

# **Searching & Sorting**

# Sorting & Searching

## ▶ Searching

- Linear/Sequential Search
- Binary Search

## ▶ Sorting

- Bubble sort
- Selection Sort
- Insertion Sort
- Quick Sort
- Merge Sort

# Linear/Sequential Search

- ▶ In computer science, **linear search** or **sequential search** is a method for finding a particular value in a list that consists of **checking every** one of its **elements**, **one at a time** and in sequence, **until the desired one is found**.
- ▶ Linear search is the simplest search algorithm.
- ▶ It is a special case of brute-force search.
- ▶ Its **worst case cost** is proportional to the **number of elements in the list**.

# Sequential Search – Algorithm & Example

# Input: Array A, integer key  
# Output: first index of key in A  
# or -1 if not found

## Algorithm: Linear\_Search

```
for i = 0 to last index of A:  
    if A[i] equals key:  
        return i
```

```
return -1
```

Search for **1** in given array

2	9	3	1	8
---	---	---	---	---

Comparing value of  $i^{\text{th}}$  index with element to be search one by one until we get searched element or end of the array

Step 1:  $i=0$

2	9	3	1	8
---	---	---	---	---

↑  
 $i$

Step 1:  $i=2$

2	9	3	1	8
---	---	---	---	---

↑  
 $i$

Step 1:  $i=1$

2	9	3	1	8
---	---	---	---	---

↑  
 $i$

Step 1:  $i=3$

2	9	3	<b>1</b>	8
---	---	---	----------	---

↑  
 $i$

Element found at  $i^{\text{th}}$  index,  $i=3$

# Binary Search

- ▶ If we have an **array** that is **sorted**, we can use a much more **efficient algorithm** called a **Binary Search**.
- ▶ In binary search **each time** we **divide array** into **two equal half** and **compare middle element** with **search element**.
- ▶ Searching Logic
  - If **middle element** is **equal to search element** then we got that element and **return that index**
  - if **middle element** is **less than search element** we **look right part** of array
  - if **middle element** is **greater than search element** we look **left part** of array.

# Binary Search - Algorithm

# Input: Sorted Array A, integer key

# Output: first index of key in A,

# or -1 if not found

**Algorithm: Binary\_Search (A, left, right)**

left = 0, right = n-1

while left < right

    middle = index halfway between left, right

    if A[middle] matches key

        return middle

    else if key less than A[middle]

        right = middle - 1

    else

        left = middle + 1

return -1

# Binary Search - Algorithm

Search for **6** in given array

	-1	5	6	18	19	25	46	78	102	114
Index	0	1	2	3	4	5	6	7	8	9
	↑ left									↑ right

Key=6, No of Elements = 10, so left = 0, right=9

Step 1:

**middle index** =  $(\text{left} + \text{right}) / 2 = (0+9)/2 = 4$

**middle element value** =  $a[4] = 19$

**Key=6** is **less than** middle element = **19**, so **right** = middle - 1 = 4 - 1 = **3**, **left** = 0

	-1	5	6	18	19	25	46	78	102	114
Index	0	1	2	3	4	5	6	7	8	9
	↑ left			↑ right						

# Binary Search - Algorithm

**Step 2:**

**middle index** =  $(\text{left} + \text{right}) / 2 = (0+3)/2 = 1$

**middle element value** =  $a[1] = 5$

**Key=6** is **greater than** middle element = **5**, so **left** = middle + 1 = 1 + 1 = **2**, **right** = 3

	-1	5	6	18	19	25	46	78	102	114
Index	0	1	2	3	4	5	6	7	8	9
			↑	↑						
			left	right						

**Step 3:**

**middle index** =  $(\text{left} + \text{right}) / 2 = (2+3)/2 = 2$

**middle element value** =  $a[2] = 6$

**Key=6** is **equals to** middle element = **6**, so **element found**

	-1	5	6	18	19	25	46	78	102	114
Index	0	1	2	3	4	5	6	7	8	9
			↑							
			Element Found							



# Selection Sort

- ▶ Selection sort is a simple sorting algorithm.
- ▶ The **list** is **divided into two parts**,
  - ↳ The **sorted part** at the **left end** and
  - ↳ The **unsorted part** at the **right end**.
  - ↳ Initially, the sorted part is empty and the unsorted part is the entire list.
- ▶ The **smallest element** is **selected** from the **unsorted array** and **swapped** with the **leftmost element**, and that element becomes a part of the sorted array.
- ▶ This process continues moving unsorted array boundary by one element to the right.
- ▶ This algorithm is **not suitable** for **large data sets** as its average and worst case complexities are of  **$O(n^2)$** , where  $n$  is the number of items.

# Selection Sort

Unsorted Array

5	1	12	-5	16	2	12	14
---	---	----	----	----	---	----	----

Step 1 :

Unsorted Array

5	1	12	-5	16	2	12	14
---	---	----	----	----	---	----	----

0 1 2 3 4 5 6 7

Step 2 :

Unsorted Array (elements 0 to 7)

