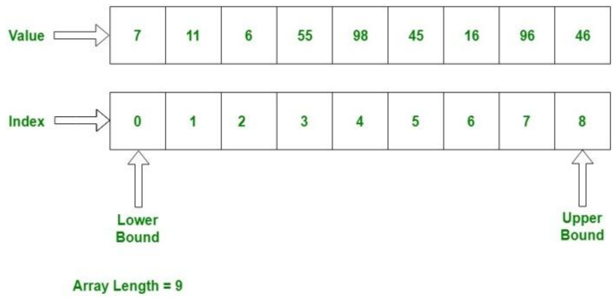
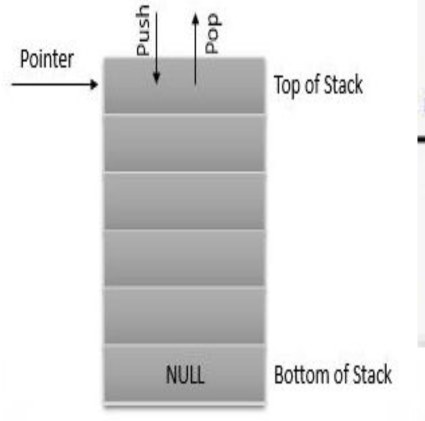
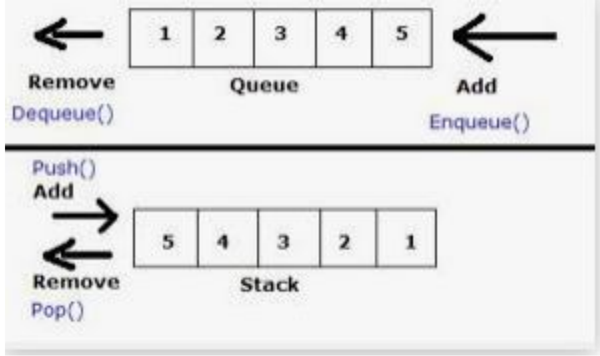
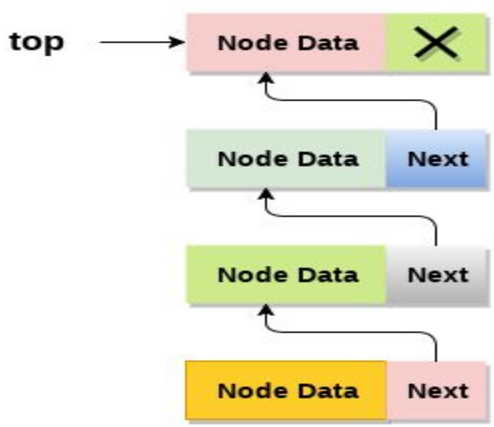
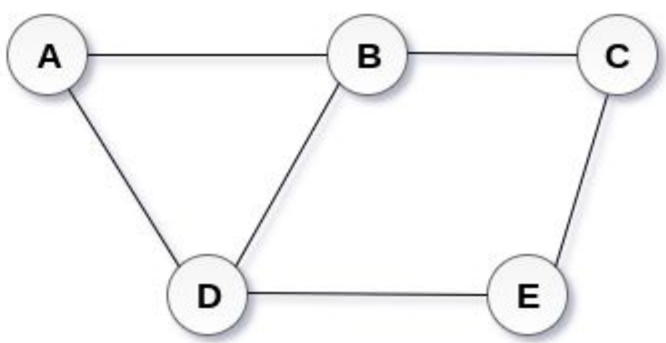


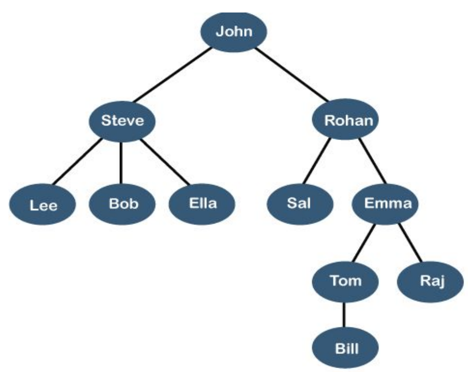




Sorting and Searching Methods







Outline

- Linear Search
- Binary Search
- Sorting Methods
  - Internal and External Sorting
  - Bubble Sort
  - Quick Sort
  - Merge Sort
  - Insertion Sort

Search

- **Search** : Locating of a particular element in a data structure.
- **Internal Search** : Searches in which entire table is constantly in main memory are called internal searches.
- **External Search** : Searches in which most of the table is kept in auxiliary storage are called external searches.
- **Storage** : Tables or a file may be organized into an array of records, a linked list, a tree, or even a graph.
  - Different search techniques may be suitable for different table organizations, a table is often designed with a specific search techniques.
  - The table may be contained completely in memory, completely in auxiliary storage, or it may be divided between the two.

Search

- **Search Algorithm**: It is an algorithm that accepts an argument **a** and tries to find a record whose key is **a**.
  - The algorithm may return the entire record or more commonly it may return a pointer to that record.
  - It is possible that the search for a particular argument in a table is **unsuccessful**; that is; there is no record in the table with argument as its key.
  - A successful search is often called a **retrieval**.
  - A table of record in which a key is used for retrieval is often called a **search table** or a **dictionary**.
- While inserting a record with primary key into a table it is desirable to see whether that record exists or not. This could be done by searching.

Search

- **Different Searching Methods**:
  - Sequential Search / Linear Search
  - Binary Search
  - Indexed Sequential Search
  - Interpolation Search
  - Tree Searching
  - Binary Search Tree.

Sequential Search / Linear 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 / Linear Search Algorithm

```
# 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
```

Sequential Search Example

Search for 1 in given array

2	9	3	1	8

Comparing value of i^{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=1

2	9	3	1	8
i				

Step 1: i=2

2	9	3	1	8
	i			

Step 1: i=3

2	9	3	1	8
		i		

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

Efficiency of Sequential Search / Linear Search

- Searching a table of constant size n
- The number of comparison depends on where the records with the argument key appears in the table.
- If the record is the first one in the table only one comparison is performed.
- If the record is last one in the table, n comparisons are necessary.
- If it is equally likely for the argument to appear at any given table position, a successful search will take (on the average) (n+1)/2 comparisons is O(n).

Sequential Search / Linear Search Program

```
#include <stdio.h>
int search(int array[], int n, int x) {
  // Going through array sequentially
  for (int i = 0; i < n; i++)
    if (array[i] == x)
      return i;
  return -1;
}

int main() {
  int array[] = {2, 4, 0, 1, 9};
  int x = 1; // int x=10;
  int n = sizeof(array) / sizeof(array[0]);
  printf("Original data\n");
  for(int i=0; i< n; i++)
    printf("%d\t",array[i]);
  printf("\n");
  int result = search(array, n, x);
  (result == -1) ? printf("Element %d not found",x) : printf("Element %d found at index:
      %d",x, result);
}
```

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 = [left + right]/2
  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 = (left + 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 = (left + 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 = (left + right) /2 = (2+3)/2 **= 2**

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

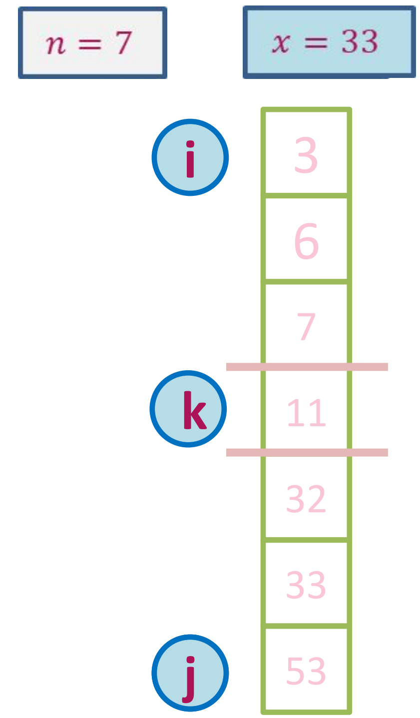
Key=6 is **equals to** middle element = **6**, so **element found**

