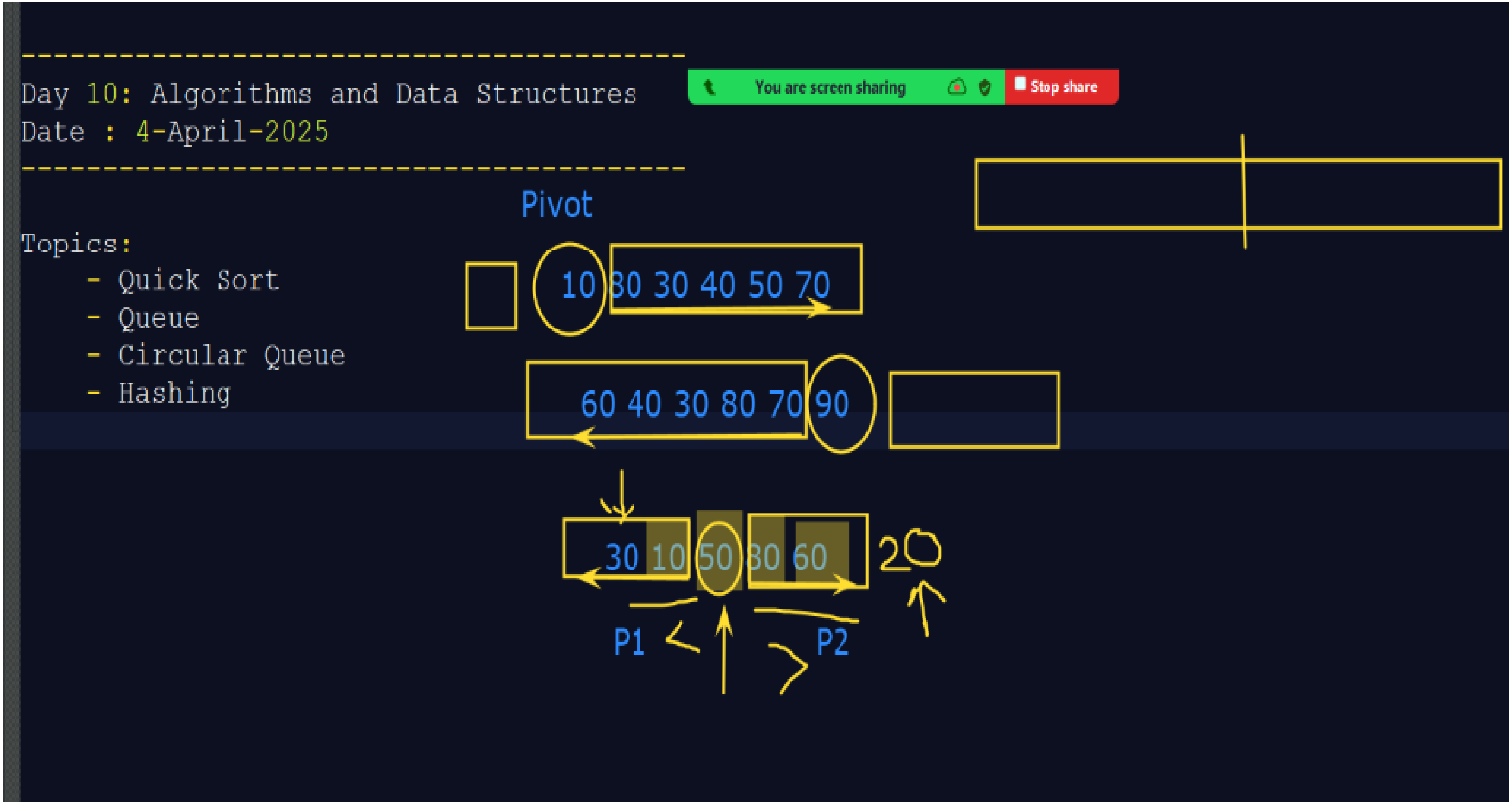
Merge Sort:

Here is the pseudocode for Merge Sort, modified to include a counter:

```
count ← 0
Merge_Sort(A, p, r)
        if p < r
                then   q ← ⌊(p + r)/2⌋
                       Merge-Sort (A, p, q)
                       Merge-Sort (A, q+1, r)
                       Merge (A, p, q, r)
```

And here is the modified algorithm for the Merge function used by Merge Sort:

```
Merge (A, p, q, r)
        n1 ← (q – p) + 1
        n2 ← (r – q)
        create arrays L[1..n1+1] and R[1..n2+1]
        for i ← 1 to n1 do
                L[i] ← A[(p + i) -1]
        for j ← 1 to n2 do
                R[j] ← A[q + j]
        L[n1 + 1] ← ∞
        R[n2 + 1] ← ∞
        i ← 1
        j ← 1
        for k ← p to r do
                count ← count + 1
                if L[I] <= R[j]
                        then  A[k] ← L[i]
                              i ←i + 1
                        else A[k] ← R[j]
                              j ← j + 1
```


Day 10: Algorithms and Data Structures
Date : 4-April-2025
You are screen sharing
Stop share
Pivot
Topics:
- Quick Sort
- Queue
- Circular Queue
- Hashing
10 80 30 40 50 70
60 40 30 80 70 90
30 10 50 80 60
20
P1
P2