-5	1	12	5	16	2	12	14
----	---	----	---	----	---	----	----

0 1 2 3 4 5 6 7

Swap

Min index = 0, value = 5

Find min value from  
Unsorted array

Index = 3, value = -5

# Selection Sort

**Step 3 :**

**Unsorted Array (elements 1 to 7)**

-5	1	12	5	16	2	12	14
0	1	2	3	4	5	6	7

**Min index = 1, value = 1**

Find min value from  
Unsorted array  
Index = 1, value = 1

No Swapping as min value is already at right place

**Step 4 :**

**Unsorted Array  
(elements 2 to 7)**

-5	1	2	5	16	12	12	14
0	1	2	3	4	5	6	7

↑                      ↑

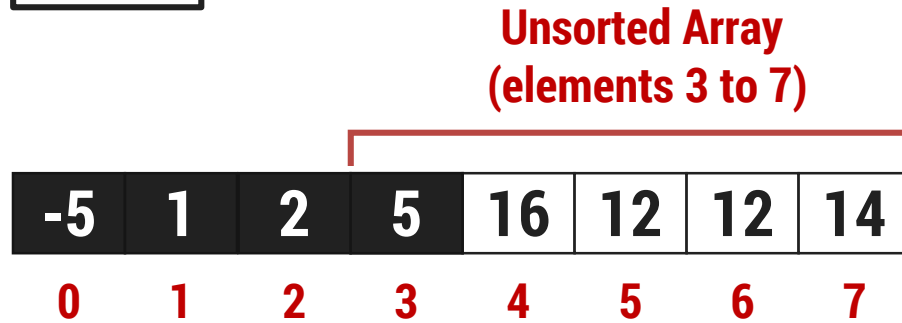
**Swap**

**Min index = 2, value = 12**

Find min value from  
Unsorted array  
Index = 5, value = 2

# Selection Sort

**Step 5 :**

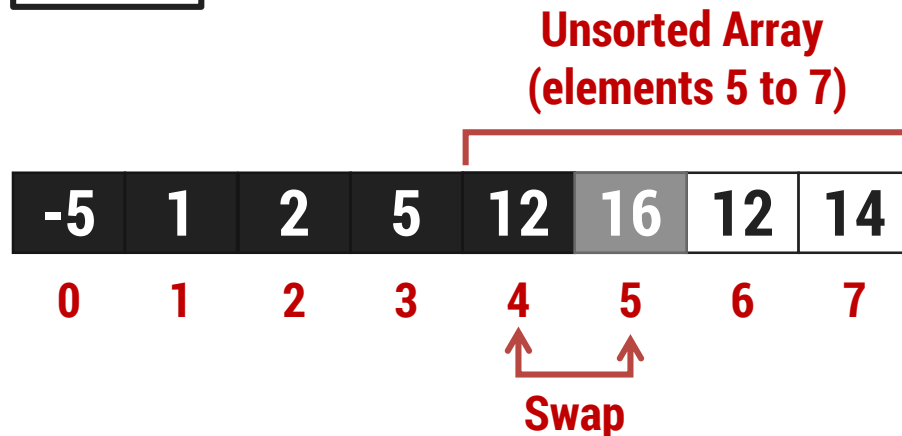


**Min index = 3, value = 5**

Find min value from  
Unsorted array  
Index = 3, value = 5

No Swapping as min value is already at right place

**Step 6 :**

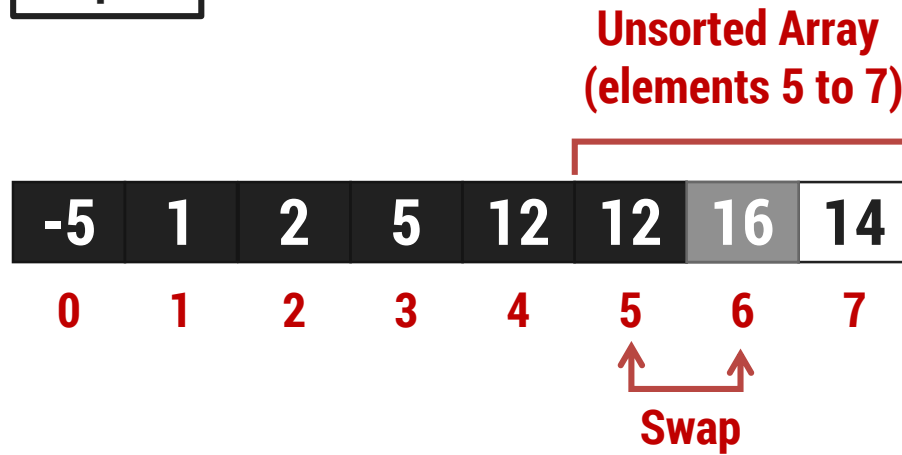


**Min index = 4, value = 16**

Find min value from  
Unsorted array  
Index = 5, value = 12

# Selection Sort

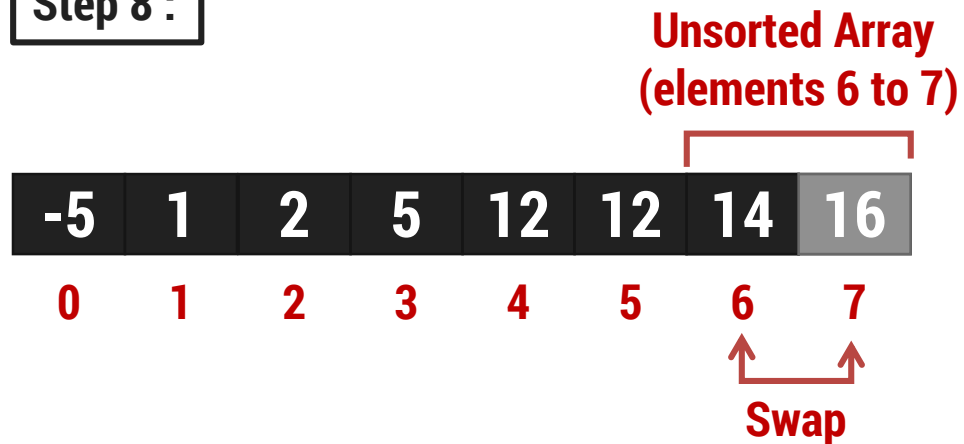
**Step 7 :**



**Min index = 5, value = 16**

Find min value from  
Unsorted array  
Index = 6, value = 12

**Step 8 :**



**Min index = 6, value = 16**

Find min value from  
Unsorted array  
Index = 7, value = 14

# SELECTION\_SORT(K,N)

- ▶ Given a **vector K** of **N** elements
- ▶ This procedure **rearrange** the **vector** in **ascending order** using **Selection Sort**
- ▶ The variable **PASS** denotes the **pass index** and position of the first element in the vector
- ▶ The variable **MIN\_INDEX** denotes the **position of the smallest element** encountered
- ▶ The variable **I** is used to index elements