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

Element Found

Binary Search Example

```
Algorithm: Function biniter(T[1,…,n], x)
    if x > T[n] then return n+1
    i ← 1;
    j ← n;
    while i < j do
        k ← (i + j ) ÷ 2
        if  x ≤ T [k]  then  j ← k-1
        else i ← k + 1
    return i
```



Binary Search Program [using iteration]

```
#include <stdio.h>
 //function to perform binary search
int binarySearch(int array[], int x, int low, int high) {
  // Repeat until the pointers low and high meet each other
  while (low <= high) {
    int mid = low + high  / 2;

    if (array[mid] == x)
      return mid;

    if (array[mid] < x)
      low = mid + 1;

    else
      high = mid - 1;
  }

  return -1;
}
```

Binary Search Program [using iteration]

```
int main(void) {
  int array[] = {3, 4, 5, 6, 7, 8, 9};
  int n = sizeof(array) / sizeof(array[0]);
  printf("\nOriginal Data\n");
  for(int i=0; i< n; i++)
    printf("%d\t",array[i]);
  int x = 4; // int x=2;
  int result = binarySearch(array, x, 0, n - 1);
  if (result == -1)
    printf("\nElement %d Not found",x);
  else
    printf("\nElement %d is found at index %d",x, result);
  return 0;
}
```

Binary Search Program [using recursive]

```
#include <stdio.h>
 //function to perform binary search
int binarySearch(int array[], int x, int low, int high) {
  if (high >= low) {
    int mid = low + high  / 2;

    // If found at mid, then return it
    if (array[mid] == x)
      return mid;

    // Search the left half
    if (array[mid] > x)
      return binarySearch(array, x, low, mid - 1);

    // Search the right half
    return binarySearch(array, x, mid + 1, high);
  }

  return -1;
}
```

Binary Search Program [using recursive]

```
int main(void) {
  int array[] = {3, 4, 5, 6, 7, 8, 9};
  int n = sizeof(array) / sizeof(array[0]);
  int x = 2; // int x = 4;
  printf("\nOriginal Data\n");
  for(int i=0; i< n; i++)
    printf("%d\t",array[i]);
   int result = binarySearch(array, x, 0, n - 1);
  if (result == -1)
    printf("\nElement %d Not found",x);
  else
    printf("\nElement %d is found at index %d",x, result);
}
```

Sorting

- Sorting is the operation of arranging the records of a table into some sequential order according to ordering criteria.
- The term sorting means arranging the elements of the array so that they are placed in some relevant order which may either be ascending order or descending order. That is, if A is an array then the elements of A are arranged in sorted order (ascending order) in such a way that, A[0] < A[1] < A[2] < …… < A[N]
- For example, if we have an array that is declared and initialized as, int A[] = {21, 34, 11, 9, 1, 0, 22};
- Then the sorted array (ascending order) can be given as, A[] = {0, 1, 9, 11, 21, 22, 34}

Sorting

- A sorting algorithm is defined as an algorithm that puts elements of a list in a certain order (that can either be numerical order, lexicographical order or any user-defined order).
- Efficient sorting algorithms are widely used to optimize the use of other algorithms like search and merge algorithms which require sorted lists to work correctly.
- There are two types of sorting:
  - ***Internal sorting*** which deals with sorting the data stored in computer's memory
  - ***External sorting*** which deals with sorting the data stored in files. External sorting is applied when there is voluminous data that cannot be stored in computer's memory.
- **Sorting** arranges data in a sequence which makes searching easier.

Bubble Sort

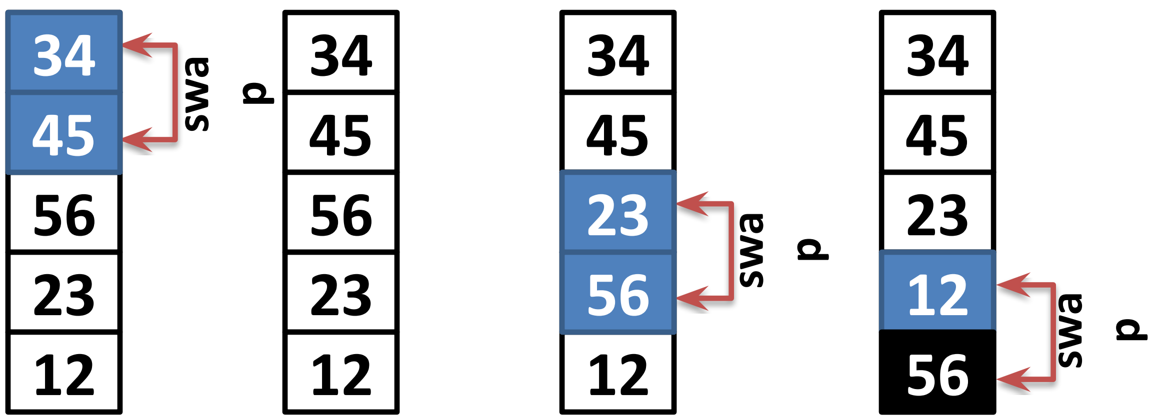
- Here two records are interchanged immediately upon discovering that they are out of order
- During the **first pass R_1 and R_2 are compared** and **interchanged in case of out of order**, this process is repeated for records R_2 and R_3, and so on.
- This method will cause records with **small key to move "bubble up"**
- **After the first pass**, the record with **largest key** will be in the n^{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 approached required at most n–1 passes, The complexity of bubble sort is $O(n^2)$

Bubble Sort

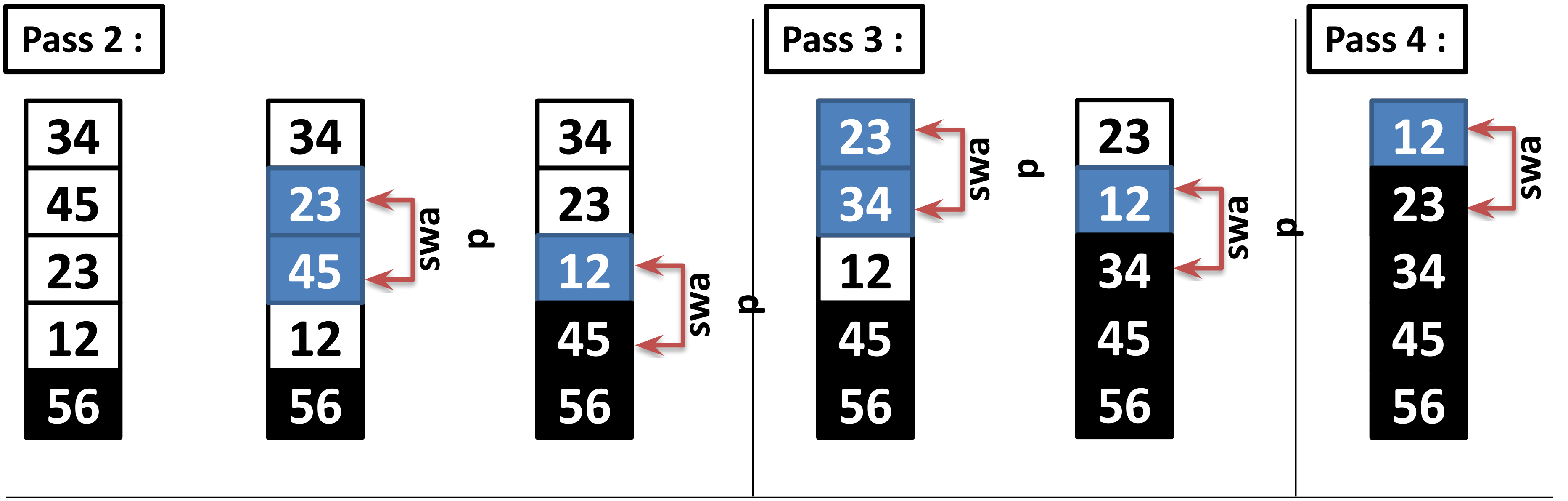
Unsorted Array

45	34	56	23	12

Pass 1 :



