

# **Algorithms & Data Structure**

## **Searching & Sorting**

**Kiran Waghmare**

# Linear Search

---

- Find 37?

0	1	2	3	4	5	6	7	8
20	35	37	40	45	50	51	55	67



≠



≠



=

**Return 2**

# Linear Search

## Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that  $K \leq N$ . Following is the algorithm to find an element with a value of **ITEM** using sequential search.

1. Start
2. Set  $J = 0$
3. Repeat steps 4 and 5 while  $J < N$
4. IF  $LA[J]$  is equal ITEM THEN GOTO STEP 6
5. Set  $J = J + 1$
6. PRINT  $J$ , ITEM
7. Stop

# Program 3

**Problem:** Given an array `arr[]` of `n` elements, write a function to search a given element `x` in `arr[]`.

**Examples :**

**Input :** `arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170}`

`x = 110;`

**Output :** 6

Element `x` is present at index 6

**Input :** `arr[] = {10, 20, 80, 30, 60, 50,`

`110, 100, 130, 170}`

`x = 175;`

**Output :** -1

Element `x` is not present in `arr[]`.

# Binary Search

- Find 37?
  - Sort Array.

0	1	2	3	4	5	6	7	8
20	35	37	40	45	50	51	55	67

# Binary Search

2. Calculate  $\text{middle} = (\text{low} + \text{high}) / 2$ .  
 $= (0 + 8) / 2 = 4$ .

0	1	2	3	4	5	6	7	8
20	35	37	40	45	50	51	55	67
↑ first				↑ middle				↑ last

If  $37 == \text{array}[\text{middle}] \rightarrow \text{return middle}$

Else if  $37 < \text{array}[\text{middle}] \rightarrow \text{high} = \text{middle} - 1$

Else if  $37 > \text{array}[\text{middle}] \rightarrow \text{low} = \text{middle} + 1$

# Binary Search

Repeat 2. Calculate  $\text{middle} = (\text{low} + \text{high}) / 2$ .  
 $= (0 + 3) / 2 = 1$ .

0	1	2	3	4	5	6	7	8
20	35	37	40	45	50	51	55	67
↑	↑		↑					
first	middle		last					

If  $37 == \text{array}[\text{middle}] \rightarrow \text{return middle}$


Else if  $37 < \text{array}[\text{middle}] \rightarrow \text{high} = \text{middle} - 1$

Else if  $37 > \text{array}[\text{middle}] \rightarrow \text{low} = \text{middle} + 1$

# Binary Search

Repeat 2. Calculate  $\text{middle} = (\text{low} + \text{high}) / 2$ .  
 $= (2 + 3) / 2 = 2$ .

0	1	2	3	4	5	6	7	8
20	35	37	40	45	50	51	55	67

  
middle first last

If  $37 == \text{array}[\text{middle}] \rightarrow \text{return middle}$

Else if  $37 < \text{array}[\text{middle}] \rightarrow \text{high} = \text{middle} - 1$

Else if  $37 > \text{array}[\text{middle}] \rightarrow \text{low} = \text{middle} + 1$



# Home Work

**Move all negative numbers to beginning and positive to end with constant extra space**

**An array contains both positive and negative numbers in random order. Rearrange the array elements so that all negative numbers appear before all positive numbers.**

**Examples :**

**Input: -12, 11, -13, -5, 6, -7, 5, -3, -6**

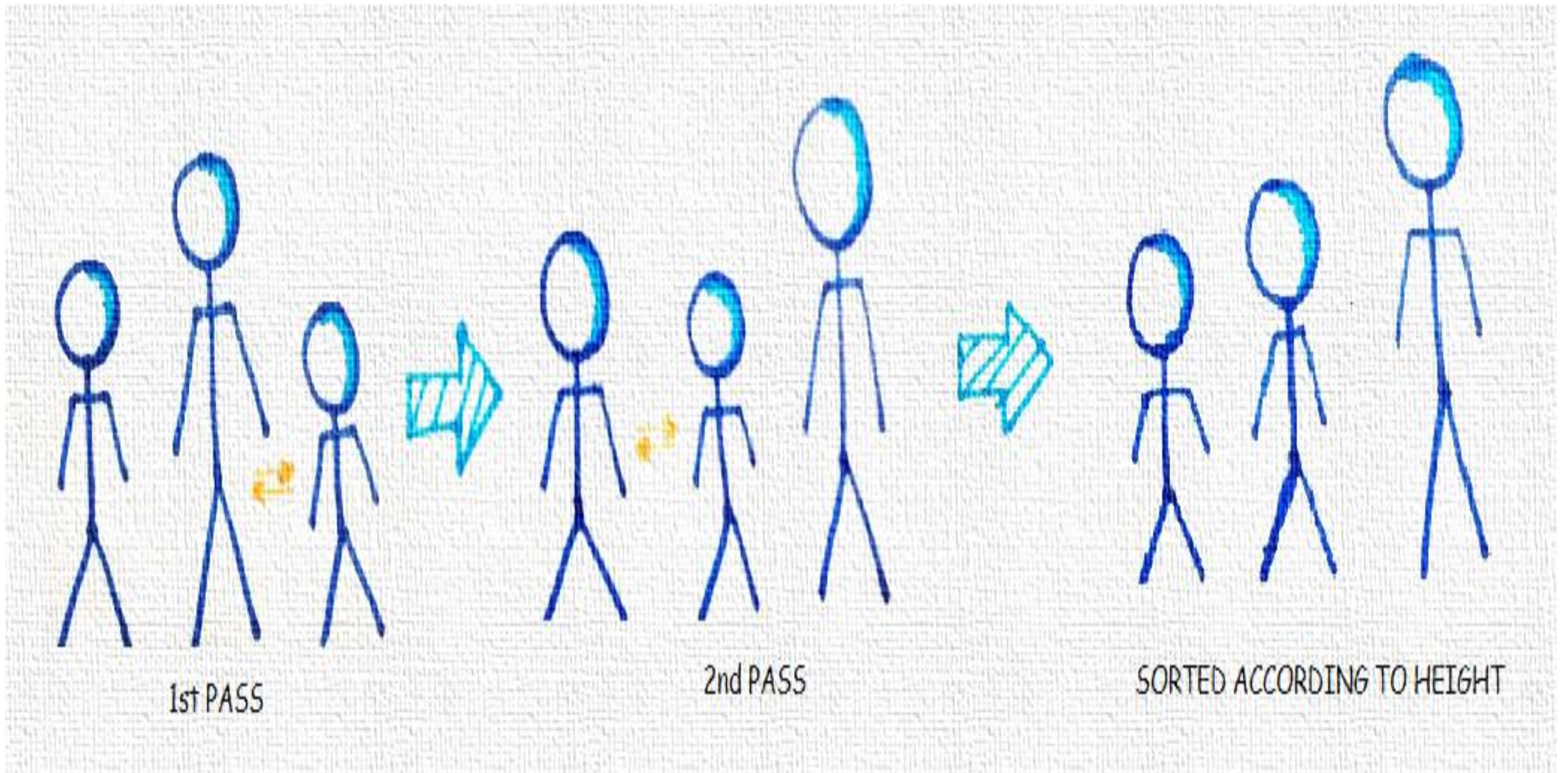
**Output: -12 -13 -5 -7 -3 -6 11 6 5**



# Sorting Techniques

# Introduction

- Sorting is among the most basic problems in algorithm design.
- We are given a sequence of items, each associated with a given key value.
- And the problem is to rearrange the items so that they are in an increasing(or decreasing) order by key.
- The methods of sorting can be divided into two categories:
  - Internal Sorting
  - External Sorting





# BUBBLESORT

- In bubble sort, each element is compared with its adjacent element.
- We begin with the 0<sup>th</sup> element and compare it with the 1<sup>st</sup> element.
- If it is found to be greater than the 1<sup>st</sup> element, then they are interchanged.
- In this way all the elements are compared (excluding last) with their next element and are interchanged if required
- On completing the first iteration, largest element gets placed at the last position. Similarly in second iteration second largest element gets placed at the second last position and soon.