# SELECTION\_SORT(K,N)

## 1. [Loop on the Pass index]

Repeat thru step 4 for PASS = 1,2,....., N-1

## 2. [Initialize minimum index]

MIN\_INDEX  $\leftarrow$  PASS

## 3. [Make a pass and obtain element with smallest value]

Repeat for I = PASS + 1, PASS + 2, ....., N

If  $K[I] < K[\text{MIN\_INDEX}]$

Then MIN\_INDEX  $\leftarrow$  I

## 4. [Exchange elements]

IF MIN\_INDEX  $\neq$  PASS

Then  $K[\text{PASS}] \leftrightarrow K[\text{MIN\_INDEX}]$

## 5. [Finished]

Return

# Bubble Sort

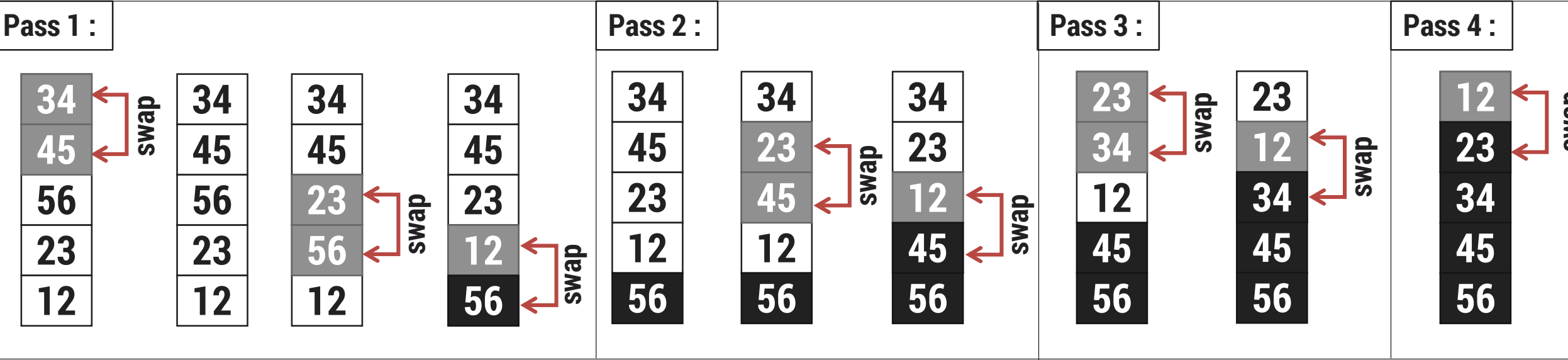
- ▶ Unlike **selection sort**, instead of finding the smallest record and performing the interchange, two records are **interchanged immediately** upon discovering that they are out of order
- ▶ During the **first pass**  $R_1$  and  $R_2$  are compared and **interchanged in case of out of order**, this process is repeated for records  $R_2$  and  $R_3$ , and so on.
- ▶ This method will cause records with **small key to move “bubble up”**,
- ▶ **After the first pass**, the record with **largest key** will be in the  $n^{\text{th}}$  position.
- ▶ On each successive pass, the records with the next largest key will be placed in position  $n-1$ ,  $n-2$  ....., 2 respectively
- ▶ This approach required at most  $n-1$  passes, The **complexity** of bubble sort is  **$O(n^2)$**



# Bubble Sort

## Unsorted Array

45	34	56	23	12
----	----	----	----	----



# BUBBLE\_SORT(K,N)

- ▶ Given a **vector K** of **N** elements
- ▶ This procedure **rearrange** the **vector** in **ascending order** using **Bubble Sort**
- ▶ The variable **PASS & LAST** denotes the **pass index** and position of the first element in the vector
- ▶ The variable **EXCHS** is used to count number of exchanges made on any pass
- ▶ The variable **I** is used to index elements

# Procedure: BUBBLE\_SORT (K, N)

## 1. [Initialize]

LAST  $\leftarrow$  N

## 2. [Loop on pass index]

Repeat thru step 5 for PASS = 1, 2, 3, ..., N-1

## 3. [Initialize exchange counter for this pass]

EXCHS  $\leftarrow$  0

## 4. [Perform pairwise comparisons on unsorted elements]

Repeat for I = 1, 2, ....., LAST - 1

IF K[I] > K[I+1]

Then K[I]  $\leftrightarrow$  K[I+1]

EXCHS  $\leftarrow$  EXCHS + 1

## 5. [Any exchange made in this pass?]

IF EXCHS = 0

Then Return (Vector is sorted, early return)

ELSE LAST  $\leftarrow$  LAST - 1

## 6. [Finished]

Return

# Merge Sort

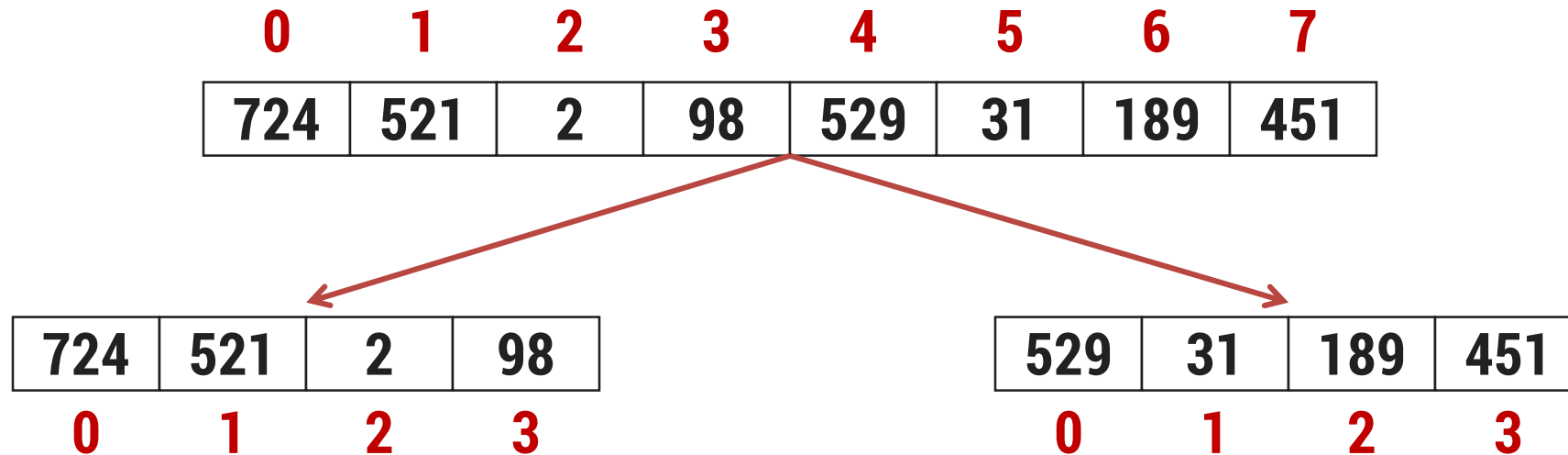
- ▶ The **operation of sorting** is closely related to **process of merging**
- ▶ Merge Sort is a **divide and conquer algorithm**
- ▶ It is based on the **idea of breaking down a list into several sub-lists** until each sub list consists of a **single element**
- ▶ **Merging those sub lists** in a manner that results into a sorted list
- ▶ **Procedure**
  - **Divide** the unsorted **list into N sub** lists, **each containing 1 element**
  - Take **adjacent pairs** of two singleton lists and **merge them** to form a **list of 2 elements**. N will now convert into N/2 lists of size 2
  - Repeat the process till a single sorted list of obtained
- ▶ Time complexity is  **$O(n \log n)$**