Bubble Sort



Pass 2 :
34
45
23
12
56
34
23
45
12
56
swa
p
34
23
12
45
56
swa
p
Pass 3 :
23
34
12
45
56
swa
p
23
12
34
45
56
swa
p
Pass 4 :
12
23
34
45
56
swa
p

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 □ 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 □ 0
4. [Perform pairwise comparisons on unsorted elements]
   Repeat for I = 1, 2, ……….., LAST – 1
      IF   K[I] > K [I+1]
      Then    K[I] □□ K[I+1]
       EXCHS □ EXCHS + 1
5. [Any exchange made in this pass?]
   IF    EXCHS = 0
   Then  Return (Vector is sorted, early return)
   ELSE  LAST □ LAST - 1
6. [Finished]
   Return
```

Bubble Sort Program

```
#include <stdio.h>
int main()
{
	int array[100], n, i, j, swap;
	printf("Enter number of elements : ");
	scanf("%d", &n);
	printf("Enter %d Numbers:\n", n);
	for(i = 0; i < n; i++)
		scanf("%d", &array[i]);
	for(i = 0 ; i < n - 1; i++)
		for(j = 0 ; j < n-i-1; j++)
			if(array[j] > array[j+1])
			{
				swap=array[j];
				array[j]=array[j+1];
				array[j+1]=swap;
			}
	printf("Sorted Array:\n");
	for(i = 0; i < n; i++)
		printf("%d\t", array[i]);
	return 0;
}
```

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

Complexity of the Insertion Sort Algorithm

- **To sort a unsorted list** with **'n'** number of **elements** we **need** to make **(1+2+3+......+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

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

5	6	3	1	8	7	2	4

3	5	6	1	8	7	2	4

Sorted | Unsorted

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

3	5	6	1	8	7	2	4

1	3	5	6	8	7	2	4

Sorted | Unsorted

Insertion Sort Example

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

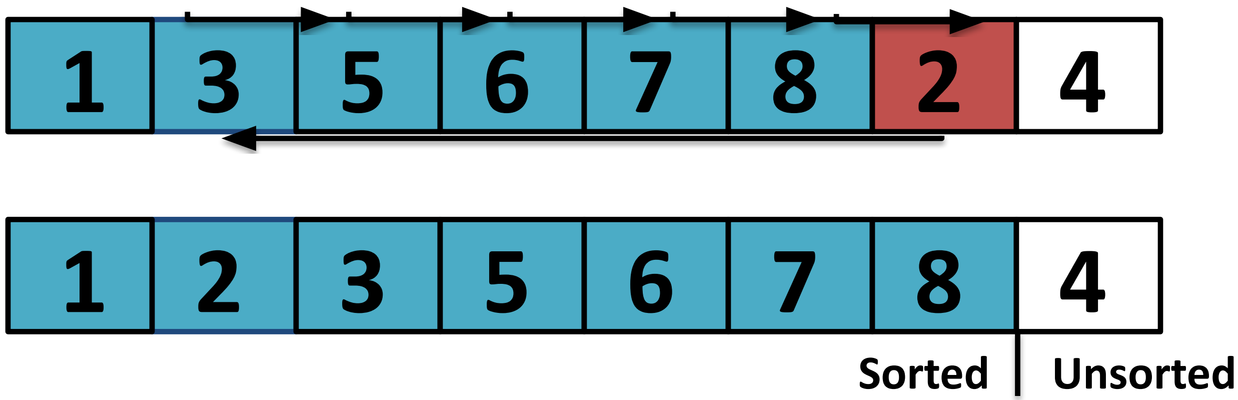
1	3	5	6	8	7	2	4

1	3	5	6	7	8	2	4

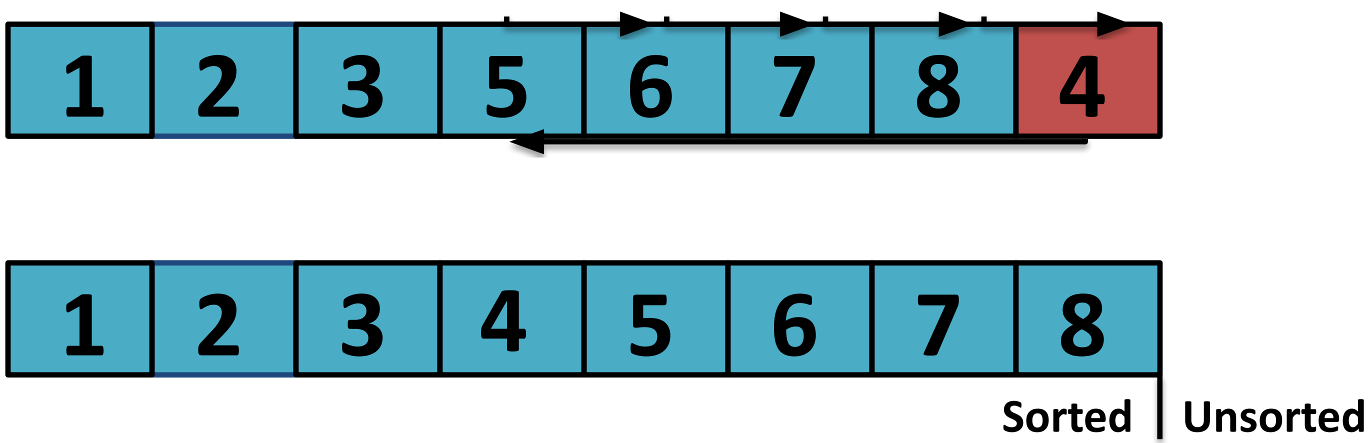
Sorted | Unsorted

Insertion Sort Example

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 Algorithm

- **Insertion sort (ARR, N)** where ARR is an array of N elements

Step 1: Repeat Steps 2 to 5 for K = 1 to N
Step 2: SET TEMP = ARR[K]
Step 3: SET J = K – 1
Step 4: Repeat while TEMP <= ARR[J]
SET ARR[J + 1] = ARR[J]
SET J = J – 1
[END OF INNER LOOP]
Step 5: SET ARR[J + 1] = TEMP
[END OF LOOP]
Step 6: EXIT

Insertion Sort Program

```
#include<stdio.h>
void main ()
{
   int i,j, k,temp;
   int a[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
   printf("\nPrinting sorted elements...\n");
   for(k=1; k<10; k++)
   {
      temp = a[k];
      j= k-1;
      while(j>=0 && temp <= a[j])  // Compare key with each element on the left of it until an element smaller than  it is found.
      {
         a[j+1] = a[j];
         j = j-1;
      }
      a[j+1] = temp;