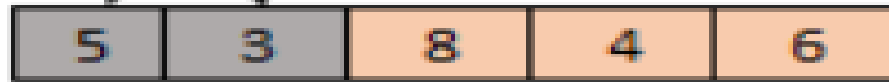
## Bubble sort example

Initial



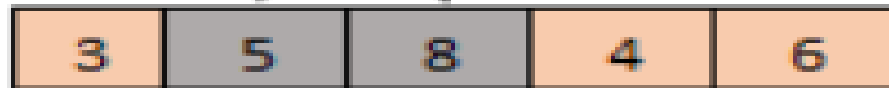
Initial Unsorted array

Step 1



Compare 1<sup>st</sup> and 2<sup>nd</sup>  
(Swap)

Step 2



Compare 2<sup>nd</sup> and 3<sup>rd</sup>  
(Do not Swap)

Step 3



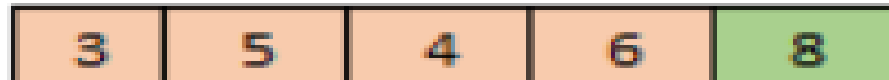
Compare 3<sup>rd</sup> and 4<sup>th</sup>  
(Swap)

Step 4



Compare 4<sup>th</sup> and 5<sup>th</sup>  
(Swap)

Step 5



Repeat Step 1-5 until  
no more swaps required

---

## Algorithm 1: Bubble sort

---

**Data:** Input array  $A[]$

**Result:** Sorted  $A[]$

*int*  $i, j, k$ ;

$N = \text{length}(A)$ ;

**for**  $j = 1$  **to**  $N$  **do**

**for**  $i = 0$  **to**  $N-1$  **do**

**if**  $A[i] > A[i+1]$  **then**

$\text{temp} = A[i]$ ;

$A[i] = A[i+1]$ ;

$A[i+1] = \text{temp}$ ;

**end**

**end**

**end**

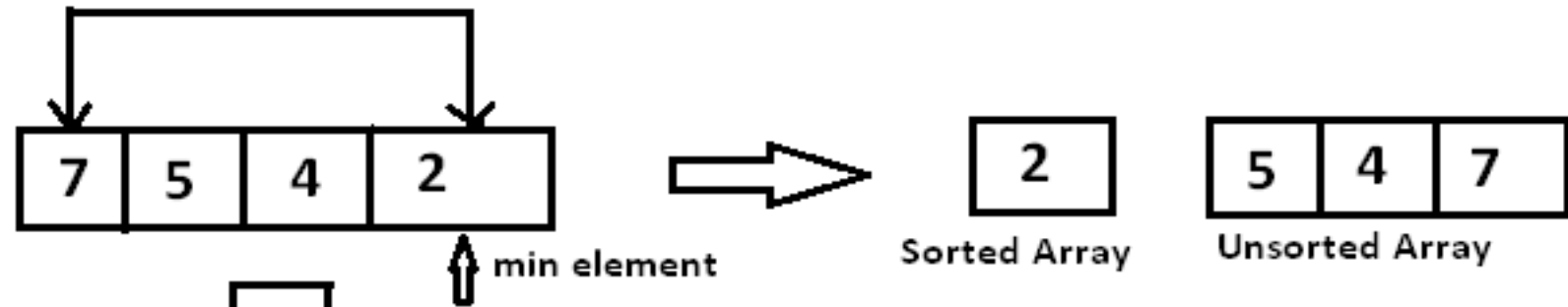
---



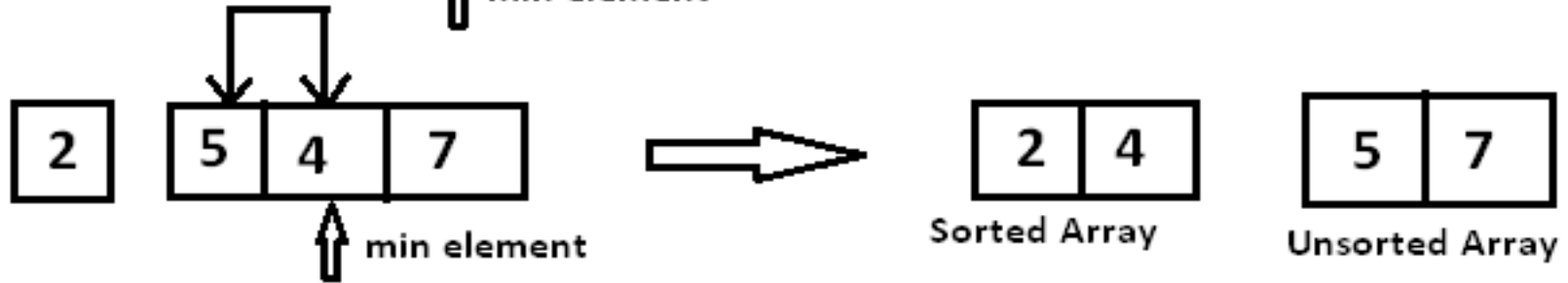
# SELECTION SORT

- Find the least( or greatest) value in the array, swap it into the leftmost(or rightmost) component, and then forget the leftmost component, Do this repeatedly.
- Let  $a[n]$  be a linear array of  $n$  elements. The selection sort works as follows:
- Pass 1: Find the location  $loc$  of the smallest element in the list of  $n$  elements  $a[0]$ ,  $a[1]$ ,  $a[2]$ ,  $a[3]$ , .....,  $a[n-1]$  and then interchange  $a[loc]$  and  $a[0]$ .
- Pass 2: Find the location  $loc$  of the smallest element in the sub-list of  $n-1$  elements  $a[1]$ ,  $a[2]$ ,  $a[3]$ , .....,  $a[n-1]$  and then interchange  $a[loc]$  and  $a[1]$  such that  $a[0]$ ,  $a[1]$  are sorted.
- Then we will get the sorted list  
 $a[0] \leq a[1] \leq a[2] \leq a[3] \dots \leq a[n-1]$

STEP 1.



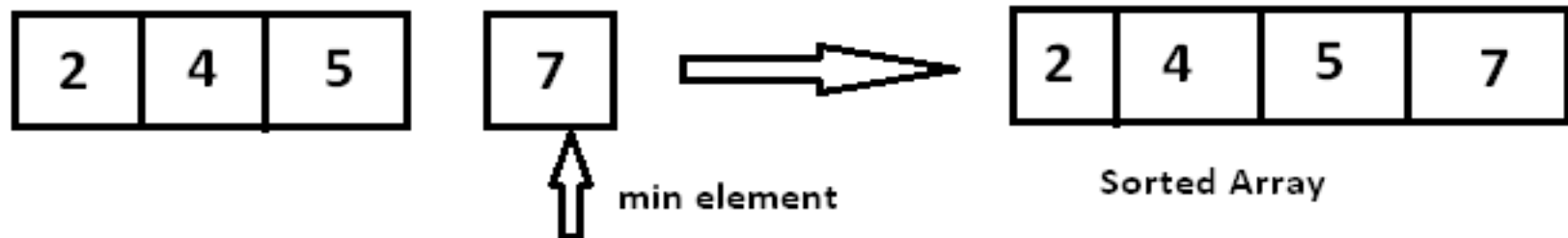
STEP 2.