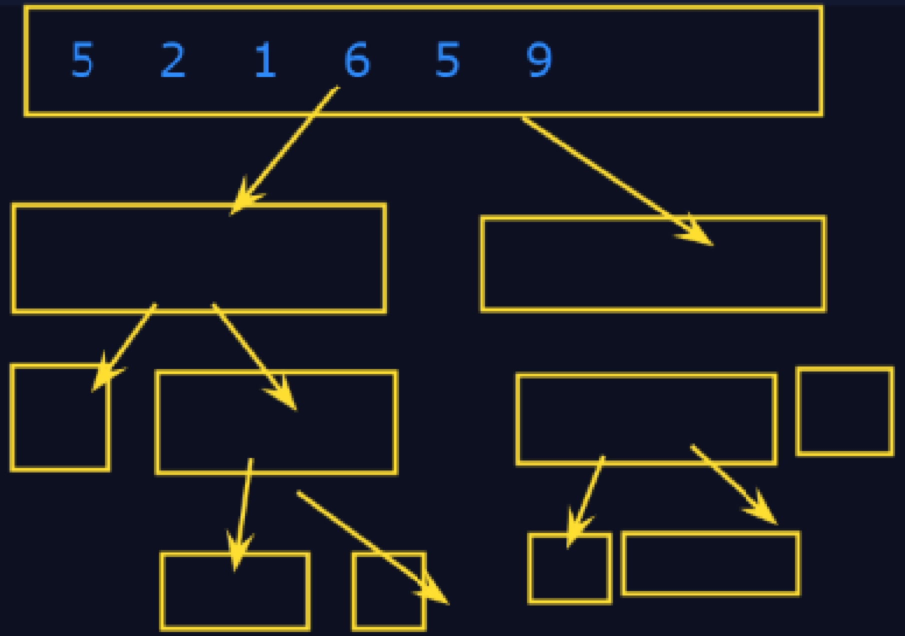

Unsorted Array
35
33
42
10
14
19
27
44
26
31

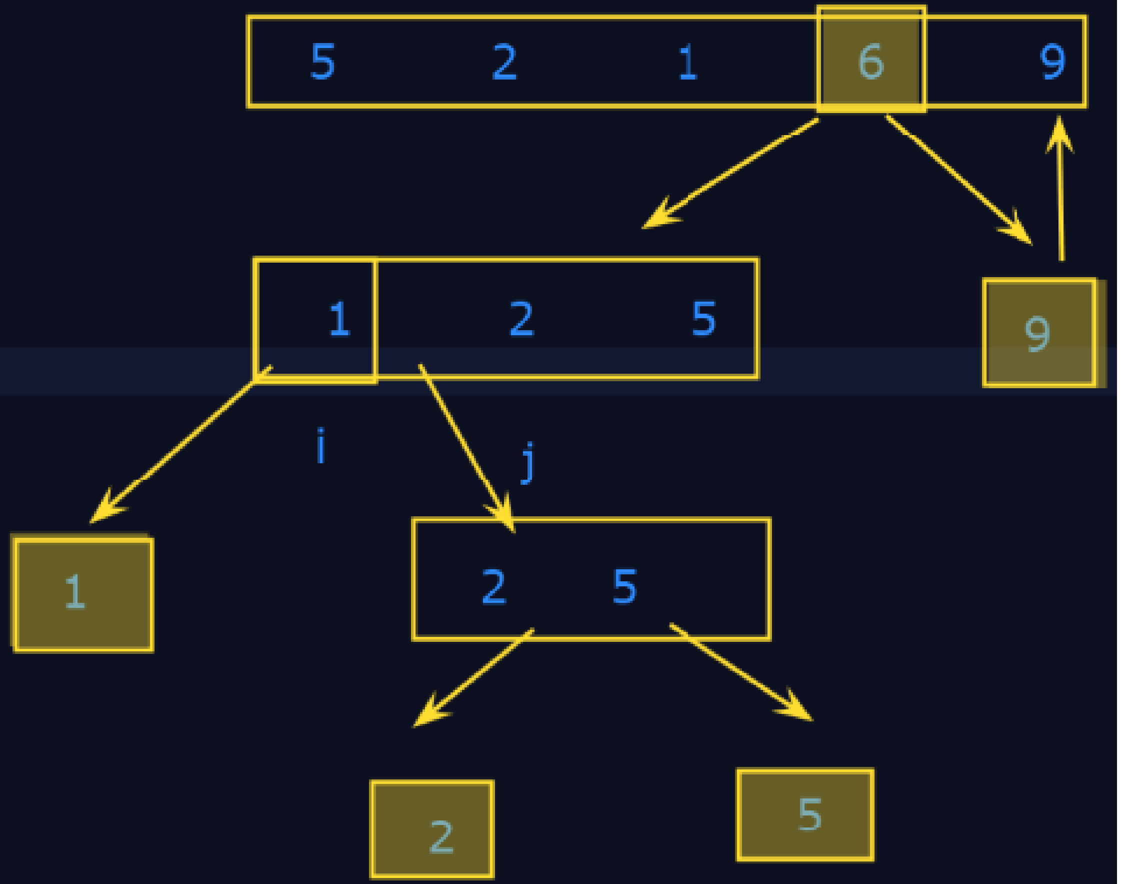
011_Amar Khare_KH raised hand View

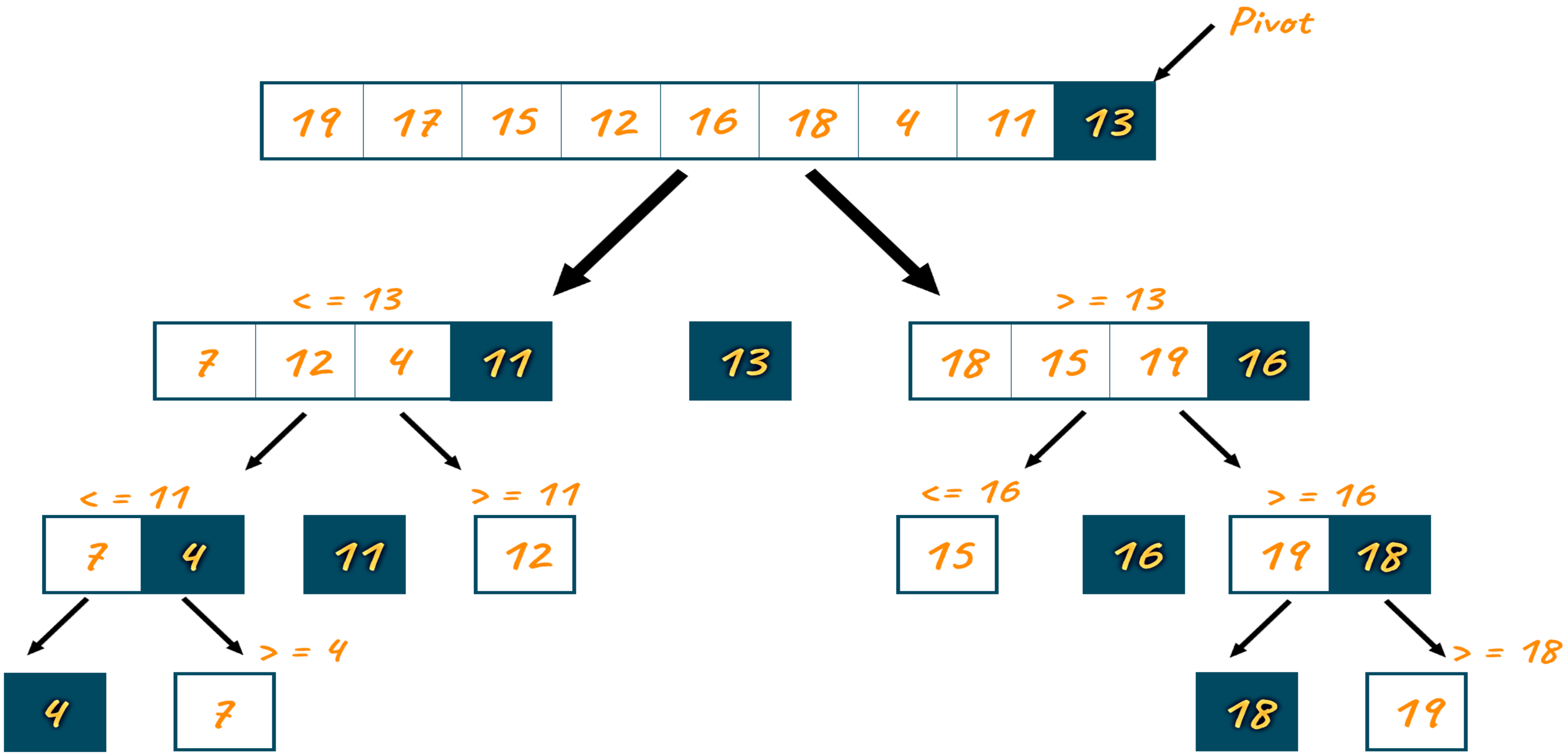
You are screen sharing Stop share

Day 10: Algorithms and Data Structures
Date : 4-April-2025

Topics:
- Quick Sort
- Queue
- Circular Queue
- Hashing







Pivot
19 17 15 12 16 18 4 11 13
< = 13
7 12 4 11
13
> = 13
18 15 19 16
< = 11
7 4
11
> = 11