   }
   for(i=0;i<10;i++)
   {
      printf("\n%d\n",a[i]);
   }
```

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 list 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)

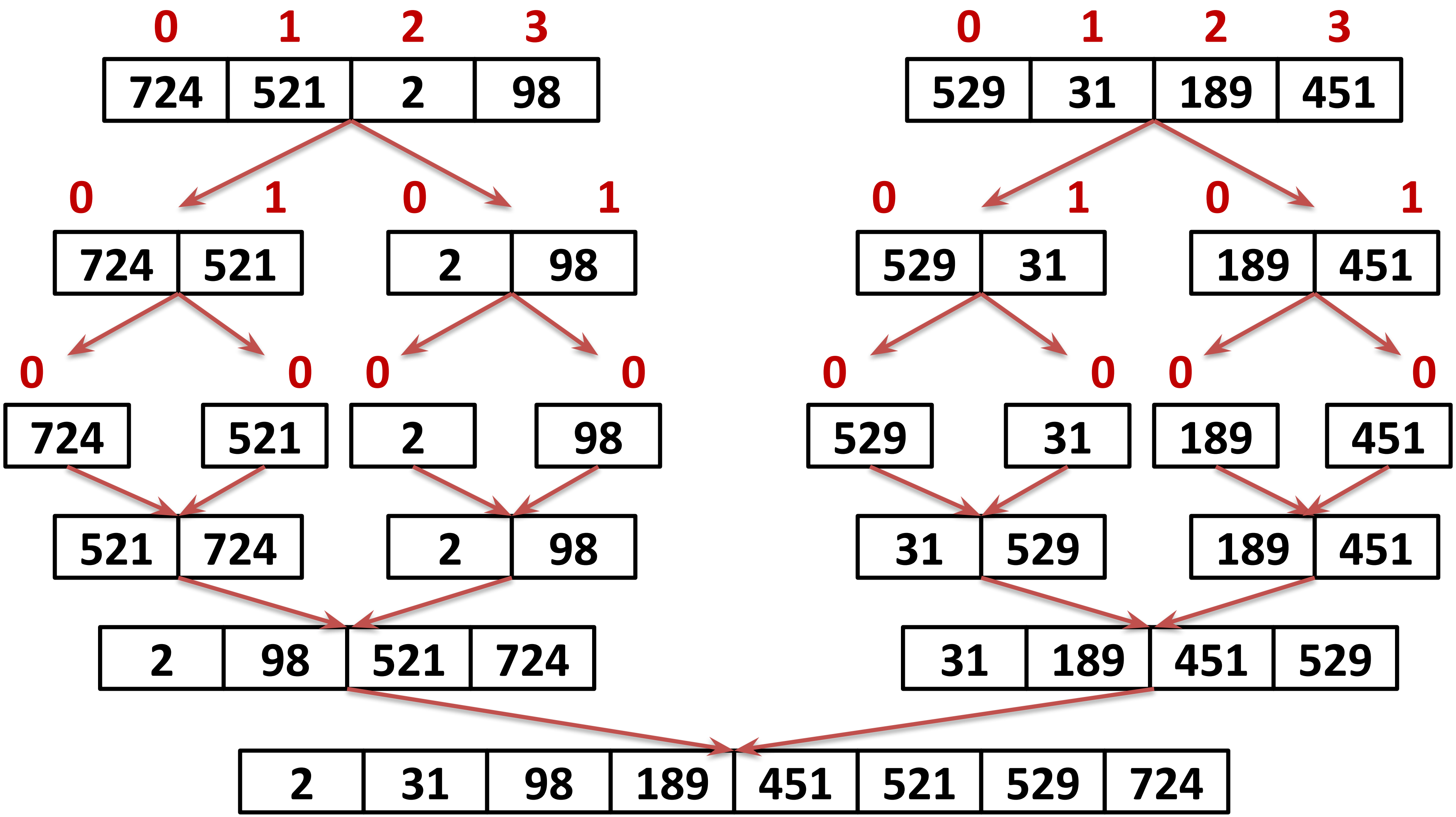
0	1	2	3	4	5	6	7
724	521	2	98	529	31	189	451

724	521	2	98
0	1	2	3

529	31	189	451
0	1	2	3

Merge Sort

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



Merge Sort Algorithm

Mergesort(Passed an array)

if array size > 1

Divide array in half

Call Mergesort on first half.

Call Mergesort on second half.

Merge two halves.

Merge(Passed two arrays)

Compare leading element in each array

Select lower and place in new array.

(If one input array is empty then place remainder of other array in output array)

Merge Sort Program

```
#include<stdio.h>

void mergesort(int a[],int i,int j);
void merge(int a[],int i1,int j1,int i2,int j2);

int main()
{
	int a[30],n,i;
	printf("Enter no of elements:");
	scanf("%d",&n);
	printf("Enter array elements:");
	for(i=0;i<n;i++)
		scanf("%d",&a[i]);
	mergesort(a,0,n-1);
	printf("\nSorted array is :");
	for(i=0;i<n;i++)
		printf("%d ",a[i]);
	return 0;
}
```

Merge Sort Program

```
void mergesort(int a[],int i,int j)
{
	int mid;
	if(i<j)
	{
		mid=(i+j)/2;
		mergesort(a,i,mid); //left recursion
		mergesort(a,mid+1,j); //right recursion
		merge(a,i,mid,mid+1,j); //merging of two sorted sub-arrays
	}
}
```

Merge Sort Program