STEP 3.



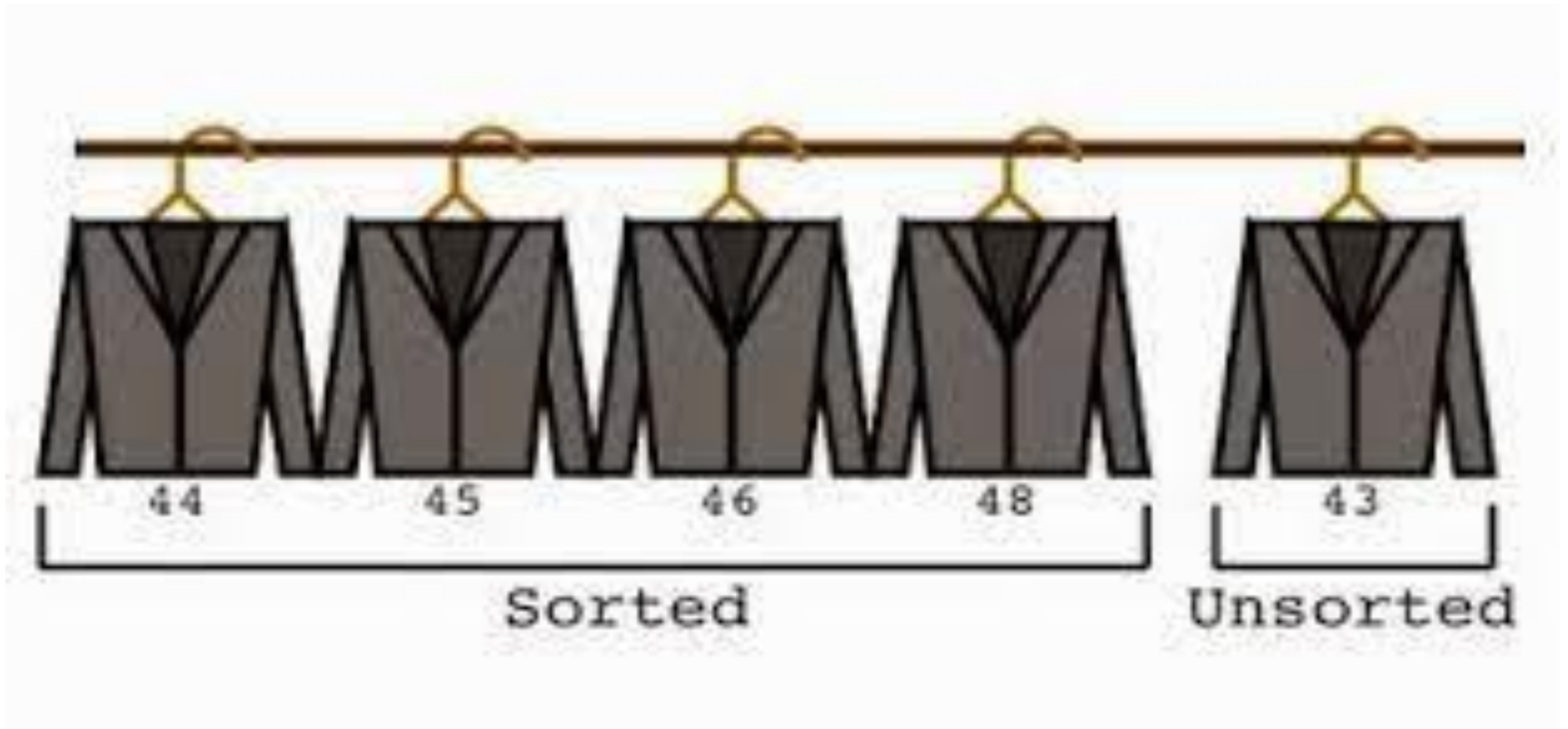
STEP 4.



## **Algorithm:**

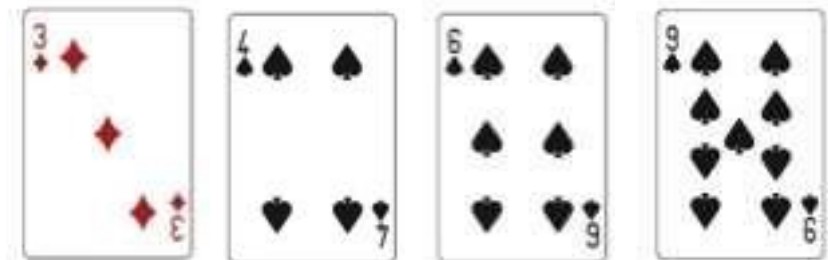
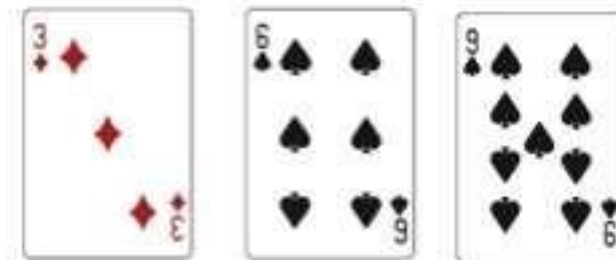
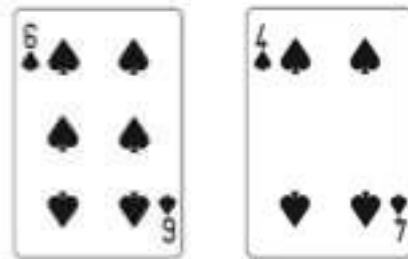
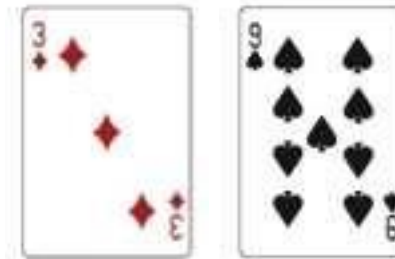
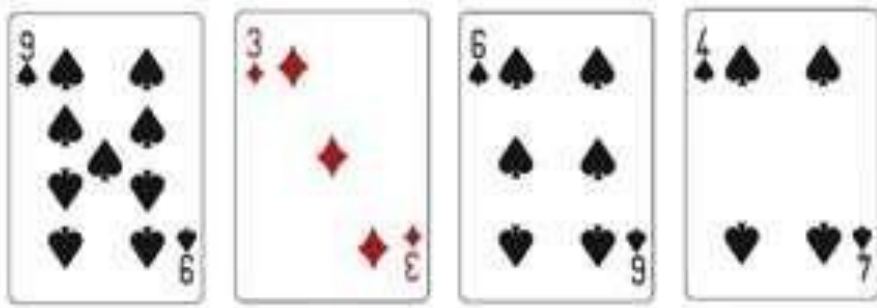
**SelectionSort(A)**

```
{  
    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]);  
}
```



# Insertion Sort

- Like sorting a hand of playing cards start with an empty hand and the cards facing down the table.
- Pick one card at a time from the table, and insert it into the correct position in the left hand.
- Compare it with each of the cards already in the hand, from right to left
- The cards held in the left hand are sorted.