12
<= 16
15
16
> = 16
19 18
4
> = 4
7
18
> = 18
19

The following procedure implements quicksort:

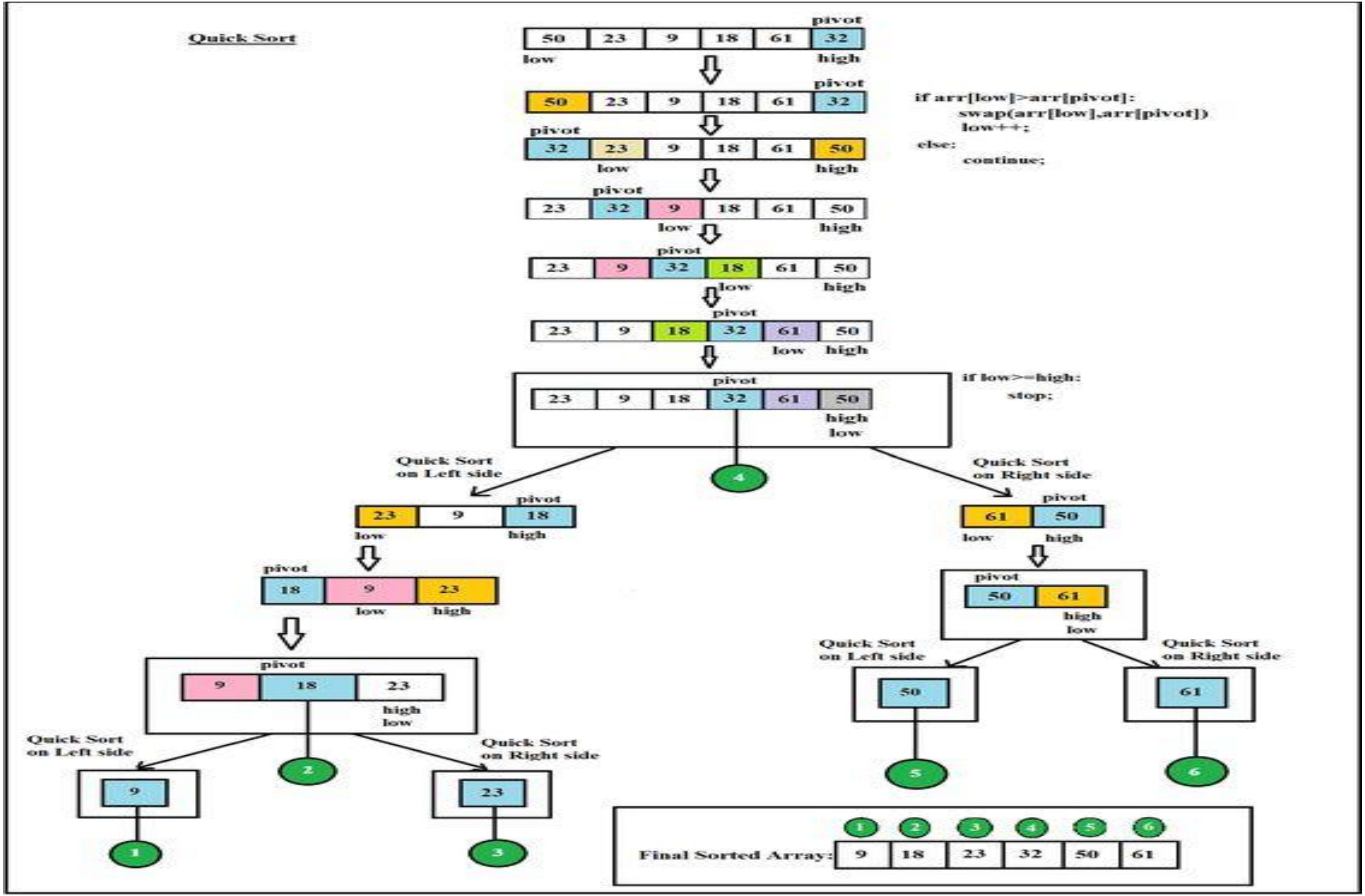
```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

To sort an entire array A, the initial call is QUICKSORT(A, 1, $A.length$).

Partitioning the array

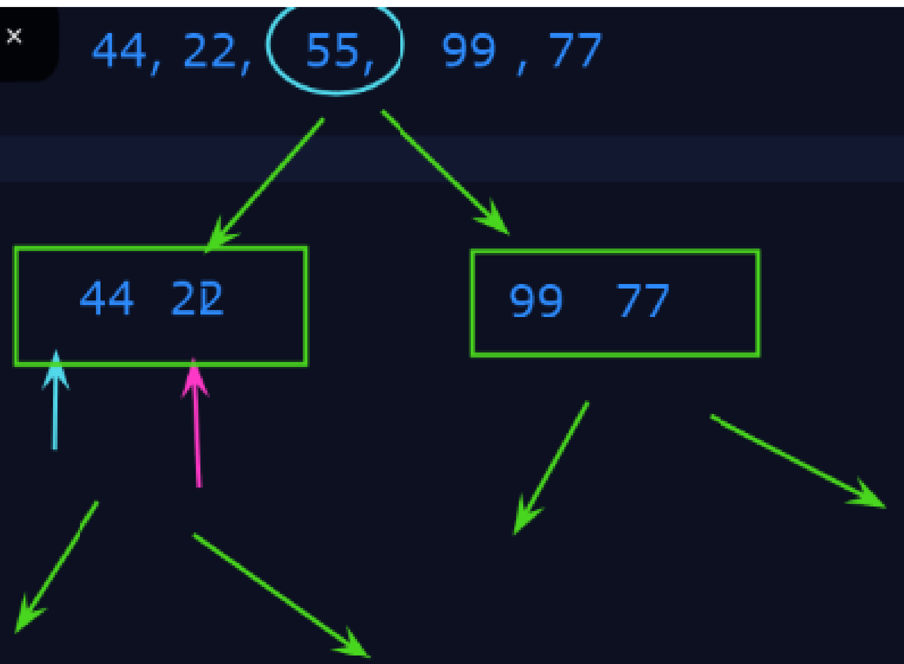
The key to the algorithm is the PARTITION procedure, which rearranges the subarray $A[p \,..\, r]$ in place.