# Merge Sort

## Unsorted Array

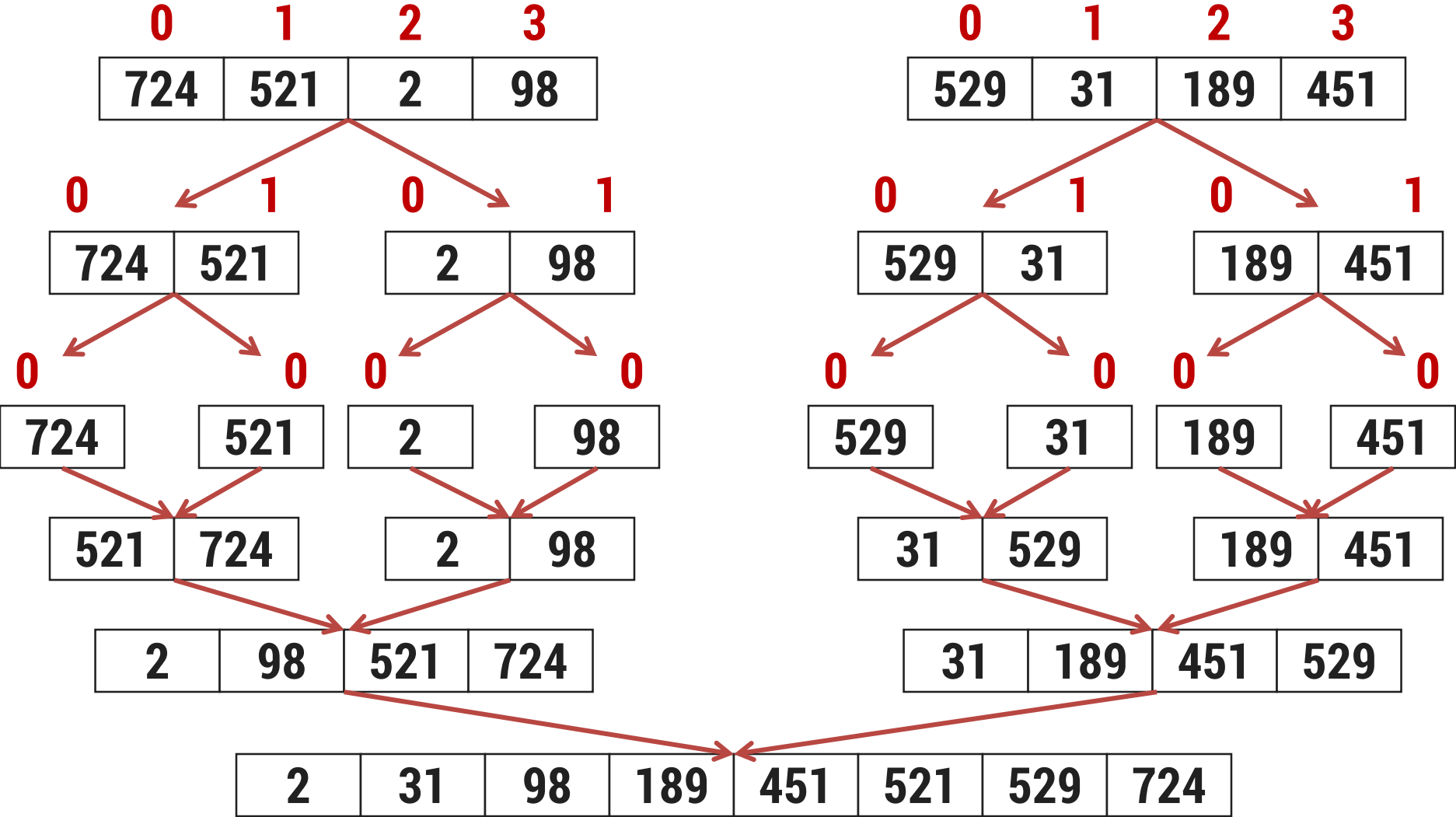
724	521	2	98	529	31	189	451
0	1	2	3	4	5	6	7

Step 1: Split the selected array (as evenly as possible)



# Merge Sort

Step: Select the left subarray, Split the selected array (as evenly as possible)



# Merge sort algorithm

MergeSort (Arr[], LB, UB)

If  $LB < UB$

1. Find the middle point to divide the array into two halves:

$Middle\ MID = (LB + UB) / 2$

2. Call mergeSort for first half:

Call mergeSort (Arr, LB, MID)

3. Call mergeSort for second half:

Call mergeSort (Arr, MID+1, UB)

4. Merge the two halves sorted in step 2 and 3:

Call merge (Arr, LB, MID, UB)

```

Merge (A, LB, MID, UB)
{
i=LB;
J=MID+1;
K=LB;
while (i<= MID && j<= UB)
{
if (a[i] <= a[j])
{
b[k]= a[i];
i++;
}
else
{
b[k]= a[j];
j++;
}
k++;
}
}

```

```

if (i> MID)
{ while (j<= UB)
{
b[k] = a[i]; j++; k++;
}
}
else
{
while (i<= MID)
{
b[k] = a[i], i++, k++;
}
}
for (k= LB; k <= UB; k++)
{ a [k] = b[k];
}
}