```
void merge(int a[],int i1,int j1,int i2,int j2)
{
	int temp[50]; //array used for merging
	int i,j,k;
	i=i1; //beginning of the first list
	j=i2; //beginning of the second list
	k=0;
	while(i<=j1 && j<=j2) //while elements in both lists
	{
		if(a[i]<a[j])
			temp[k++]=a[i++];
		else
			temp[k++]=a[j++];
	}
	while(i<=j1) //copy remaining elements of the first list
		temp[k++]=a[i++];
	while(j<=j2) //copy remaining elements of the second list
		temp[k++]=a[j++];
	//Transfer elements from temp[] back to a[]
	for(i=i1,j=0;i<=j2;i++,j++)
		a[i]=temp[j];
}
```

Merge Sort Program Output

Enter no of elements:7

Enter array elements:38

27

43

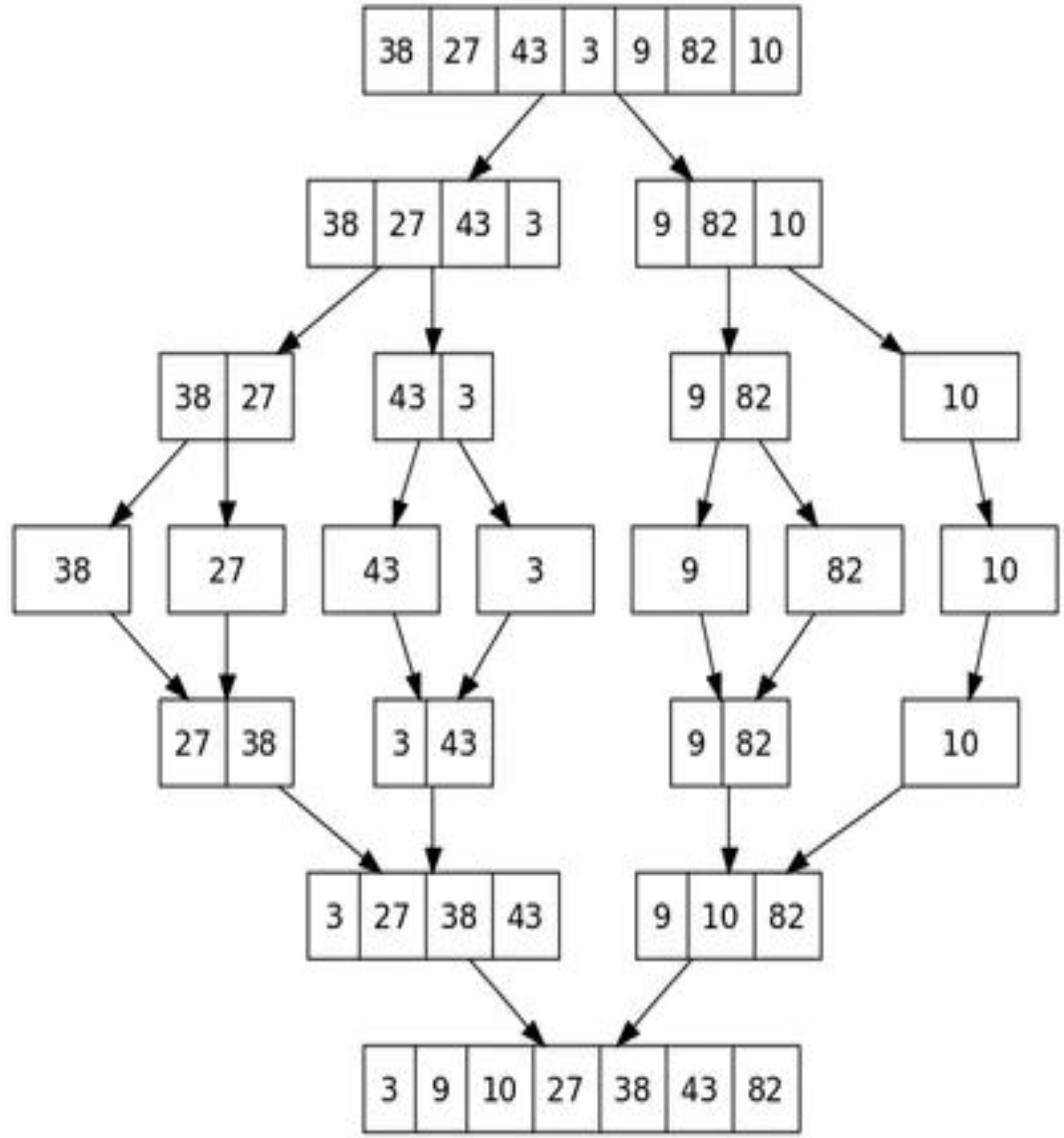
3

9

82

10

Sorted array is :3 9 10 27 38 43 82

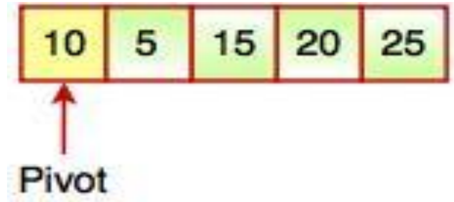


Quick Sort /Partition Exchange Sort

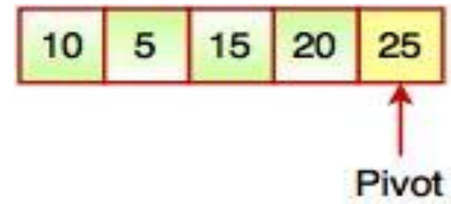
- **Quick sort** is a highly efficient sorting algorithm and is based on **partitioning of array** of data into **smaller arrays**.
- Quick Sort is **divide and conquer** algorithm.
- At each step of the method, the goal is to place a particular record in its final position within the table.
- In doing 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

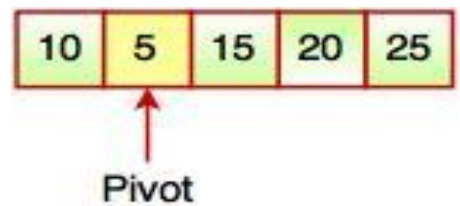
- There are many **different versions** of Quick Sort **that pick pivot** in different ways.
  - Always pick **first element as pivot**. (in our case we have consider this version).



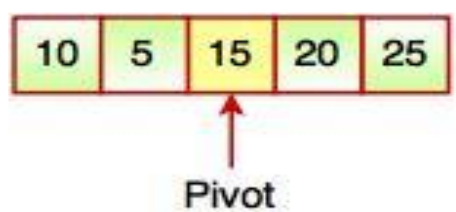
  - Always pick **last element as pivot**



  - Pick a **random element as pivot**.



  - Pick **median as pivot**.



Quick Sort

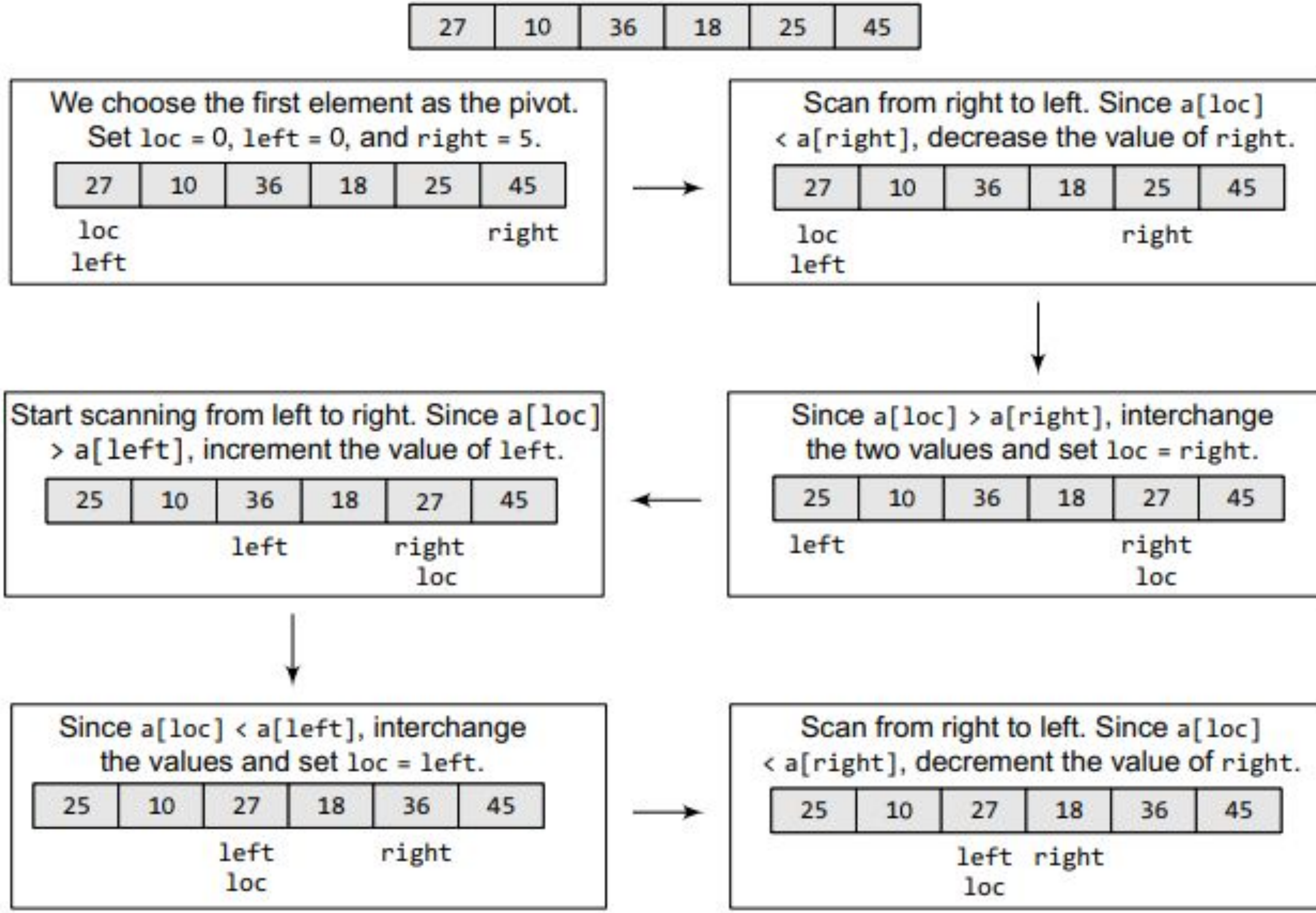
- Quick sort partitions an array and then calls itself recursively twice to sort the two resulting sub arrays.
- This algorithm is quite **efficient for large-sized data sets**
- Its average and **worst case complexity** are of **$O(n^2)$**, where n is the number of items.

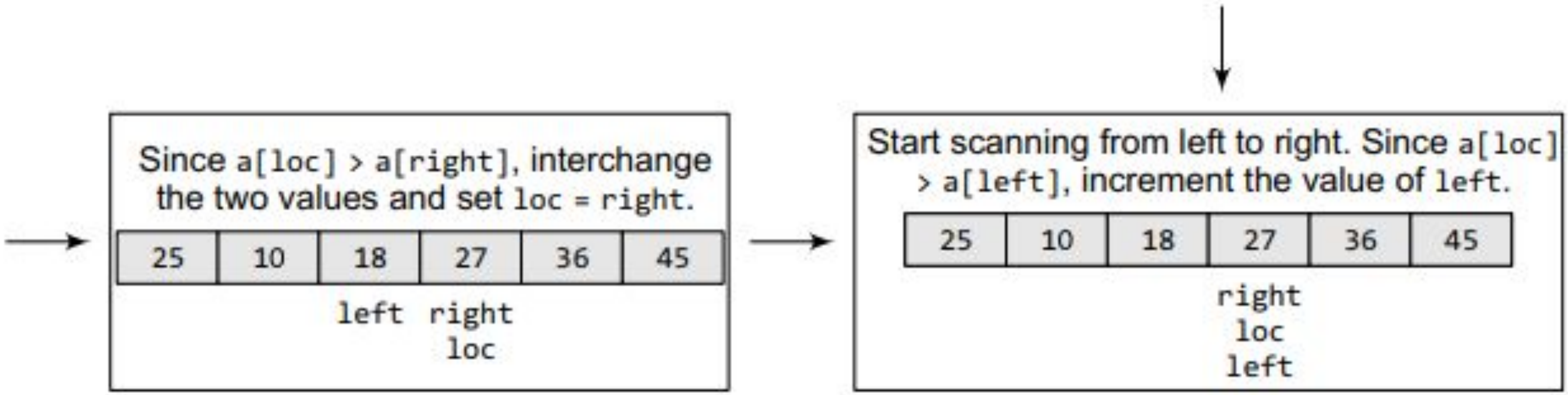
Quick Sort Step by Step Process

- In Quick sort algorithm, partitioning of the list is performed using following steps...
  1. Consider the first element of the list as **pivot** (i.e., Element at first position in the list).
  2. Define two variables i and j. Set i and j to first and last elements of the list respectively.
  3. Increment i until list[i] > pivot then stop.
  4. Decrement j until list[j] < pivot then stop.
  5. If i < j then exchange list[i] and list[j].
  6. Repeat steps 3,4 & 5 until i > j.
  7. Exchange the pivot element with list[j] element.