```
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
```

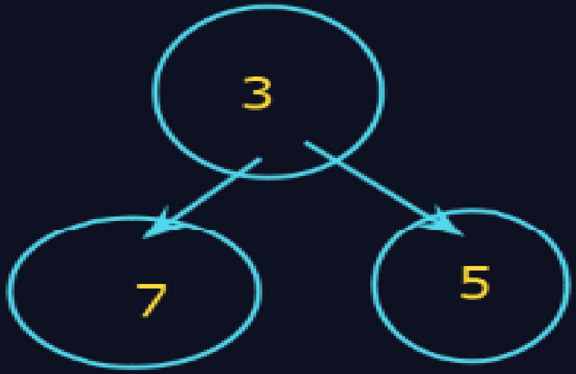

Quick Sort
pivot
50 23 9 18 61 32
low high
if arr[low]>arr[pivot]:
swap(arr[low],arr[pivot])
low++;
else:
continue;
32 23 9 18 61 50
23 32 9 18 61 50
23 9 32 18 61 50
23 9 18 32 61 50
if low>=high:
stop;
Quick Sort on Left side
Quick Sort on Right side
23 9 18
18 9 23
9 18 23
61 50
50 61
Final Sorted Array: 9 18 23 32 50 61

```java
static void quicksort(int arr[], int low, int high){

    if(low < high)
    {
        int pi = partition(arr, low, high);
        quicksort(arr, low, pi-1);//Left array : P1
        quicksort(arr, pi+1, high);//Right array : P2

    }
}

static int partition(int arr[], int low, int high){
    int pivot = arr[high];
    int i = low-1;
    for(int j=low;j<=high-1;j++){
        if(arr[j] < pivot)
        {
            i++;
            swap(arr,i,j);
        }

    }
    swap(arr, i+1, high);
    return(i+1);
}
```

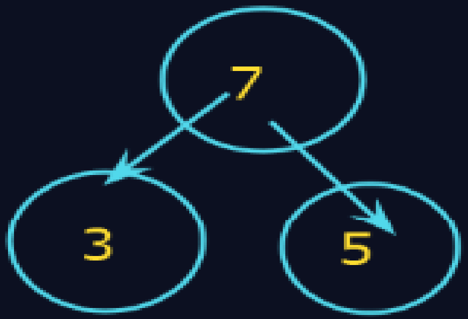
Pooja Nalawade raised hand View



HEap Sort:



Min Heap



Max Heap

Heap:
-A special form of complete binary tree such that the key value of each is either smaller or greater than the root node.

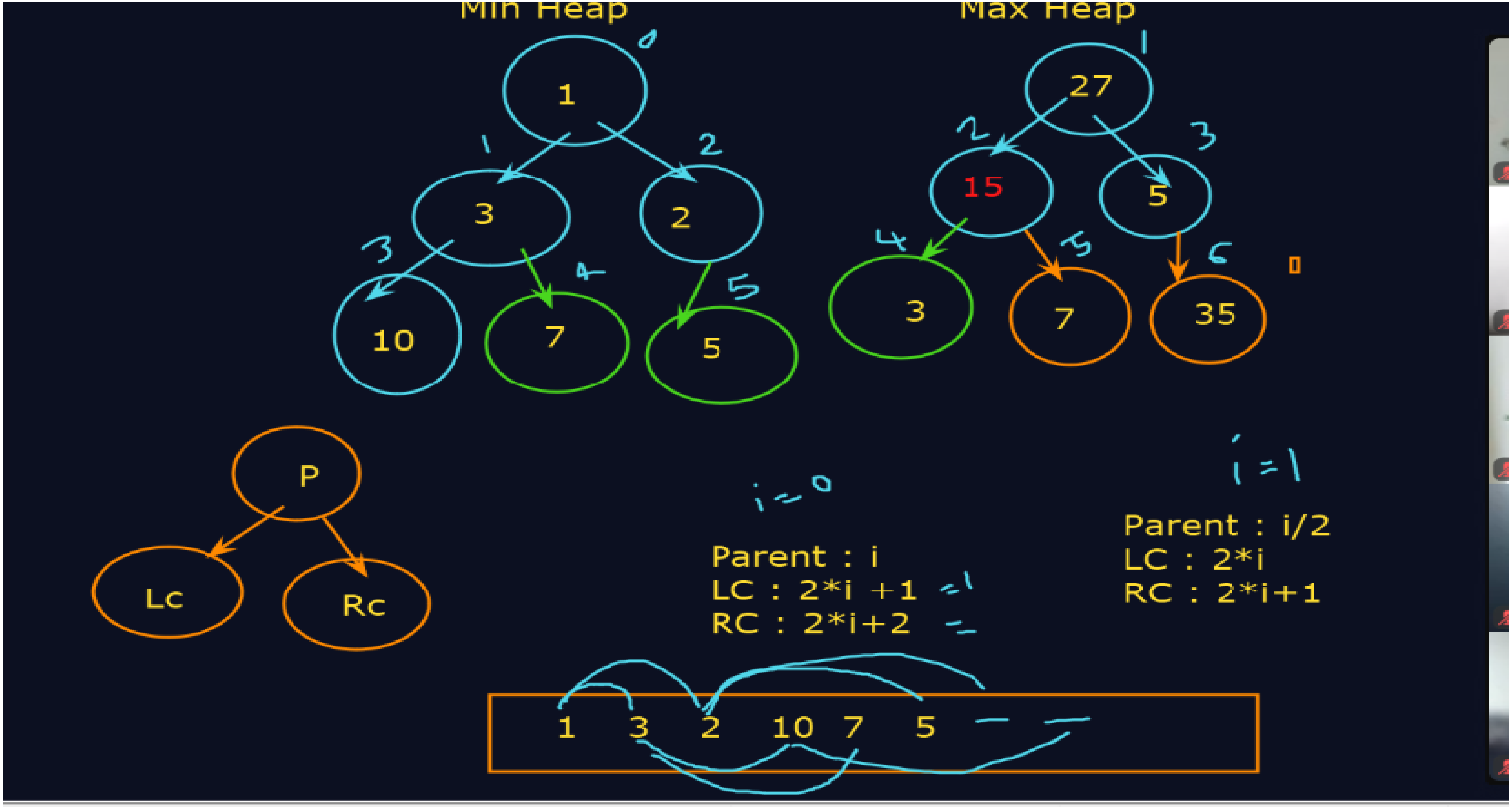
Heap

Types of heap:
----
1. Max heap:

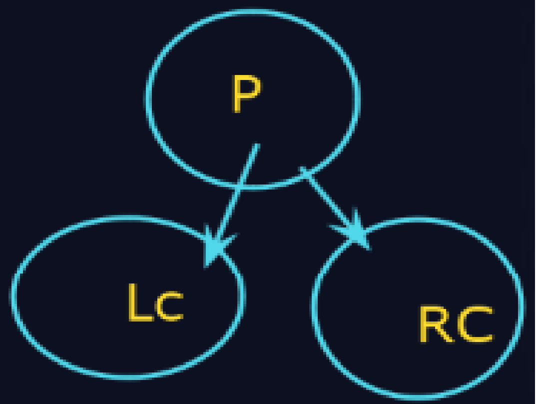
-A max heap in which the key value of the root node is greater than the othe

2. Min heap:

-A max heap in which the key value of the root node is smaller than the othe


Min Heap
Max Heap
1
3
2
10
7
5
27
15
5
3
7
35
P
Lc
Rc
i=0
Parent : i
LC : 2*i +1
RC : 2*i+2
i=1
Parent : i/2
LC : 2*i
RC : 2*i+1
1 3 2 10 7 5