## 2. Binary search:

Best case :  $O(1)$  (Mid element = key)

Worst Case :  $O(\log n)$

Space Complexity:  $O(1)$

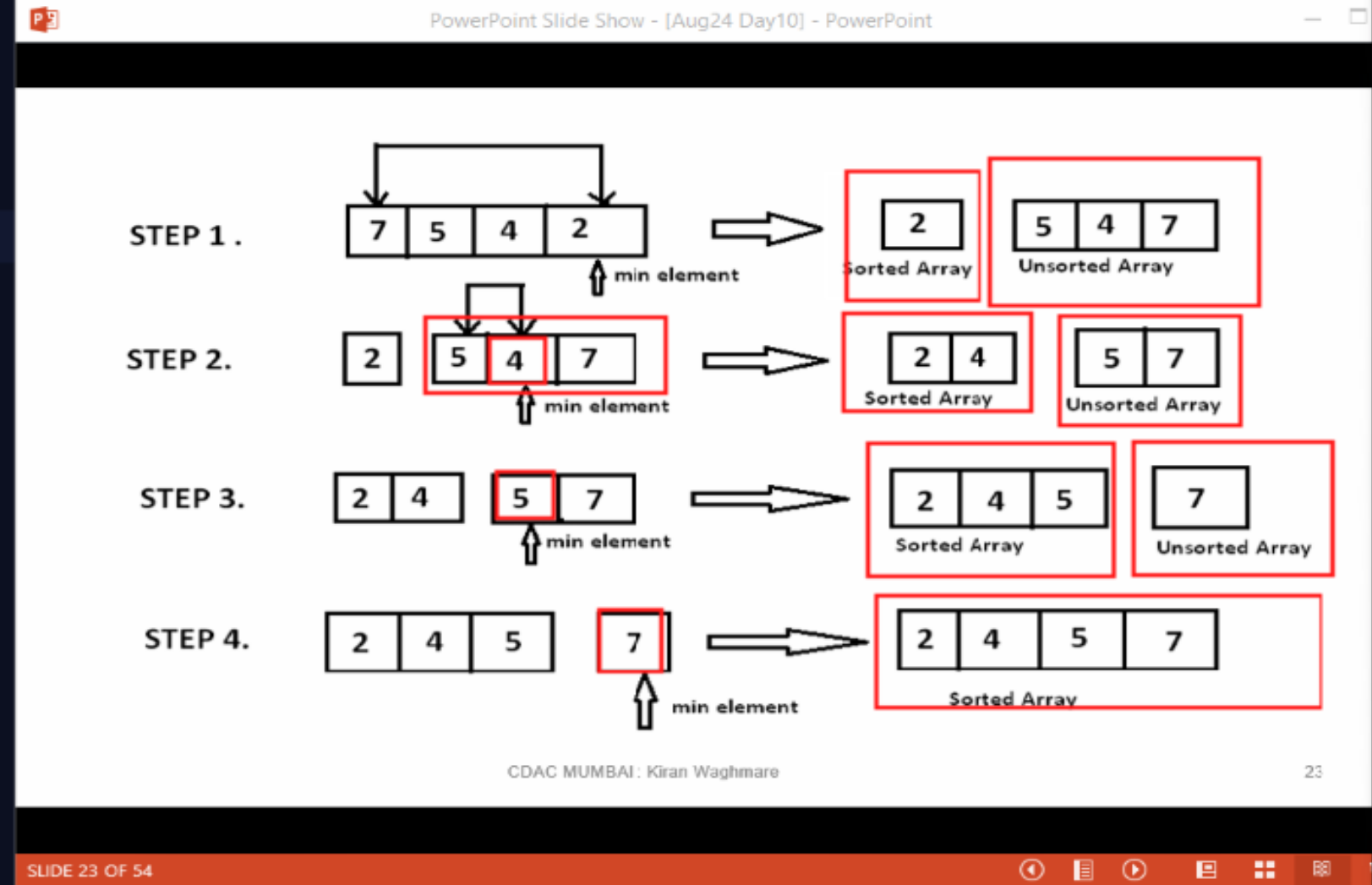
## Sorting:

### Bubble sort:

Best case :  $O(n)$

Worst Case :  $O(n^2)$

Space Complexity:  $O(1)$



# Insertion Sort

- Suppose an array  $a[n]$  with  $n$  elements. The insertion sort works as follows:

Pass 1:  $a[0]$  by itself is trivially sorted.

Pass 2:  $a[1]$  is inserted either before or after  $a[0]$  so that  $a[0], a[1]$  is sorted.

Pass 3:  $a[2]$  is inserted into its proper place in  $a[0], a[1]$  that is before  $a[0]$ , between  $a[0]$  and  $a[1]$ , or after  $a[1]$  so that  $a[0], a[1], a[2]$  is sorted.

pass  $N$ :  $a[n-1]$  is inserted into its proper place in  $a[0], a[1], a[2], \dots, a[n-2]$  so that  $a[0], a[1], a[2], \dots, a[n-1]$  is sorted with  $n$  elements.



## INSERTION-SORT(A)

<b>for</b> $j \leftarrow 2$ <b>to</b> $n$	$c_1$	$n$
<b>do</b> $\text{key} \leftarrow A[j]$	$c_2$	$n-1$
▷ Insert $A[j]$ into the sorted sequence $A[1 \dots j-1]$	0	$n-1$
$i \leftarrow j - 1$	$c_4$	$n-1$
<b>while</b> $i > 0$ and $A[i] > \text{key}$	$c_5$	$\sum_{j=2}^n t_j$
<b>do</b> $A[i+1] \leftarrow A[i]$	$c_6$	$\sum_{j=2}^n (t_j - 1)$
$i \leftarrow i - 1$	$c_7$	$\sum_{j=2}^n (t_j - 1)$
$A[i+1] \leftarrow \text{key}$	$c_8$	$n-1$