```
PARTITION (ARR, BEG, END, LOC)

Step 1: [INITIALIZE] SET LEFT = BEG, RIGHT = END, LOC = BEG, FLAG = 0
Step 2: Repeat Steps 3 to 6 while FLAG = 0
Step 3: Repeat while ARR[LOC] <= ARR[RIGHT] AND LOC != RIGHT
                SET RIGHT = RIGHT - 1
        [END OF LOOP]
Step 4: IF LOC = RIGHT
                SET FLAG = 1
        ELSE IF ARR[LOC] > ARR[RIGHT]
                SWAP ARR[LOC] with  ARR[RIGHT]
                SET LOC = RIGHT
        [END OF IF]
Step 5: IF FLAG = 0
                Repeat while ARR[LOC] >= ARR[LEFT] AND LOC != LEFT
                SET LEFT = LEFT + 1
                [END OF LOOP]
Step 6:         IF LOC = LEFT
                        SET FLAG = 1
                ELSE IF ARR[LOC] < ARR[LEFT]
                        SWAP ARR[LOC] with  ARR[LEFT]
                        SET LOC = LEFT
                [END OF IF]
        [END OF IF]
Step 7: [END OF LOOP]
Step 8: END
```

```
QUICK_SORT (ARR, BEG, END)

Step 1: IF (BEG < END)
            CALL PARTITION (ARR, BEG, END, LOC)
            CALL QUICKSORT(ARR, BEG, LOC - 1)
            CALL QUICKSORT(ARR, LOC + 1, END)
        [END OF IF]
Step 2: END
```


27 10 36 18 25 45
We choose the first element as the pivot. Set loc = 0, left = 0, and right = 5.
27 10 36 18 25 45
loc
left
right
Scan from right to left. Since a[loc] < a[right], decrease the value of right.
27 10 36 18 25 45