```
class Hsort{

    void heapify(int arr[],int n, int i){

        int largest = i;//Parent
        int l = 2*i+1; //LC
        int r = 2*i+2; //RC

        if(l < n && arr[l] > arr[largest])
            largest = l;
        if(r < n && arr[r] > arr[largest])
            largest = r;

        if( largest != i){
            int temp = arr[i];
            arr[i] = arr[largest];
            arr[largest] = temp;

            heapify(arr, n, largest);
        }
```

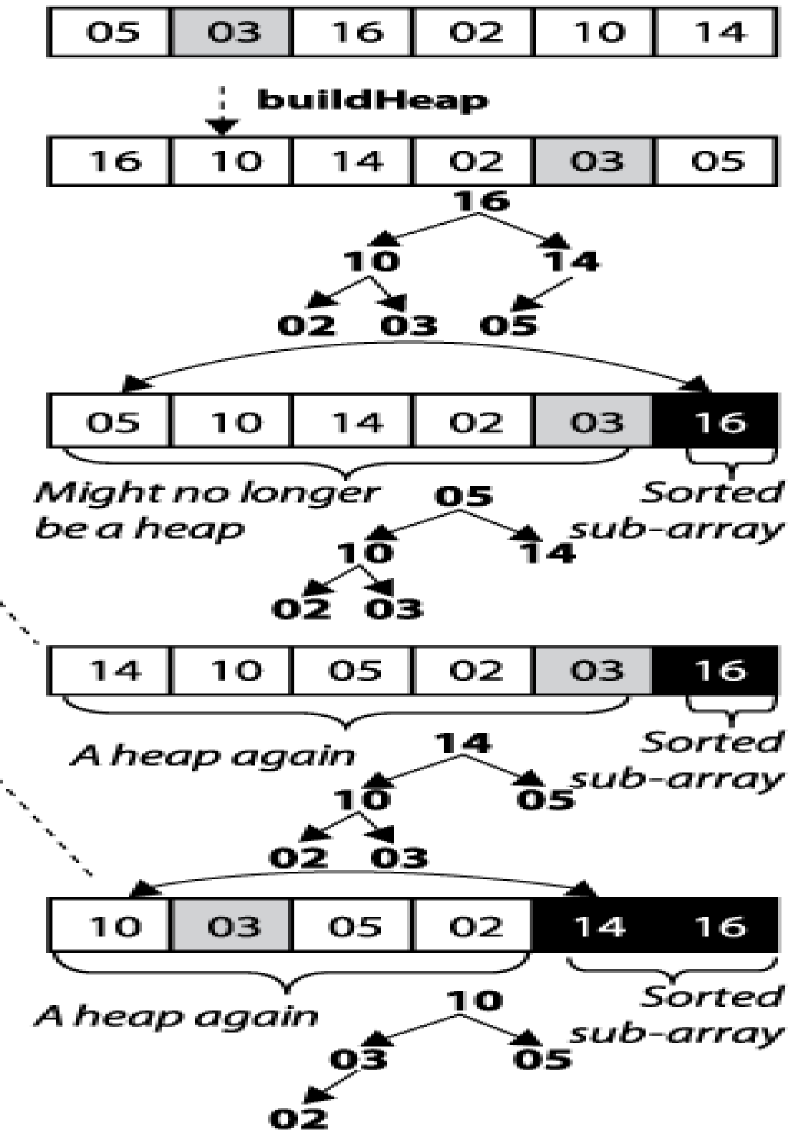


Heap Sort

Best	Average	Worst
O(n log n)	O(n log n)	O(n log n)

Array Recursion Binary Heap

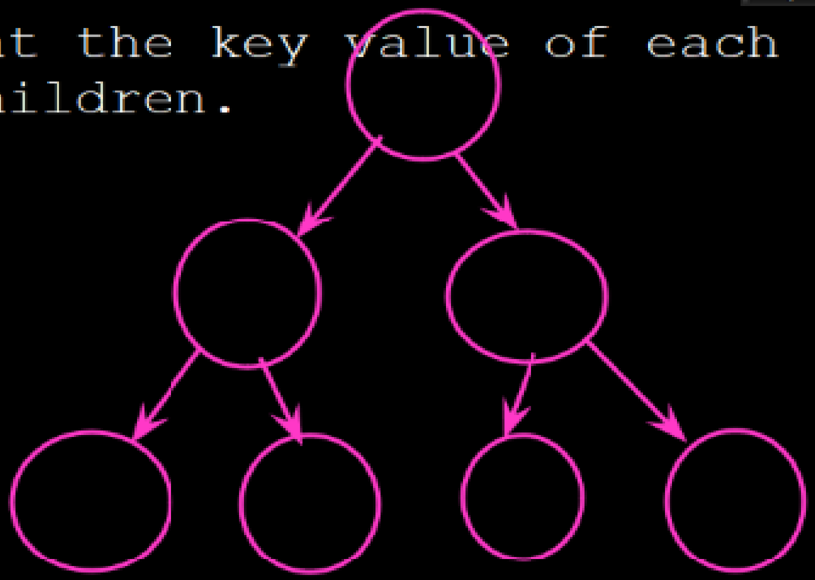
```
sort (A)
1.     buildHeap(A)
2.     for i = n − 1  downto 1 do
3.        swap A[0] with A[i]
4.        heapify (A, 0, i)
end