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

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

## Insertion Sort Execution Example

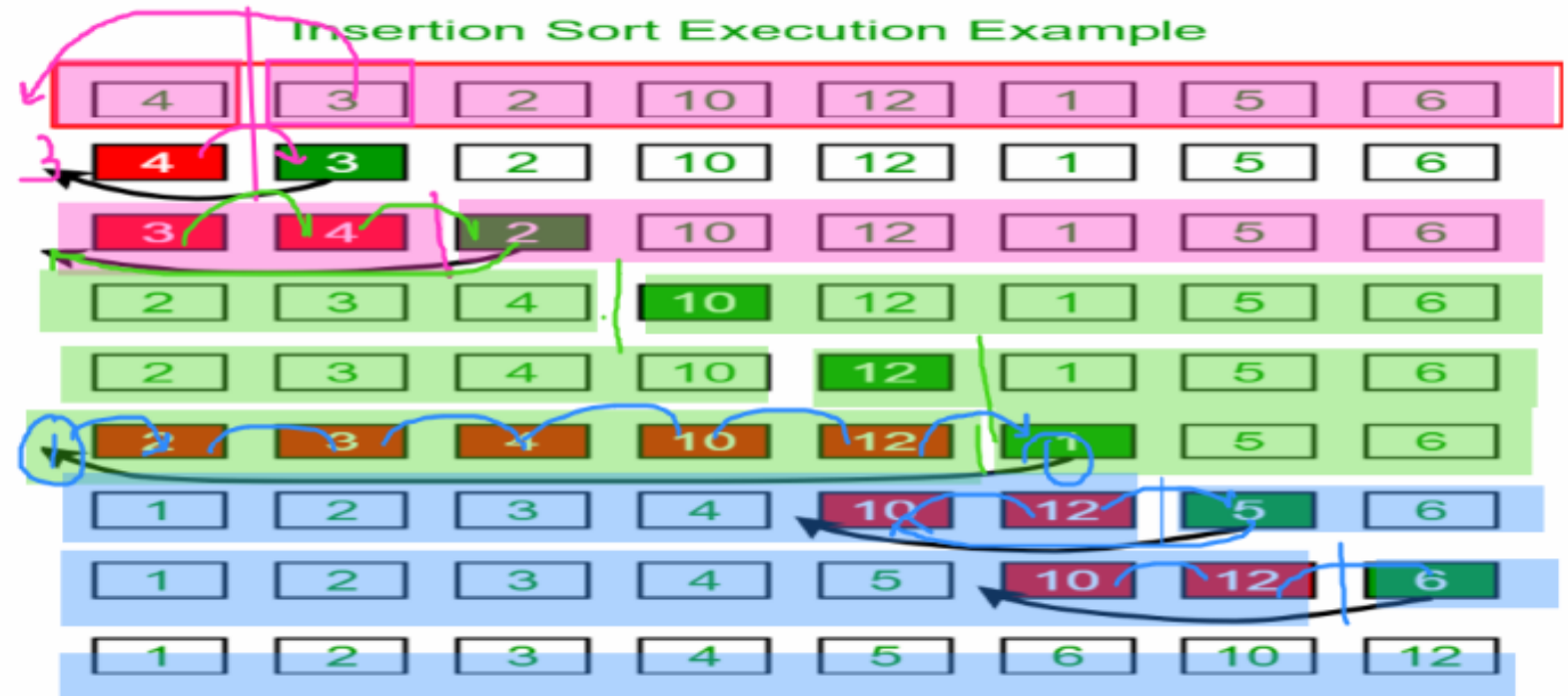


```
class ISorting{
```

```
static void isort(int a1[])  
{  
    int n = a1.length;  
}
```

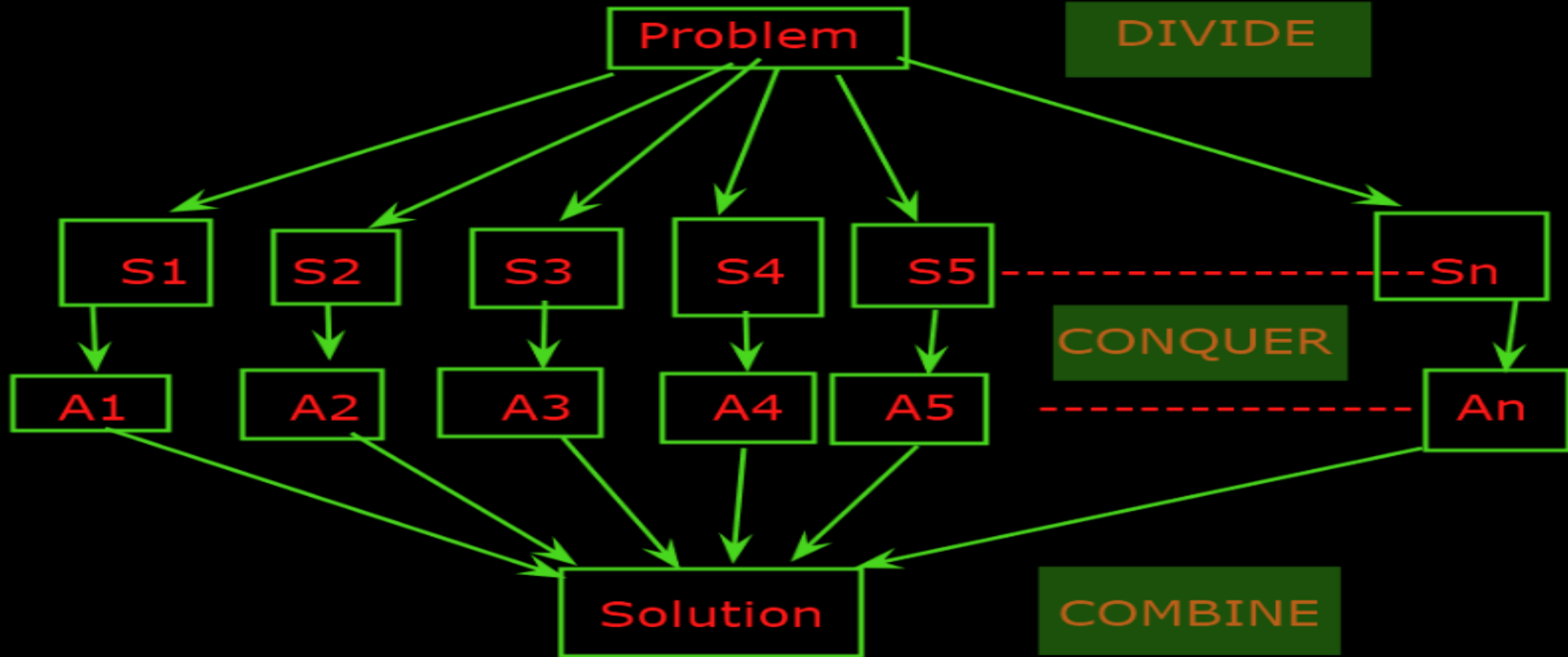
```
static void display(int a1[])  
{  
    for(int i=0;i<a1.length  
    {  
        System.out.print(  
    }  
}
```

```
public static void main(String args[])  
{
```