```



# Quick Sort

- ▶ **Quick sort** is a highly efficient sorting algorithm and is based on **partitioning of array** of data into **smaller arrays**.
- ▶ Quick Sort is **divide and conquer** algorithm.
- ▶ At each step of the method, the goal is to place a particular record in its final position within the table,
- ▶ In doing so all the records which precedes this record will have smaller keys, while all records that follows it have larger keys.
- ▶ This particular records is **termed pivot element**.
- ▶ The same process can then be applied to each of these subtables and repeated until all records are placed in their positions

# Quick Sort

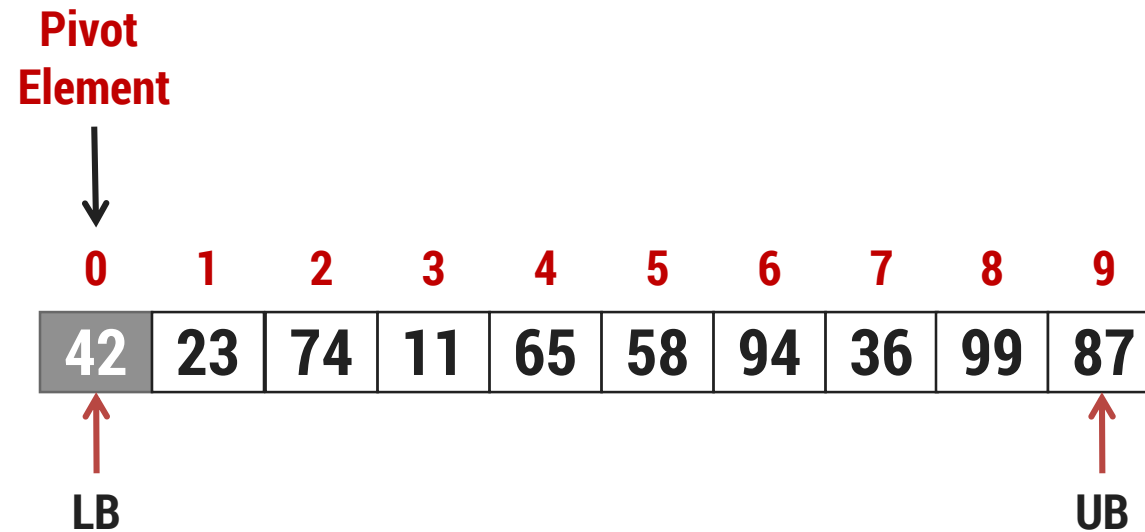
- ▶ There are many **different versions** of Quick Sort **that pick pivot** in different ways.
  - ➔ Always pick **first element as pivot**. (in our case we have consider this version).
  - ➔ Always pick **last element as pivot** (implemented below)
  - ➔ Pick a **random element as pivot**.
  - ➔ Pick **median as pivot**.
- ▶ Quick sort partitions an array and then calls itself recursively twice to sort the two resulting sub arrays.
- ▶ This algorithm is quite **efficient for large-sized data sets**
- ▶ Its average and **worst case complexity** are of  **$O(n^2)$** , where n is the number of items.

# Quick Sort

## Sort Following Array using Quick Sort Algorithm

We are considering **first element as pivot element**, so **Lower bound** is **First Index** and **Upper bound** is **Last Index**

We need to find our proper position of Pivot element in sorted array and perform same operations recursively for two sub array



# Quick Sort

FLAG  $\leftarrow$  true

IF LB < UB

Then

I  $\leftarrow$  LB

J  $\leftarrow$  UB + 1

KEY  $\leftarrow$  K[LB]

**Repeat While FLAG = true**

I  $\leftarrow$  I+1

Repeat While K[I] < KEY

I  $\leftarrow$  I + 1

J  $\leftarrow$  J - 1

Repeat While K[J] > KEY

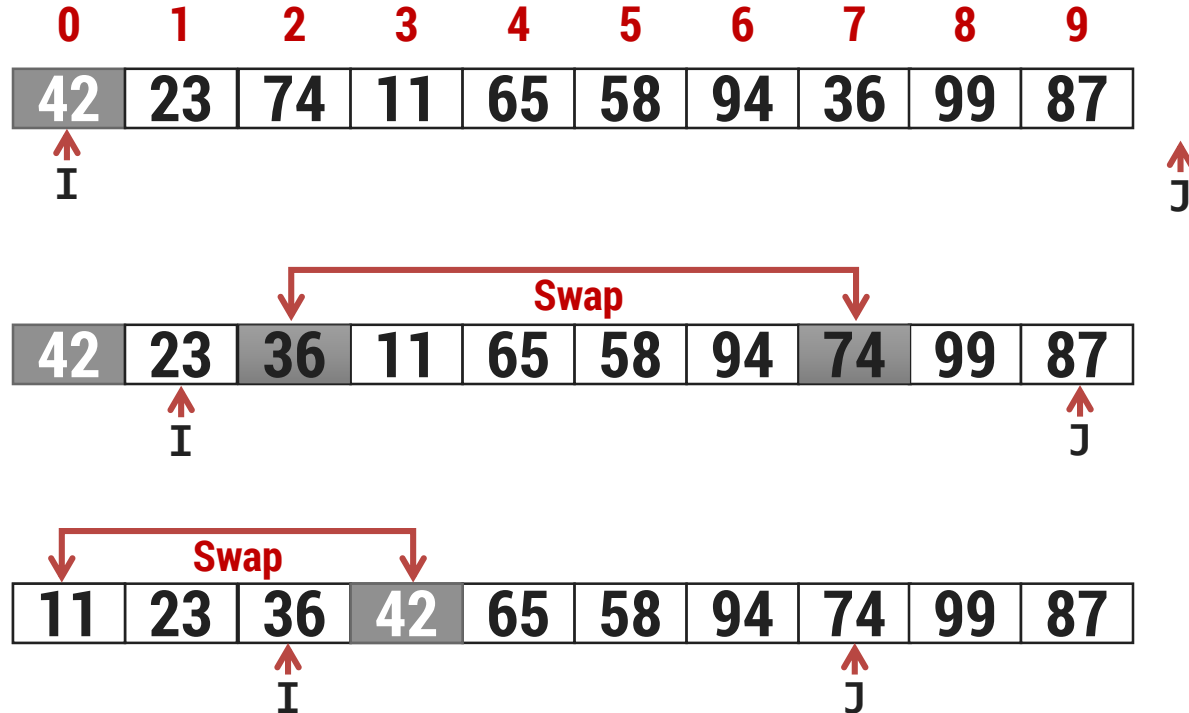
J  $\leftarrow$  J - 1

IF I < J

Then K[I]  $\longleftrightarrow$  K[J]