buildHeap (A)
1.     for i = ⌊n/2⌋ − 1 downto 0 do
2.        heapify (A, i, n)
end

heapify (A, idx, max)
1.     left = 2*idx + 1
2.     right = 2*idx + 2
3.     if (left < max and A[left] > A[idx]) then
4.        largest = left
5.     else largest = idx
6.     if (right < max and A[right] > A[largest]) then
7.        largest = right
8.     if (largest ≠ idx) then
9.        swap A[i] and A[largest]
10.       heapify (A, largest, max)
end
```



Heap

DEfinition:

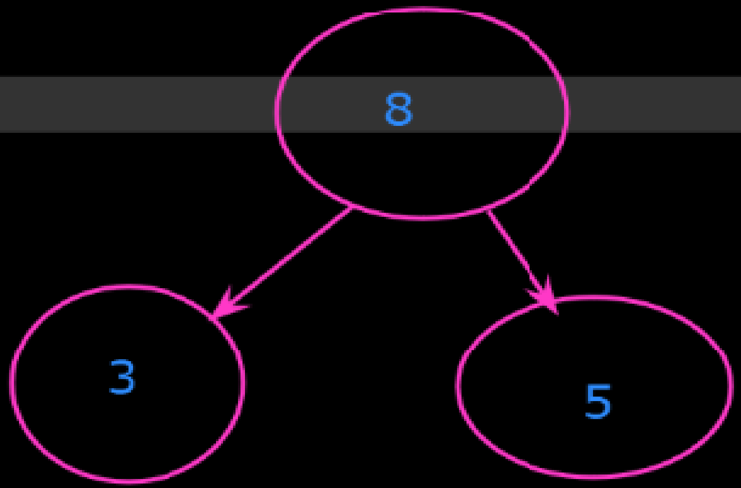
-a special form of complete binary tree that the key value of each no
smaller(larger) than the key value of it children.



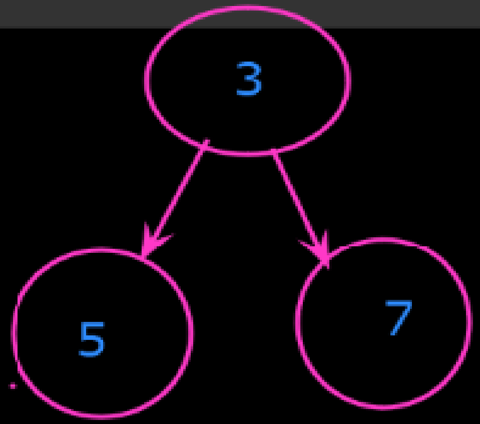
Types of heap:

1. Max-Heap: root node has largest value

2. Min-Heap: root node has smallest value



Max Heap



Min Heap

Heap

Heap

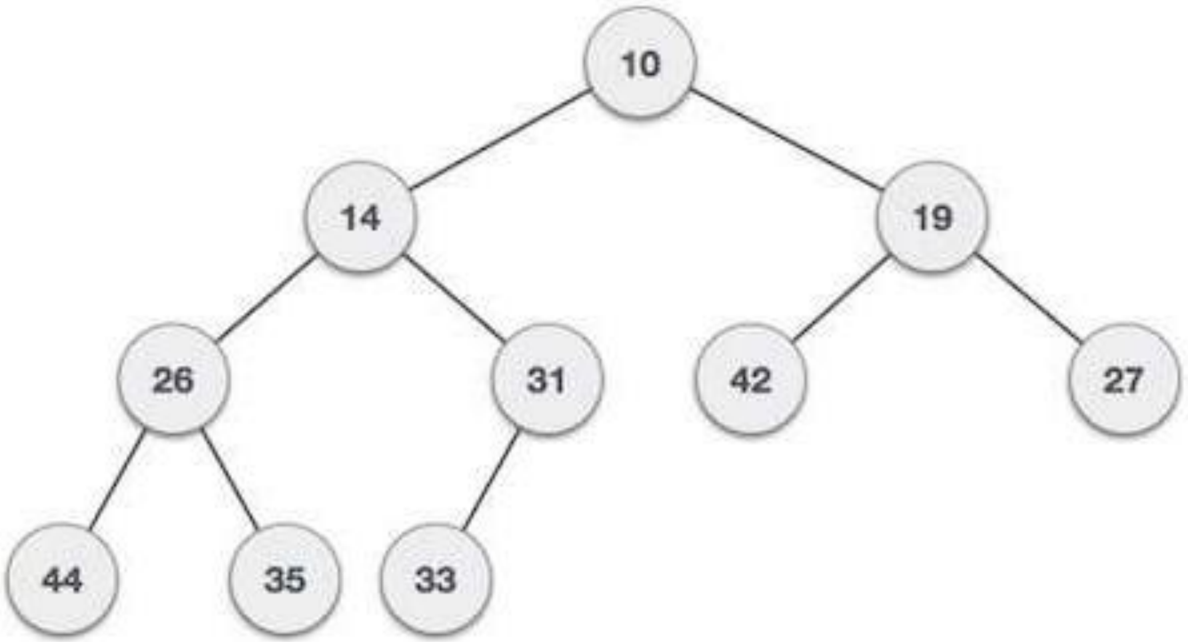
- **Definition in Data Structure**