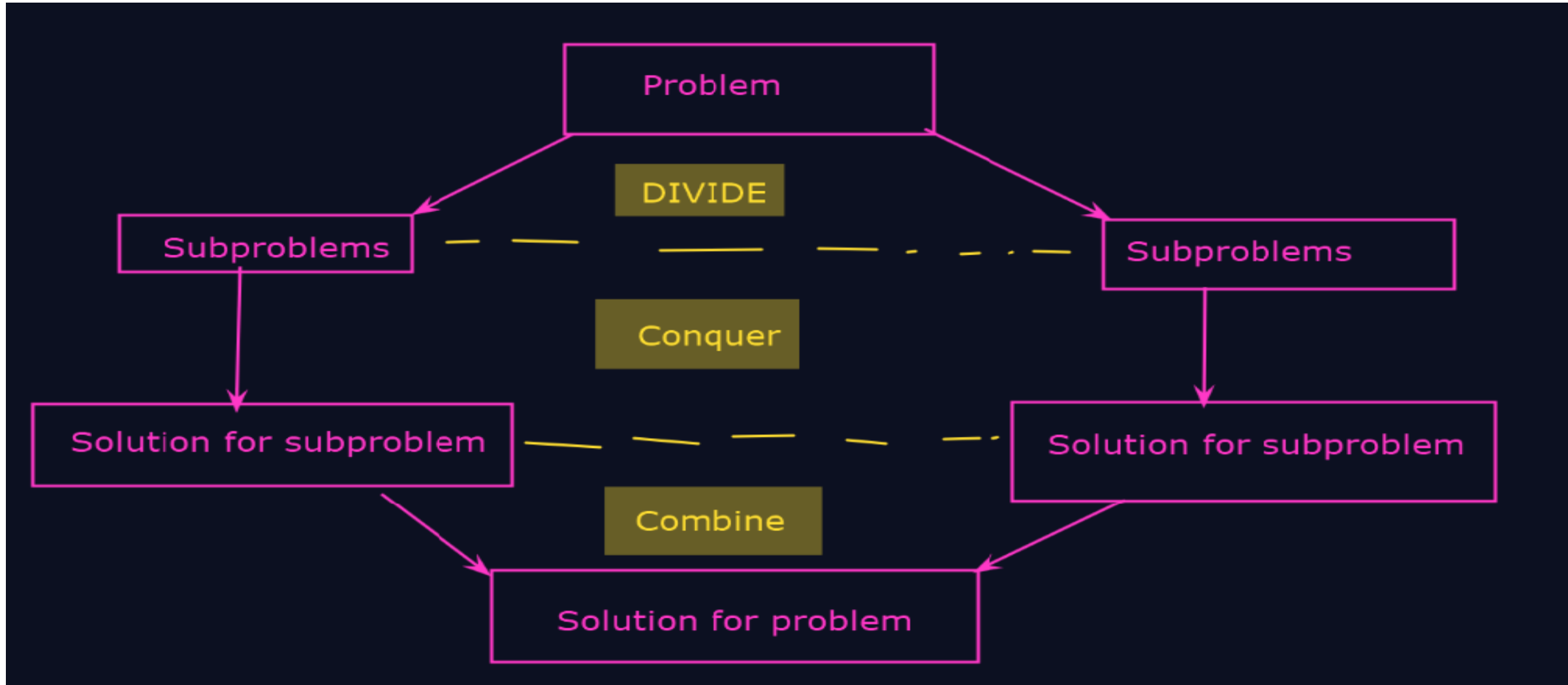
# Divide and conquer algorithms

- The sorting algorithms we've seen so far have worst-case running times of  $O(n^2)$
- When the size of the input array is large, these algorithms can take a long time to run.
- Now we will discuss two sorting algorithms whose running times are better
  - Merge Sort
  - Quick Sort



- breaks a problem into sub problems
- similar to original problem
- recursive strategy is used to solve problem
- combine all to solution to get the final solution of big problems

# Divide-and-conquer



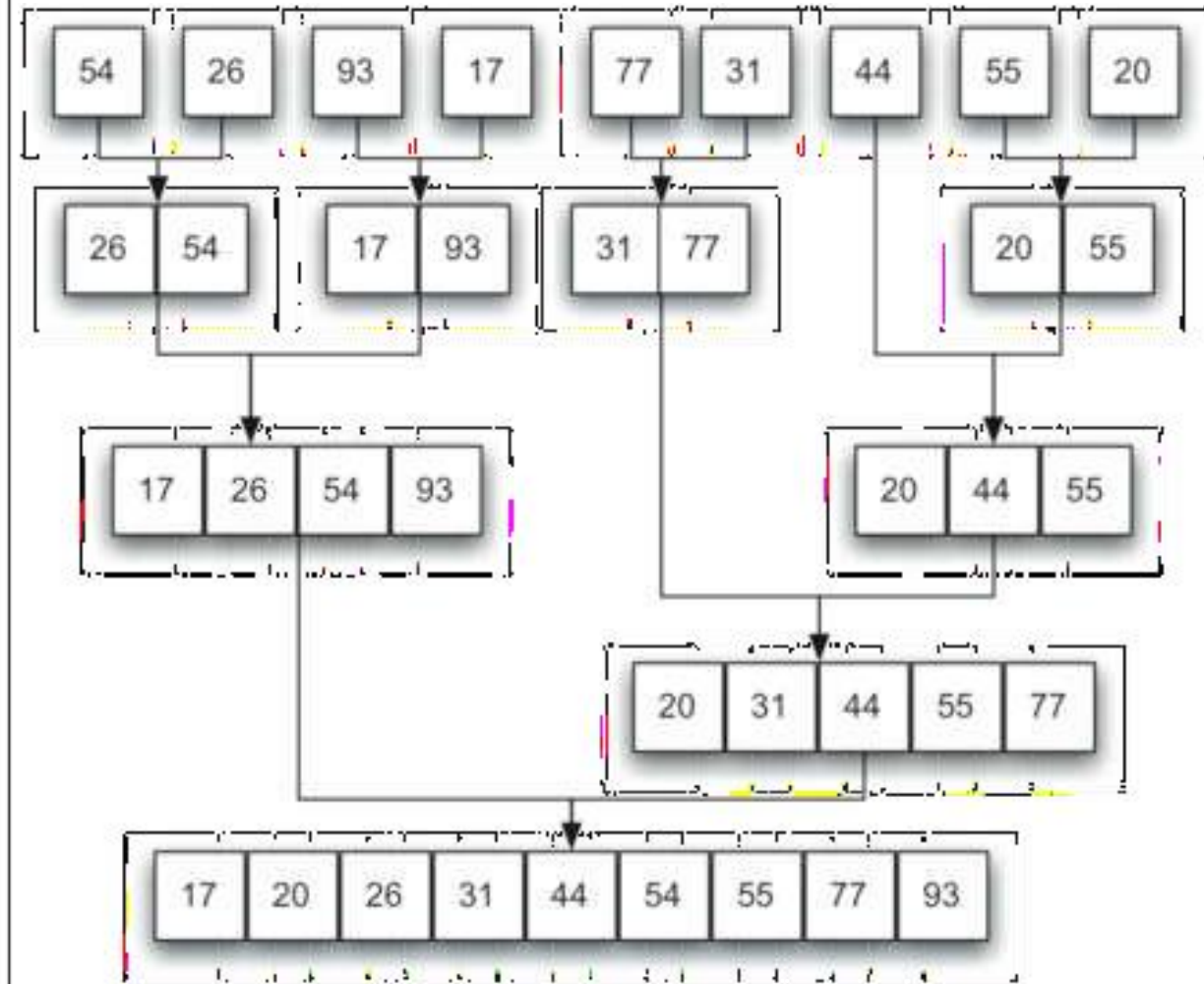
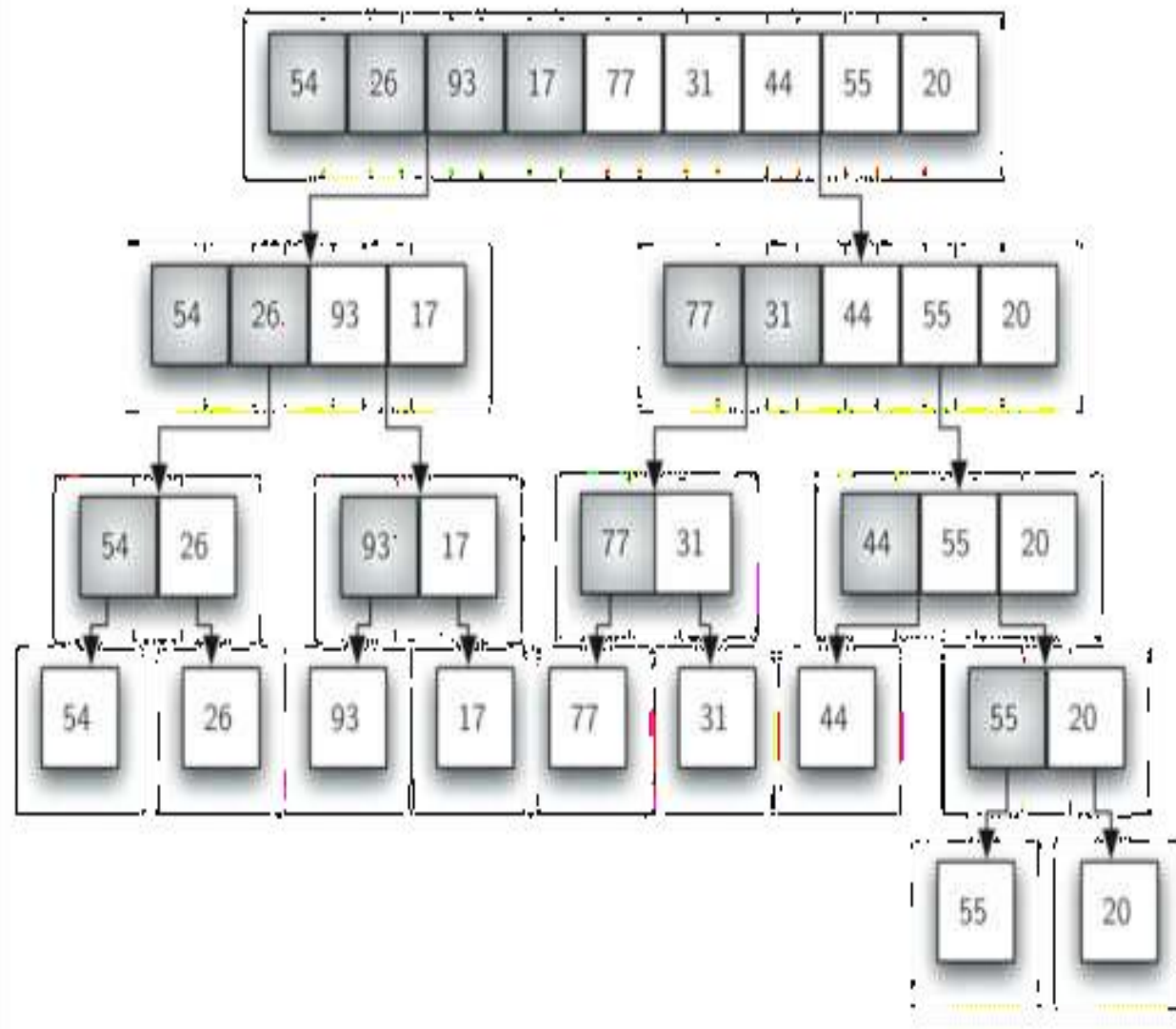
# Merge Sort