Else FLAG  $\leftarrow$  FALSE

K[LB]  $\longleftrightarrow$  K[J]



LB = 0, UB = 9      KEY = 42

I = 0

J = 10

FLAG = true

# Quick Sort

```
FLAG ← true
```

```
IF LB < UB
```

```
Then
```

```
  I ← LB
```

```
  J ← UB + 1
```

```
  KEY ← K[LB]
```

```
  Repeat While FLAG = true
```

```
    I ← I+1
```

```
    Repeat While K[I] < KEY
```

```
      I ← I + 1
```

```
    J ← J - 1
```

```
    Repeat While K[J] > KEY
```

```
      J ← J - 1
```

```
    IF I < J
```

```
      Then K[I] ←---→ K[J]
```

```
    Else FLAG ← FALSE
```

```
K[LB] ←---→ K[J]
```

LB			UB						
0	1	2	3	4	5	6	7	8	9
11	23	36	42	65	58	94	74	99	87

11	23	36
↑		↑
I		J

		LB	UB						
11	23	36	42	65	58	94	74	99	87

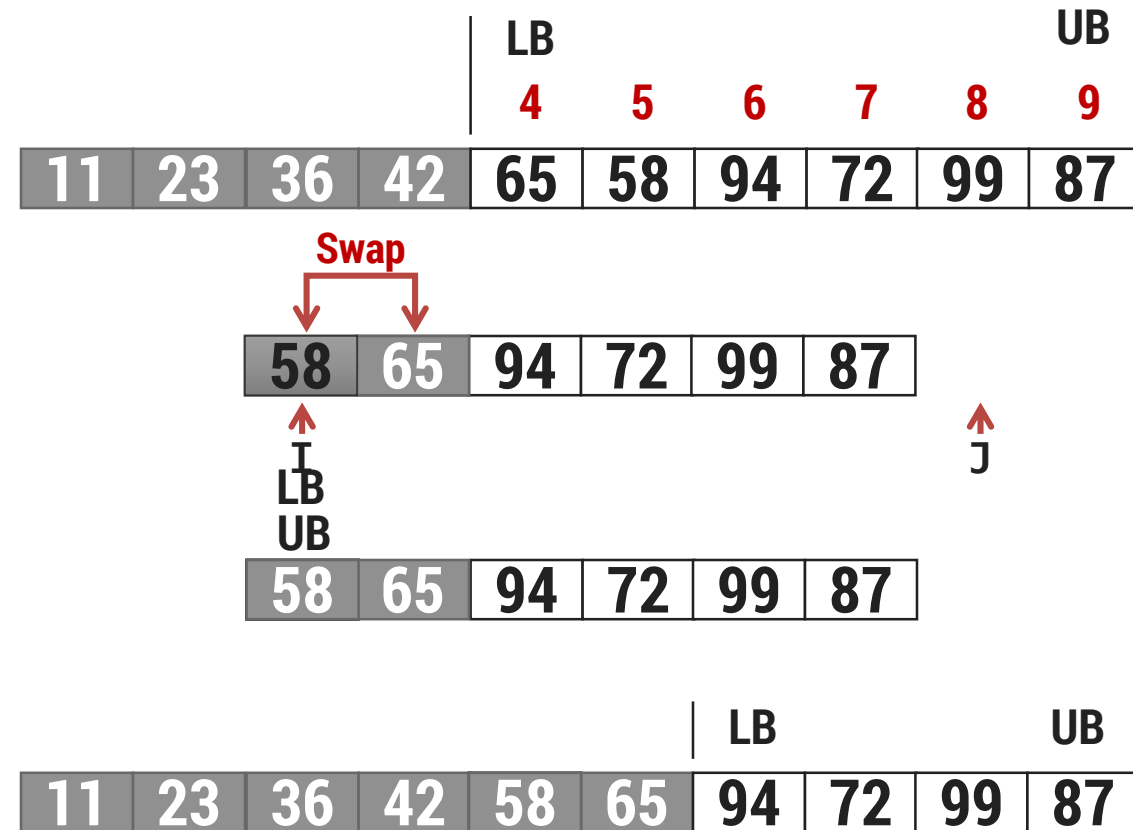
23	36
↑	↑
I	J

			LB	UB					
11	23	36	42	65	58	94	74	99	87

# Quick Sort

```
FLAG ← true
IF LB < UB
Then
  I ← LB
  J ← UB + 1
  KEY ← K[LB]
  Repeat While FLAG = true
    I ← I + 1
    Repeat While K[I] < KEY
      I ← I + 1
    J ← J - 1
    Repeat While K[J] > KEY
      J ← J - 1
    IF I < J
      Then K[I] ←---→ K[J]
    Else FLAG ← FALSE

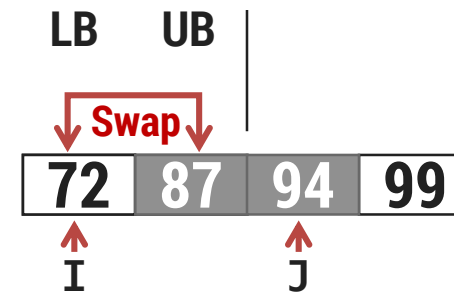
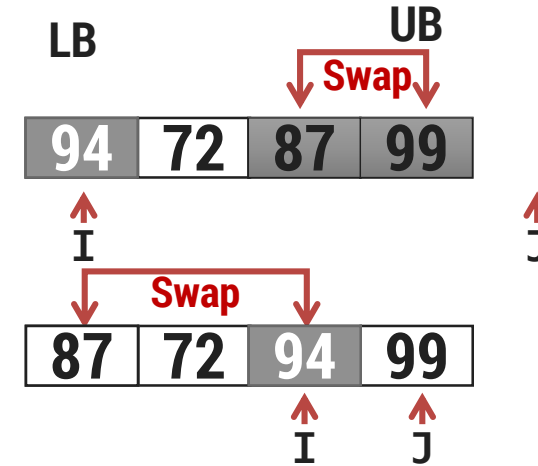
  K[LB] ←---→ K[J]
```