  - **Heap:** A special form of complete binary tree that key value of each node is no smaller (larger) than the key value of its children (if any).

- **Max-Heap: root node has the largest key.**
  - A *max tree* is a tree in which the key value in each node is no smaller than the key values in its children.
  - A *max heap* is a complete binary tree that is also a max tree.

- **Min-Heap: root node has the smallest key.**
  - A *min tree* is a tree in which the key value in each node is no larger than the key values in its children.
  - A *min heap* is a complete binary tree that is also a min tree.

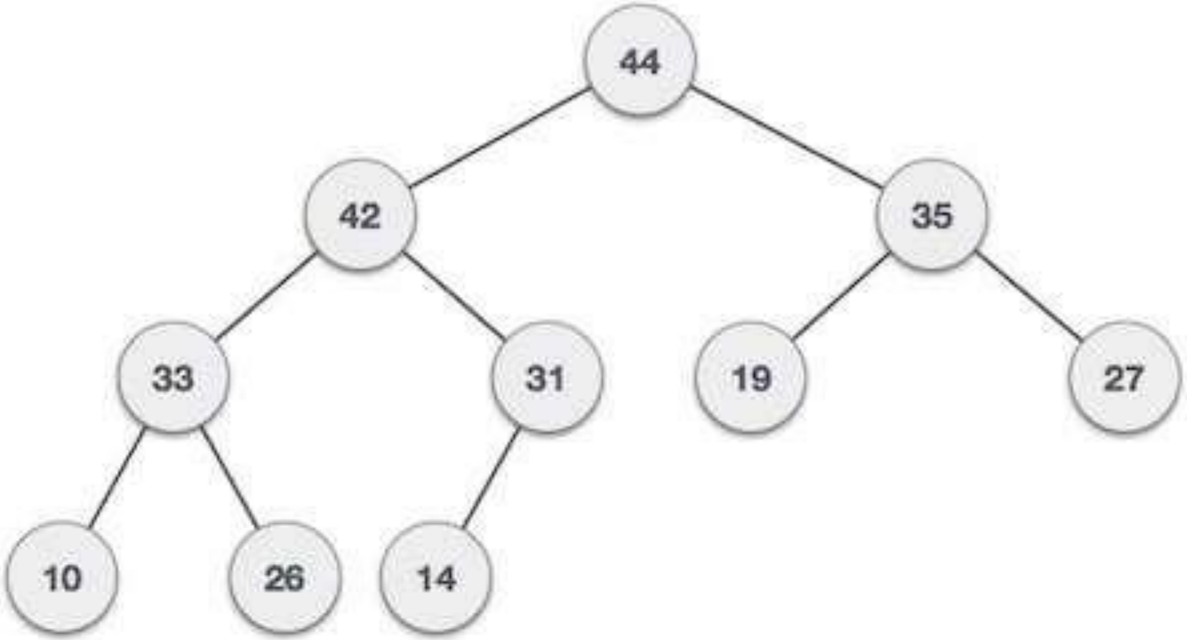
Heap

- Min-Heap
  - Where the value of the root node is less than or equal to either of its children
  - For input 35 33 42 10 14 19 27 44 26 31



Heap

- Max-Heap −
  - where the value of root node is greater than or equal to either of its children.
  - For input 35 33 42 10 14 19 27 44 26 31



Types of heap:

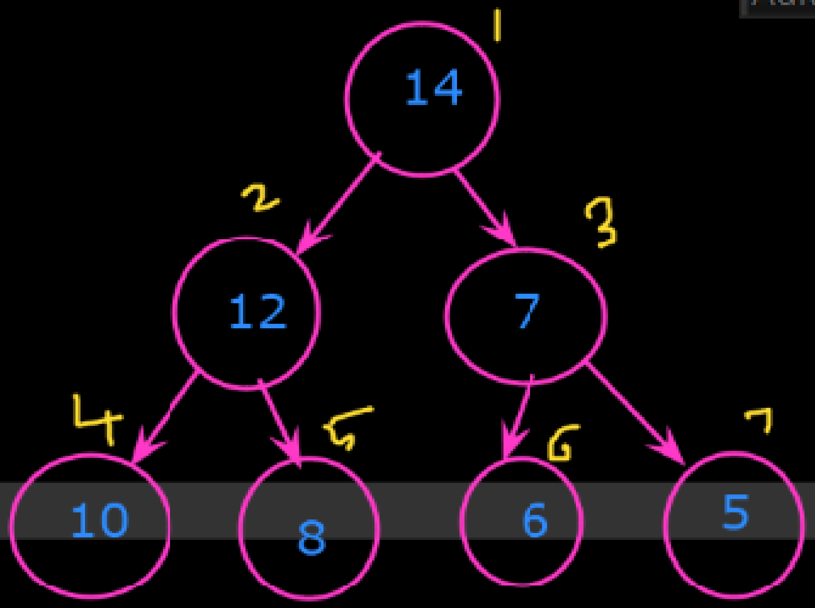