- Merge sort is a sorting technique based on divide and conquer technique.
- Merge sort first divides the array into equal halves and then combines them in a sorted manner.
- With worst-case time complexity being  $O(n \log n)$ , it is one of the most respected algorithms.

# Merge Sort

- Because we're using divide-and-conquer to sort, we need to decide what our sub problems are going to be.
- Full Problem: Sort an entire Array
- Sub Problem: Sort a sub array
- Lets assume  $\text{array}[p..r]$  denotes this subarray of array.
- For an array of  $n$  elements, we say the original problem is to sort  $\text{array}[0..n-1]$





# Merge Sort

- Here's how merge sort uses divide and conquer
  1. *Divide* by finding the number  $q$  of the position midway between  $p$  and  $r$ . Do this step the same way we found the midpoint in binary search: add  $p$  and  $r$ , divide by 2, and round down.
  2. *Conquer* by recursively sorting the subarrays in each of the two sub problems created by the divide step. That is, recursively sort the subarray  $\text{array}[p..q]$  and recursively sort the subarray  $\text{array}[q+1..r]$ .
  3. *Combine* by merging the two sorted subarrays back into the single sorted subarray  $\text{array}[p..r]$ .

### *Merge Sort:*

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

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

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

```
Merge (A, p, q, r)
1   n1 ← (q - p) + 1
2   n2 ← (r - q)
3   create arrays L[1..n1+1] and R[1..n2+1]
4   for i ← 1 to n1 do
5       L[i] ← A[(p + i) - 1]
6   for j ← 1 to n2 do
7       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 do
12.5    count ← count + 1
13      if L[i] ≤ R[j]
14          then A[k] ← L[i]
15              i ← i + 1
16      else A[k] ← R[j]
17          j ← j + 1
```

```

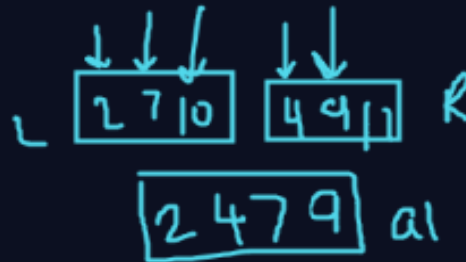
{
    int n1=m-l+1;
    int n2=h-m;

    int L[] = new int[n1];
    int R[] = new int[n2];

    for(int i=0;i<n1;i++)
        L[i]=a1[l+i]; //Left array elements
    for(int j=0;j<n2;j++)
        R[j]=a1[m+1+j]; //Right array elements

    int i=0, j=0;
    int k=l;
    while(i<n1 && j<n2)
    {
        if(L[i] <= R[j])
        {
            a1[k]=L[i];
            i++;
        }
        else
        {
            a1[k]=R[j];
            j++;
        }
        k++;
    }
    while(i<n1)
    {

```



Sort:

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

```

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

```

Here is the modified algorithm for the Merge function used by Merge Sort:

```

(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

```

CDAC MUMBAI : Kiran Waghmare

# Quick Sort

- Quick sort is one of the most popular sorting techniques.
- As the name suggests the quick sort is the fastest known sorting algorithm in practice.
- It has the best average time performance.
- It works by partitioning the array to be sorted and each partition in turn sorted recursively. Hence also called partition exchange sort.

# Quick Sort

- In partition one of the array elements is chosen as a pivot element
- Choose an element  $\text{pivot} = a[n-1]$ . Suppose that elements of an array  $a$  are partitioned so that pivot is placed into position  $i$  and the following condition hold:
  - Each element in position 0 through  $i-1$  is less than or equal to pivot
  - Each of the elements in position  $i+1$  through  $n-1$  is greater than or equal to key
- The pivot remains at the  $i^{\text{th}}$  position when the array is completely sorted. Continuously repeating this process will eventually sort an array.