loc
left
right
Since a[loc] > a[right], interchange the two values and set loc = right.
25 10 36 18 27 45
left
right
loc
Start scanning from left to right. Since a[loc] > a[left], increment the value of left.
25 10 36 18 27 45
left
right
loc
Since a[loc] < a[left], interchange the values and set loc = left.
25 10 27 18 36 45
left
loc
right
Scan from right to left. Since a[loc] < a[right], decrement the value of right.
25 10 27 18 36 45
left
loc
right



Now left = loc, so the procedure terminates, as the pivot element (the first element of the array, that is, 27) is placed in its correct position. All the elements smaller than 27 are placed before it and those greater than 27 are placed after it.

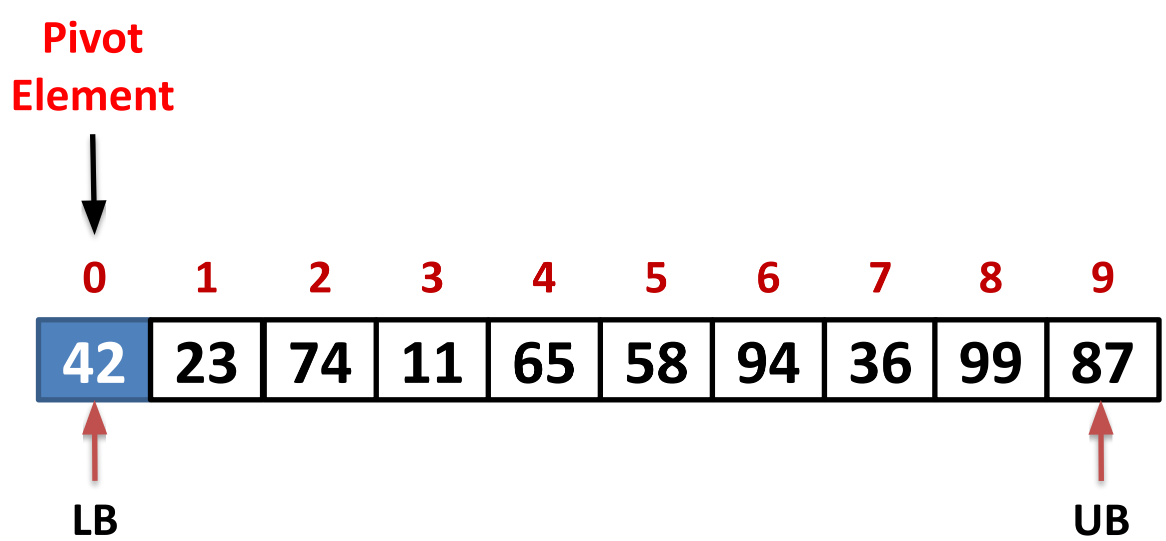
The left sub-array containing 25, 10, 18 and the right sub-array containing 36 and 45 are sorted in the same manner.

Quick Sort

Sort Following Array using Quick Sort Algorithm

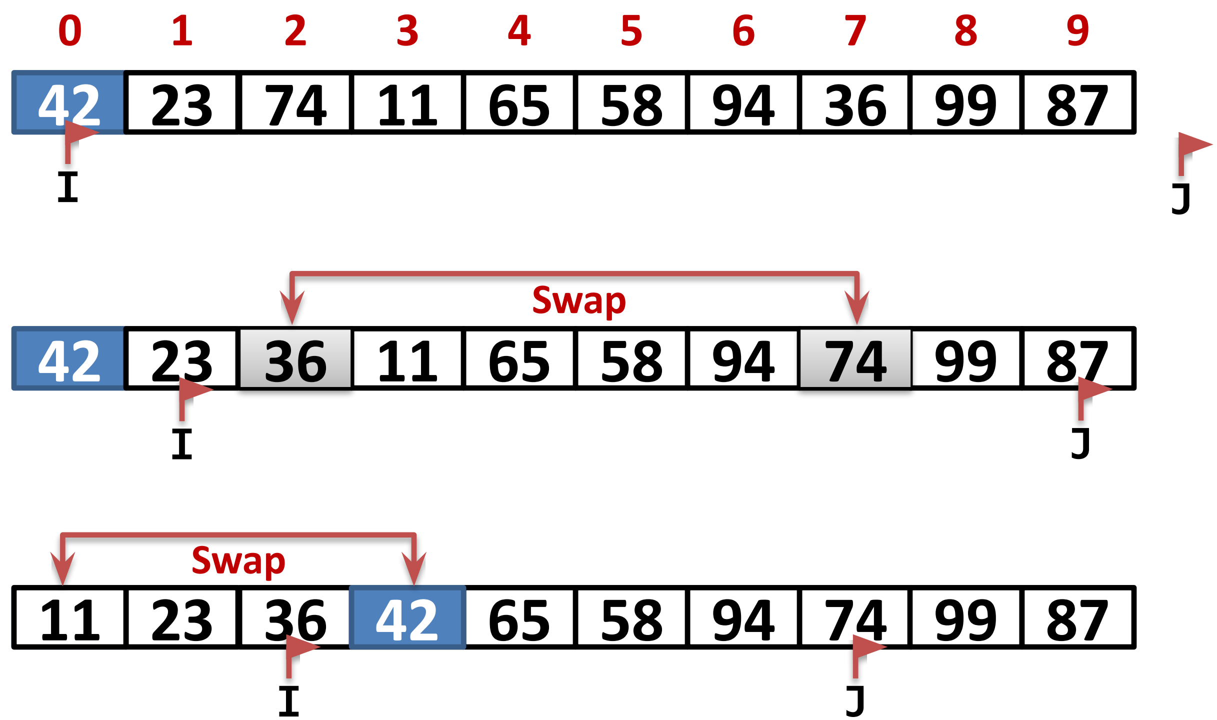
We are considering **first element as pivot element**, so **Lower bound** is **First Index** and **Upper bound** is **Last Index**

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 □ 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 = 0, UB = 9 **KEY = 42**

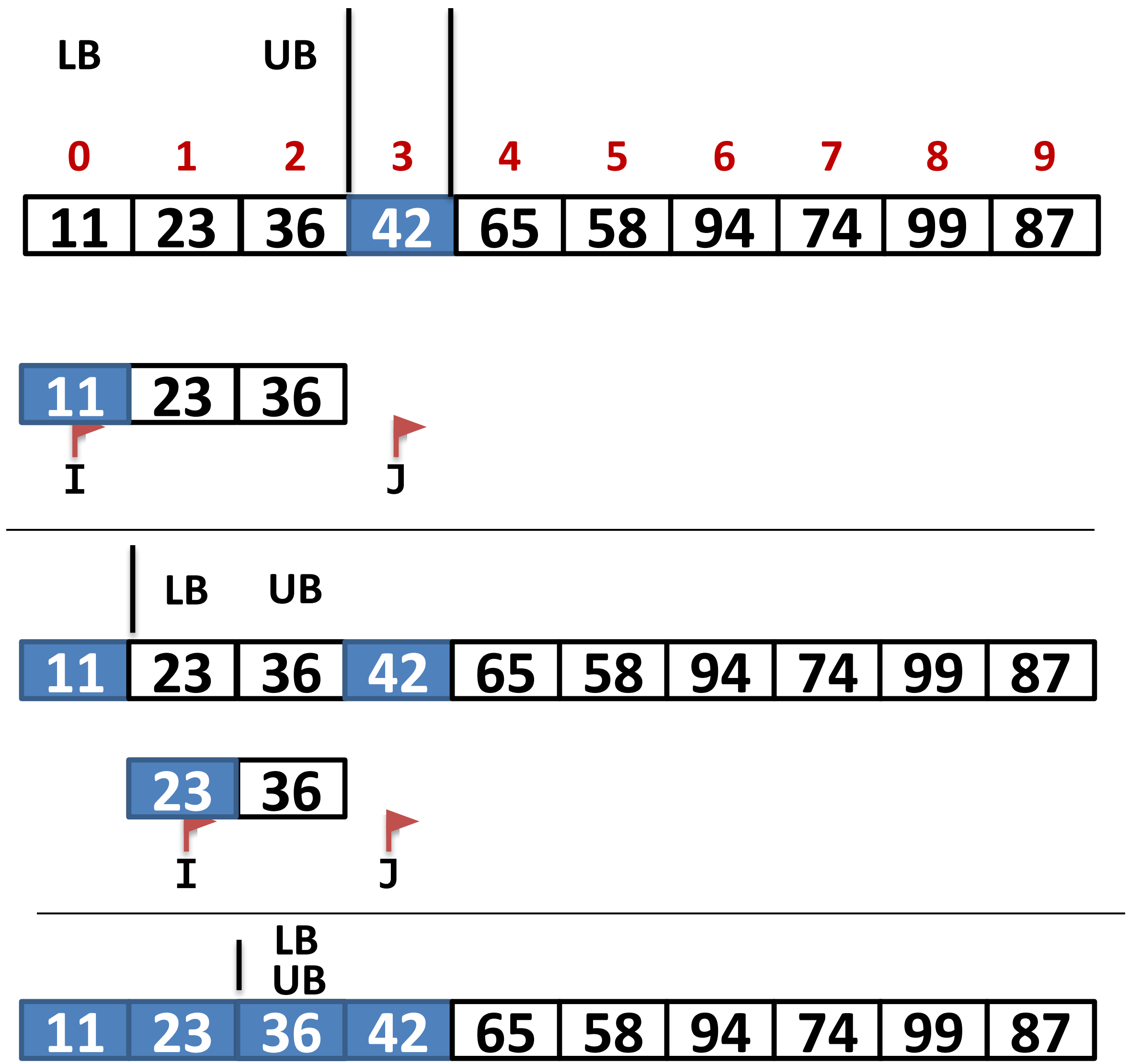
I= 0

J= 10

FLAG= true

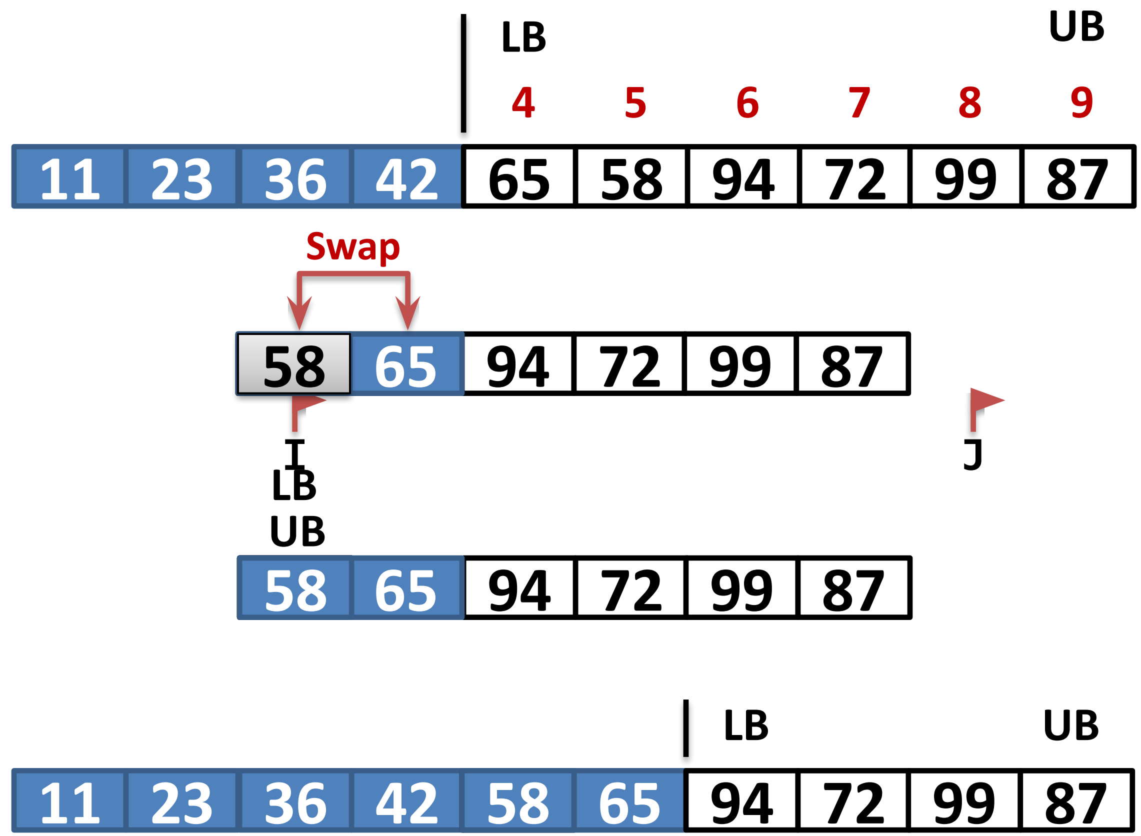
Quick Sort