1. Max-Heap: root node has largest value

2. Min-Heap: root node has smallest value

Parent = i/2

Lc = 2i

RC = 2i+1



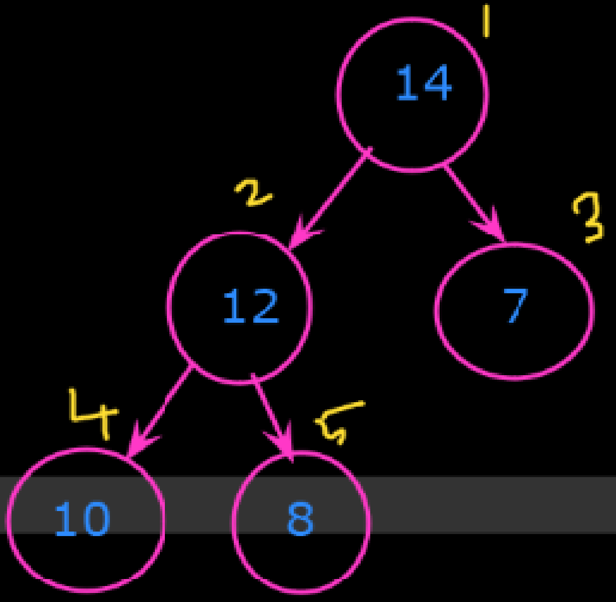
Max heap

14	12	7	10	8	6	5
1	2	3	4	5	6	7

Types of heap:

1. Max-Heap: root node has largest value

2. Min-Heap: root node has smallest value



Parent = i/2

Lc = 2i

RC = 2i+1

Max heap

14	12	7	10	8		
1	2	3	4	5	6	7

Types of heap:

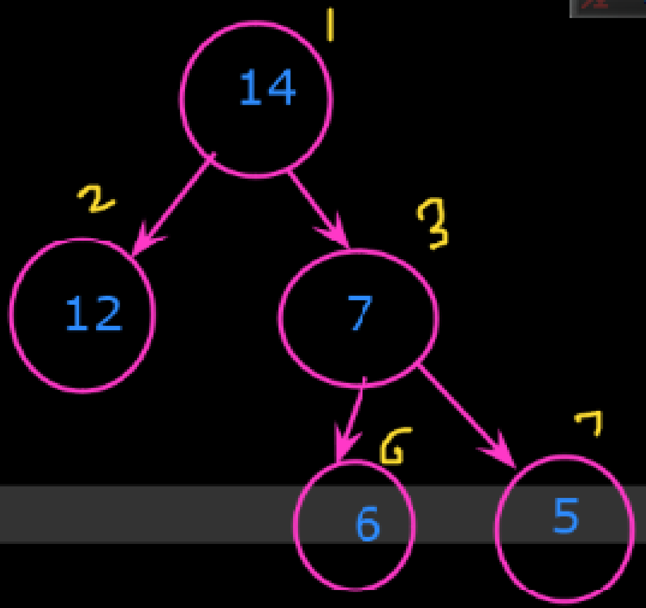
1. Max-Heap: root node has largest value

2. Min-Heap: root node has smallest value

Parent = i/2

Lc = 2i

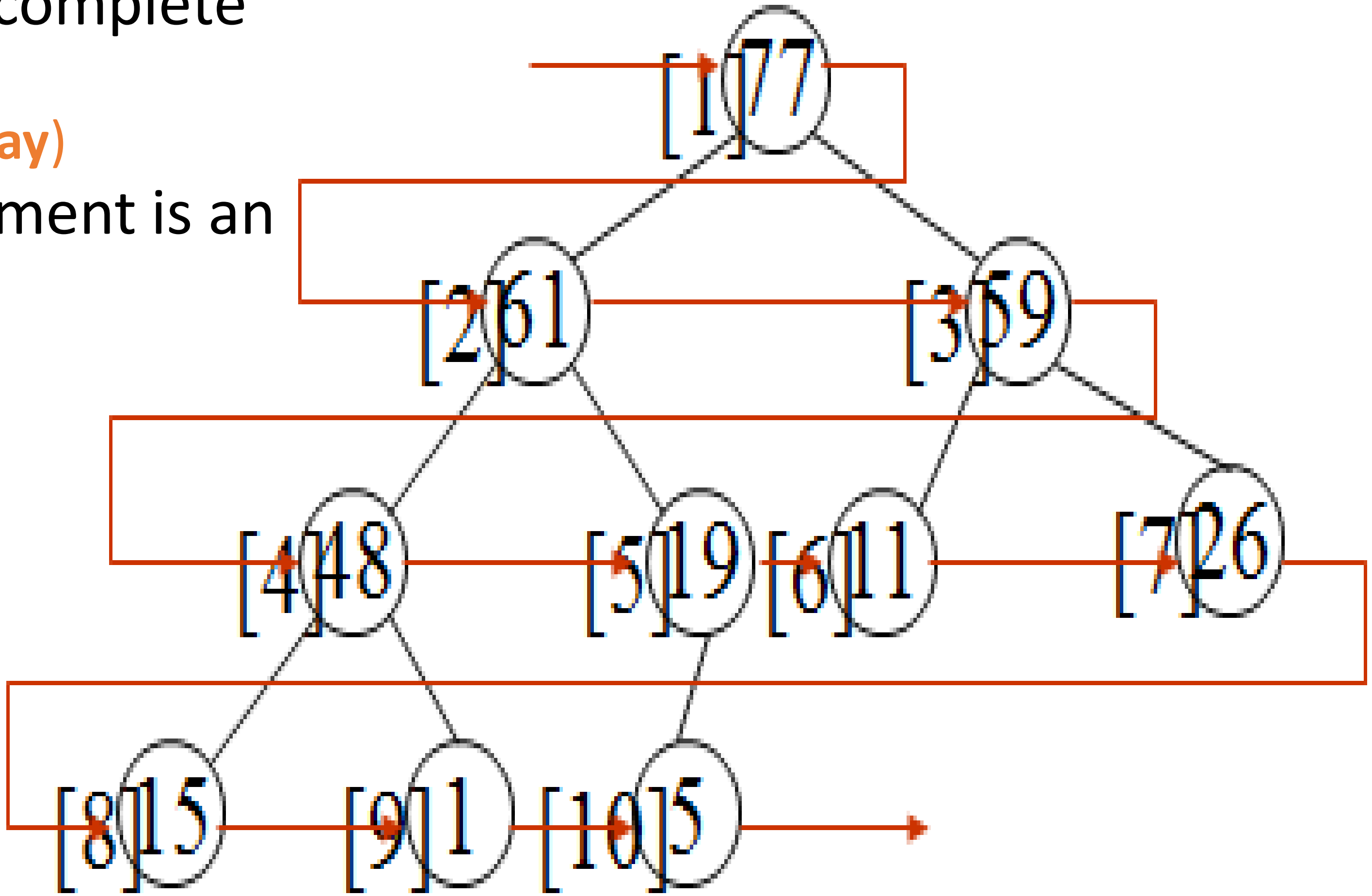
RC = 2i+1



Max heap

14	12	7	-	-	6	5
1	2	3	4	5	6	7

- **Note:**
  - Heap in data structure is a complete binary tree!
    - **(Nice representation in Array)**
  - Heap in C program environment is an array of memory.
-



— Stored using array in C

index	1	2	3	4	5	6	7	8	9	10
value	77	61	59	48	19	11	26	15	1	5

Example:- The fig. shows steps of heap-sort for list (2 3 7 1 8 5 6)