## Merge Sort:

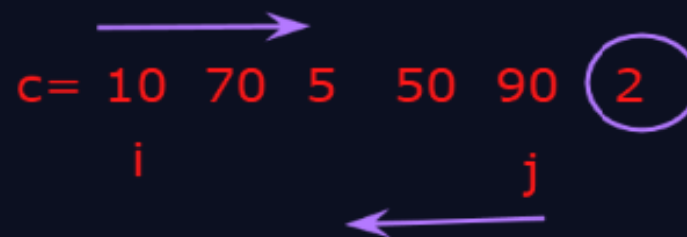
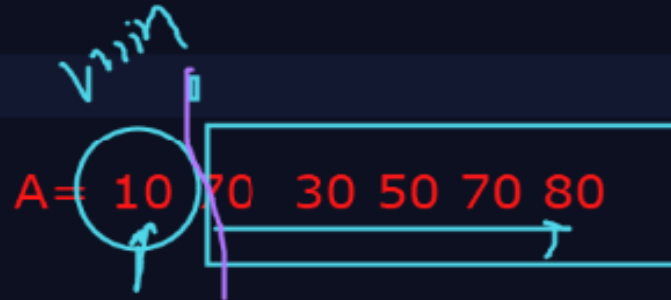
Time Complexity:  $O(n \log n)$

Space Complexity:  $O(n)$

Both sorting: Internal/External  
stable technique

## Quick sort:

Pivot element



# Algorithm

- Choosing a pivot
  - To partition the list we first choose a pivot element
- Partitioning
  - Then we partition the elements so that all those with values less than pivot are placed on the left side and the higher value on the right
  - Check if the current element is less than the pivot.
    - If lesser replace it with the current element and move the wall up one position
    - else move the pivot element to current element and vice versa
- Recur
  - Repeat the same partitioning step unless all elements are sorted

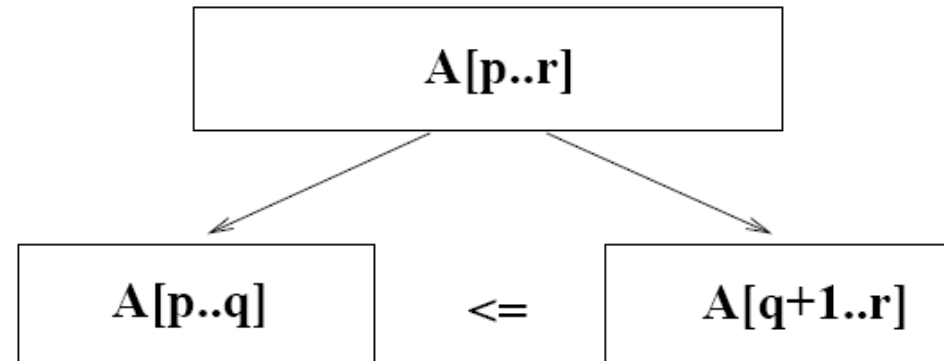
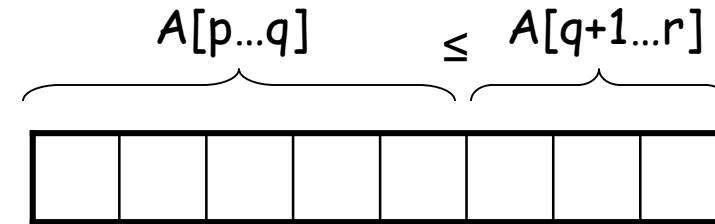


# Quicksort

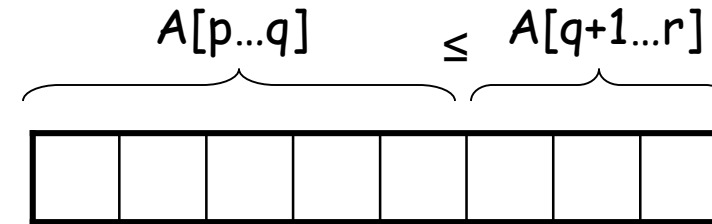
- Sort an array  $A[p..r]$

- Divide

- Partition the array  $A$  into 2 subarrays  $A[p..q]$  and  $A[q+1..r]$ , such that each element of  $A[p..q]$  is smaller than or equal to each element in  $A[q+1..r]$
- Need to find index  $q$  to partition the array



# Quicksort



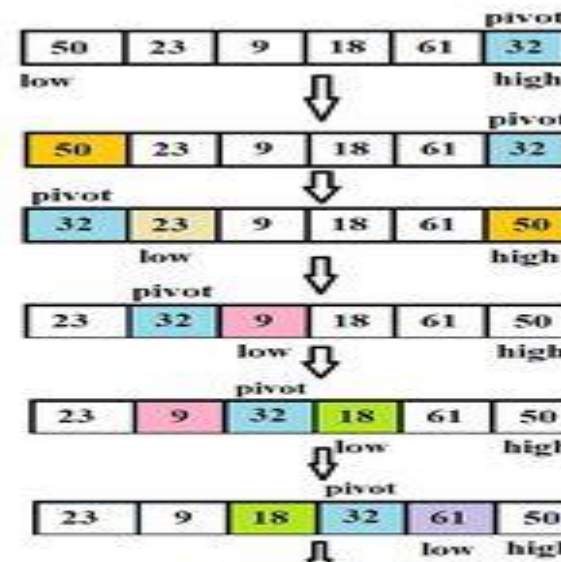
- **Conquer**

- Recursively sort  $A[p..q]$  and  $A[q+1..r]$  using Quicksort

- **Combine**

- Trivial: the arrays are sorted in place
- No additional work is required to combine them
- The entire array is now sorted

# Quick Sort



```

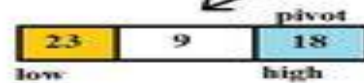
if arr[low] > arr[pivot]:
    swap(arr[low], arr[high])
    low++;
else:
    continue;
    
```



if low >= high:  
stop;

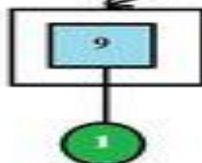
Quick Sort  
on Left side

Quick Sort  
on Right side



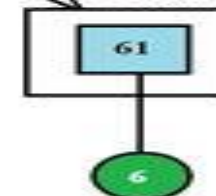
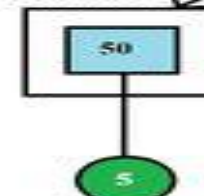
Quick Sort  
on Left side

Quick Sort  
on Right side



Quick Sort  
on Left side

Quick Sort  
on Right side



Final Sorted Array:



The following procedure implements quicksort:

QUICKSORT( $A, p, r$ )

```
1  if  $p < r$ 
2       $q = \text{PARTITION}(A, p, r)$ 
3      QUICKSORT( $A, p, q - 1$ )
4      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( $A, p, r$ )

```
1   $x = A[r]$ 
2   $i = p - 1$ 
3  for  $j = p$  to  $r - 1$ 
4      if  $A[j] \leq x$ 
5           $i = i + 1$ 
6          exchange  $A[i]$  with  $A[j]$ 
7  exchange  $A[i + 1]$  with  $A[r]$ 
8  return  $i + 1$ 
```

# QUICKSORT

Best

Average

Worst

$O(n \log n)$

$O(n \log n)$

$O(n^2)$



Recursion



Divide and Conquer

Array

**sort** (A)

1. quickSort (A, 0, n - 1)

**end**

**quickSort** (A, left, right)

1. **if** (left < right) **then**

2. pi = partition (A, left, right)

3. quickSort (A, left, pi - 1)

4. quickSort (A, pi + 1, right)

**end**

**partition** (A, left, right)

1. p = select pivot in A[left, right]

2. swap A[p] and A[right]

3. store = left

4. **for** i = left **to** right - 1 **do**

5. **if** (A[i] ≤ A[right]) **then**

6. swap A[i] and A[store]

7. store++

8. swap A[store] and A[right]

9. **return** store

**end**