```
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]
```



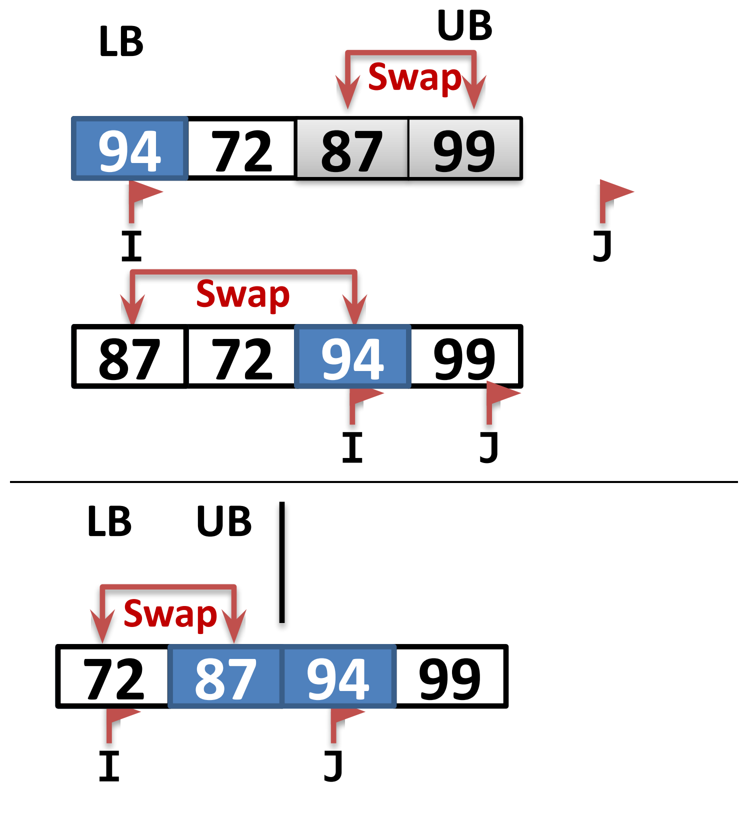
Quick Sort

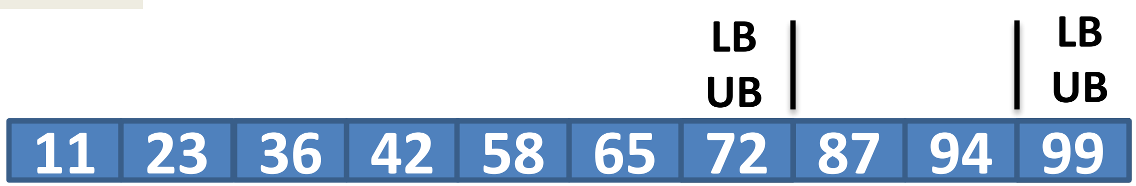
```
FLAG □ true
IF   LB < UB
Then
  I □ LB
  J □ UB + 1
  KEY □ K[LB]
  Repeat While FLAG = true
     I □ I+1
     Repeat While K[I] < KEY
       I □ I + 1
     J □ J – 1
     Repeat While K[J] > KEY
       J □ J – 1
     IF   I<J
     Then K[I] □---□ K[J]
     Else FLAG □ FALSE

   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
```

Quick Sort Program

```
#include<stdio.h>
void quickSort(int [10],int,int);
int size;
void main(){
  int list[20],i;
  printf("Enter size of the list: ");
  scanf("%d",&size);
  printf("Enter %d integer values: ",size);
  for(i = 0; i < size; i++)
      scanf("%d",&list[i]);
   quickSort(list,0,size-1);
   printf("\nList after sorting is: ");
   for(i = 0; i < size; i++)
        printf(" %d",list[i]);
}
```

Quick Sort Program

```
void quickSort(int list[10],int first,int last)
{
    int pivot,i,j,temp,x;
    if(first < last)
    {
        pivot = first;
        printf("\n%d pivot element is \n",list[pivot]);
        i = first;
        j = last;
       while(i < j)
       {
           while(list[i] <= list[pivot] && i < last)
               i++;
          while(list[j] > list[pivot])
            j--;
          if(i <j)
          {
              temp = list[i];
              list[i] = list[j];
              list[j] = temp;
           }
       }
```

Quick Sort Program

```
     temp = list[pivot];
     list[pivot] = list[j];
     list[j] = temp;
     printf("\nintermediate results\n");
    for(i=0;i<size;i++)
        printf("%d  ",list[i]);
    quickSort(list,first,j-1);
    quickSort(list,j+1,last);
  }//end of if
}
```

Comparison of different sorting techniques

Analysis Type	Bubble Sort	Insertion Sort	Merge Sort	Quick Sort
Best Case	$O(n^2)$	$O(n)$	O(log n)	O(log n)
Average Case	$O(n^2)$	$O(n^2)$	O(log n)	O(log n)
Worst Case	$O(n^{2)}$	$O(n^2)$	O(log n)	$O(n^2)$
Space Complexity	O(1)	O(1)	O(n)	O(log n)

When to use which sorting techniques

- When to use bubble sort?
  - Bubble sort algorithm works well with large datasets where the items are almost sorted because it takes only one iteration to detect whether the list is sorted or not.
  - if the list is unsorted to a large extend then this algorithm holds good for collections of less number of elements.
  - Bubble sort is fastest on an extremely small or nearly sorted set of data.

When to use which sorting techniques

- When to use insertion sort?
  - It is not advised to apply insertion sort when elements are in reverse order. In that case insertion sort takes maximum time to sort.
  - When the elements are sorted it takes the minimum time i.e O(N).If the data is almost sorted or when the list is small as it has a complexity of O(N^2).If the list is sorted a minimum number of elements will slide over to insert the element at it's correct location i.e at sorted array part.
  - Algorithm is stable and it has fast running case when the list is nearly sorted.

When to use which sorting techniques

- When to use merge sort?
  - Merge sort is used when the data structure doesn't support random access since it works with pure sequential access that is forward iterators, rather than random access iterators.
  - It is widely used for external sorting, where random access can be very, very expensive compared to sequential access.
  - It is used where it is known that the data is similar data.
  - Merge sort is fast in the case of a linked list because need for random access in linked list is low.

When to use which sorting techniques

- When to use quick sort?
  - Quick sort is fastest, but it is not always O(Nlog N), as it becomes O(N^2) in worst cases.
  - Quicksort is probably more effective for datasets that fit in memory. For larger data sets it proves to be inefficient so algorithms like merge sort are preferred in that case.
  - Quick Sort in is an in-place sort (i.e. it doesn't require any extra storage) so it is appropriate to use it for collections like arrays.

Thank you

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)


Insertion
Selection
Bubble
Shell
Merge
Heap
Quick
Quick3
Random
Nearly Sorted
Reversed
Few Unique


Insertion
Selection
Bubble
Shell
Merge
Heap
Quick
Quick3
Random
Nearly Sorted
Reversed
Few Unique