# Quick Sort

```
FLAG ← true
IF LB < UB
Then
  I ← LB
  J ← UB + 1
  KEY ← K[LB]
  Repeat While FLAG = true
    I ← I + 1
    Repeat While K[I] < KEY
      I ← I + 1
    J ← J - 1
    Repeat While K[J] > KEY
      J ← J - 1
    IF I < J
    Then K[I] ←---→ K[J]
    Else FLAG ← FALSE

  K[LB] ←---→ K[J]
```



# Algorithm: QUICK\_SORT(K, LB, UB)

```
1. [Initialize]
   FLAG ← true
2. [Perform Sort]
   IF LB < UB
   Then I ← LB
       J ← UB + 1
       KEY ← K[LB]
       Repeat While FLAG = true
           I ← I+1
           Repeat While K[I] < KEY
               I ← I + 1
           J ← J - 1
           Repeat While K[J] > KEY
               J ← J - 1
           IF I < J
           Then K[I] ←---→ K[J]
           Else FLAG ← FALSE

       K[LB] ←---→ K[J]
```

```
CALL QUICK_SORT(K, LB, J-1)
CALL QUICK_SORT(K, J+1, UB)
```

```
CALL QUICK_SORT(K, LB, J-1)
```

```
3. [Finished]
   Return
```



# Insertion Sort

In insertion sort, **every iteration moves** an **element** from **unsorted portion** to **sorted portion** until all the elements are sorted in the list.

## Steps for Insertion Sort

1

Assume that **first element** in the list is in **sorted portion** of the list and **remaining all elements** are in **unsorted portion**.

2

Select **first element** from the **unsorted list** and **insert** that element **into the sorted list** in **order specified**.

3

**Repeat** the above process **until all** the **elements** from the **unsorted list** are **moved into** the **sorted list**.

This algorithm is not suitable for large data sets

# Insertion Sort cont.

## Complexity of the Insertion Sort Algorithm

To sort a **unsorted list** with 'n' number of **elements** we need to make  $(1+2+3+\dots+n-1) = (n(n-1))/2$  number of **comparisons** in the worst case.

If the list **already sorted**, then it requires 'n' number of **comparisons**.

- Worst Case :  $\Theta(n^2)$
- Best Case :  $\Omega(n)$
- Average Case :  $\Theta(n^2)$

# Insertion Sort Example

Sort given array using Insertion Sort

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

Pass - 1 : Select First Record and considered as Sorter Sub-array

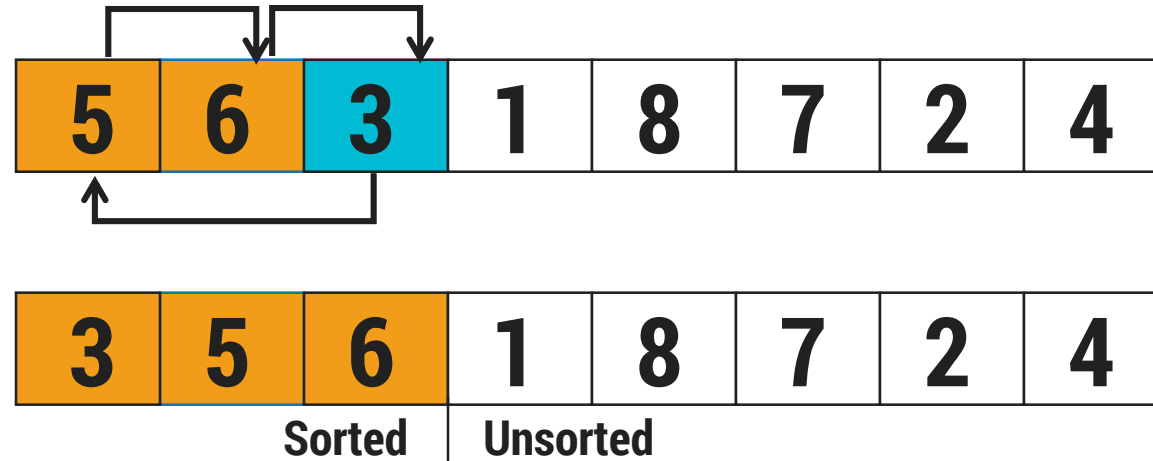
6	5	3	1	8	7	2	4
Sorted	Unsorted						

Pass - 2 : Select Second Record and Insert at proper place in sorted array

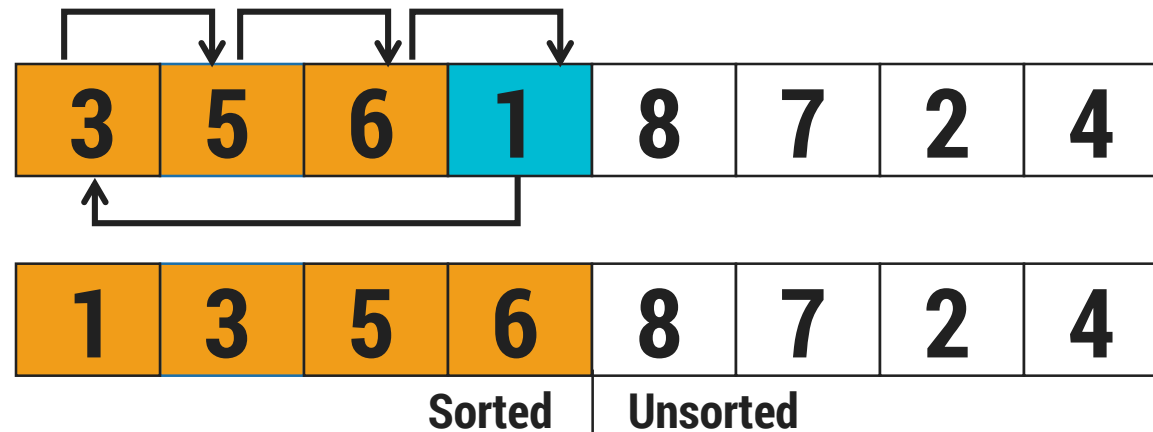
6	5	3	1	8	7	2	4
5	6	3	1	8	7	2	4
Sorted		Unsorted					

# Insertion Sort Example Cont.

Pass - 3 : Select Third record and Insert at proper place in sorted array



Pass - 4 : Select Forth record and Insert at proper place in sorted array



# Insertion Sort Example Cont.

**Pass - 5 : Select Fifth record and Insert at proper place in sorted array**

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

8 is at proper position

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

Sorted

Unsorted

---

**Pass - 6 : Select Sixth Record and Insert at proper place in sorted array**

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



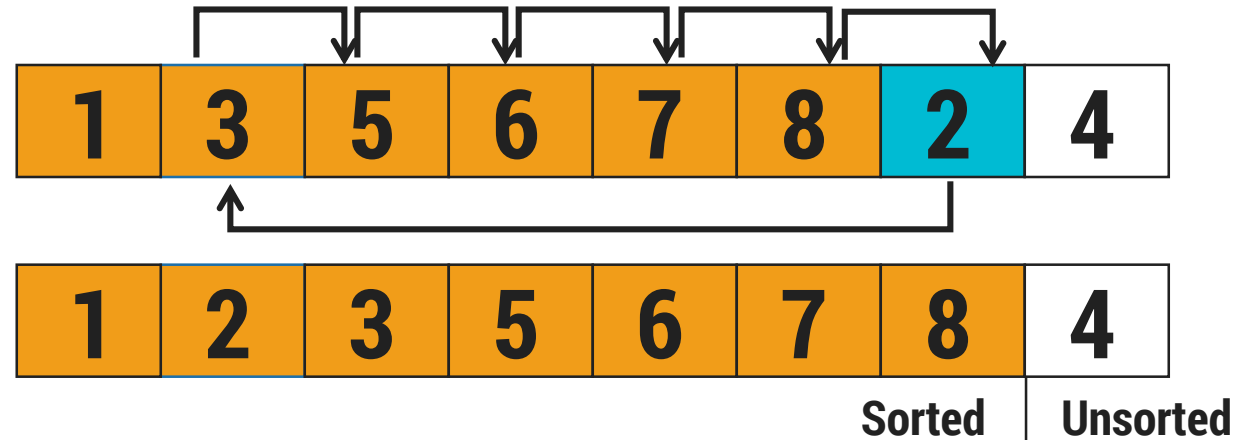
1	3	5	6	7	8	2	4
---	---	---	---	---	---	---	---

Sorted

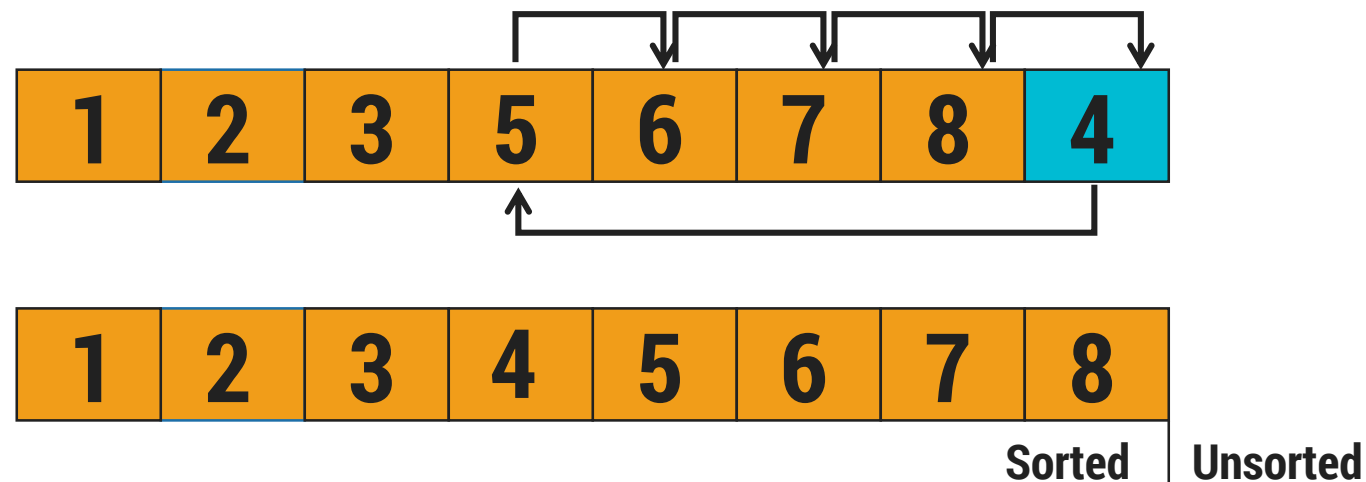
Unsorted

# Insertion Sort Example Cont.

**Pass - 7 : Select Seventh record and Insert at proper place in sorted array**



**Pass - 8 : Select Eighth Record and Insert at proper place in sorted array**



# Insertion sort

- ▶ **Step 1** - If the element is the first element, assume that it is already sorted. Return 1.
- ▶ **Step2** - Pick the next element, and store it separately in a **key**.
- ▶ **Step3** - Now, compare the **key** with all elements in the sorted array.
- ▶ **Step 4** - If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.
- ▶ **Step 5** - Insert the value.
- ▶ **Step 6** - Repeat until the array is sorted.

```
void insertion_sort ( int A[ ] , int n)
{
    for( int i = 0 ;i < n ; i++ )
    { /*storing current element whose left side is checked for its correct position .*/
        int temp = A[ i ];
        int j = i;
        /* check whether the adjacent element in left side is greater or less than the current element. */
        while( j > 0 && temp < A[ j -1])
        { // moving the left side element to one position forward.
            A[ j ] = A[ j-1];
            j= j - 1;
        }
        // moving current element to its correct position.
        A[ j ] = temp;
    }
}
```