UNIVERSITY OF ENGINEERING & MANAGEMENT, KOLKATA

Course Name : Data Structure using C
Topic: Sorting

Kaustuv Bhattacharjee

Sorting-Introduction

- Sorting means arranging the elements of an array so that they are placed in either ascending or descending order.
- If A is an array, then the elements of A are arranged in a sorted order (ascending order) in such a way that A[0] < A[1] < A[2] < ...... < A[N].
- A sorting algorithm is defined as an algorithm that puts the elements of a list in a certain order, which can be either 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.

Sorting-Types

- **Two types :**
  - Internal sorting which deals with sorting the data stored in the 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 the memory.

Sorting-Methods

- **Bubble Sort**
- **Insertion Sort**
- **Selection Sort**
- **Quick Sort**
- **Merge Sort**

Bubble Sort

- Most simple method
- Sorts the array elements by repeatedly moving the largest element to the highest index position of the array segment (for ascending order)
- Consecutive adjacent pairs of elements in the array are compared with each other
- If the element at the lower index is greater than the element at the higher index, the two elements are interchanged so that the element is placed before the bigger one
- This process will continue till the list of unsorted elements exhausts

Bubble Sort - Technique

- The basic methodology of the working of bubble sort is given as follows:

  (a) In Pass 1, **A[0]** and **A[1]** are compared, then **A[1]** is compared with **A[2], A[2]** is compared with **A[3],** and so on. Finally, **A[N–2]** is compared with **A[N–1].** Pass 1 involves n–1 comparisons and places the biggest element at the highest index of the array.

  (b) In Pass 2, **A[0]** and **A[1]** are compared, then **A[1]** is compared with **A[2], A[2]** is compared with **A[3]**, and so on. Finally, **A[N–3]** is compared with **A[N–2].** Pass 2 involves n–2 comparisons and places the second biggest element at the second highest index of the array.

  (c) In Pass 3, **A[0]** and **A[1]** are compared, then **A[1]** is compared with **A[2], A[2]** is compared with **A[3],** and so on. Finally, **A[N–4]** is compared with **A[N–3].** Pass 3 involves n–3 comparisons and places the third biggest element at the third highest index of the array.

  (d) In Pass n–1, **A[0]** and **A[1]** are compared so that **A[0]<A[1].** After this step, all the elements of the array are arranged in ascending order.

Bubble Sort - Example

- Consider an array A[] that has the following elements **A[] = {30, 52, 29, 87, 63, 27, 19, 54}**
- **Pass 1:**

  (a) Compare **30** and **52**. Since **30 < 52**, no swapping is done.

  (b) Compare **52** and **29**. Since **52 > 29**, swapping is done.

  30, 29, 52, 87, 63, 27, 19, 54

  (c) Compare **52** and **87**. Since **52 < 87**, no swapping is done.

  (d) Compare **87** and **63**. Since **87 > 63**, swapping is done.

  30, 29, 52, 63, 87, 27, 19, 54

  (e) Compare **87** and **27**. Since **87 > 27**, swapping is done.

  30, 29, 52, 63, 27, 87, 19, 54

  (f) Compare **87** and **19**. Since **87 > 19**, swapping is done.

  30, 29, 52, 63, 27, 19, 87, 54

  (g) Compare **87** and **54.** Since **87 > 54**, swapping is done.

  30, 29, 52, 63, 27, 19, 54, 87

- After end of the first pass, the largest element is placed at the highest index of the array. All the other elements are still unsorted.
- This is continued in next few passes until all the elements are sorted in ascending order.

Bubble Sort - Algorithm

BUBBLE_SORT(A, N)

Step 1: Repeat **Step 2** For I =0 to N-1

Step 2: Repeat For J =0 to N – I - 1

Step 3: IF A[J] > A[J + 1]
SWAP A[J] and A[J+1]
[END OF INNER LOOP]
[END OF OUTER LOOP]

Step 4: EXIT

Bubble Sort - Optimization

- What if the array is already sorted?
- No swapping is done, still to continue n-1 passes unnecessarily.
- Optimize the bubble sort once the array is found to be sorted (using a flag)
- Set a variable flag to TRUE before every pass, change the status to FALSE if any swapping is performed

UNIVERSITY OF ENGINEERING & MANAGEMENT, KOLKATA

Bubble Sort – Optimization Code

```
void bubble_sort(int *arr, int n)
{
        int i, j, temp, flag = 0;                    12,15,23,34,45,56,68,89,90
        for(i=0; i<n; i++)
        {
                flag=0;
                for(j=0; j<n–i–1; j++)
                {
                        if(arr[j]>arr[j+1])
                        {
                                flag = 1;
                                temp = arr[j+1];
                                arr[j+1] = arr[j];
                                arr[j] = temp;
                        }
                }
                if(flag == 0) // array is sorted
                        break;
        }
}
```

Insertion Sort

- This algorithm inserts each item into its proper place in the final list.

- To move the current data element past the already sorted values and repeatedly interchanging it with the preceding value until it is in its correct place.

Insertion Sort-Technique

- Insertion sort works as follows:
- The array of values to be sorted is divided into two sets. One that stores sorted values and another that contains unsorted values.
- The sorting algorithm will proceed until there are elements in the unsorted set.
- Suppose there are **n** elements in the array. Initially, the element with index **0** (assuming **LB = 0**) is in the sorted set. Rest of the elements are in the unsorted set.
- The first element of the unsorted partition has array index **1 (if LB = 0).**
- During each iteration of the algorithm, the first element in the unsorted set is picked up and inserted into the correct position in the sorted set.

Insertion Sort-Explanation

- Sort the values in the array using insertion sort:

	39	9	45	63	18	81	108	54	72	36
Pass 1	39	9	45	63	18	81	108	54	72	36
Pass 2	9	39	45	63	18	81	108	54	72	36
Pass 3	9	39	45	63	18	81	108	54	72	36
Pass 4	9	39	45	63	18	81	108	54	72	36
Pass 5	9	18	39	45	63	81	108	54	72	36
Pass 6	9	18	39	45	63	81	108	54	72	36
Pass 7	9	18	39	45	63	81	108	54	72	36
Pass 8	9	18	39	45	54	63	81	108	72	36
Pass 9	9	18	39	45	54	63	72	81	108	36
Pass 10	9	18	36	39	45	54	63	72	81	108

Unsorted		Sorted	

- **Explanation:**
- Initially, **A[0]** is the only element in the sorted set. In Pass 1, **A[1]** will be placed either before or after **A[0]**, so that the array A is sorted. In Pass 2, **A[2]** will be placed either before **A[0]**, in between **A[0]** and **A[1]**, or after **A[1]**. In Pass 3, **A[3]** will be placed in its proper place. In Pass N–1, **A[N–1]** will be placed in its proper place to keep the array sorted.

Insertion Sort-Algorithm

INSERTION-SORT (ARR, N)

Step 1: Repeat **Steps 2 to 5** for K = 1 to N – 1

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

39	9	45	63	18	81	108	54	72	36

Selection Sort

Technique:

First find the smallest value in the array and place it in the first position. Then, find the second smallest value in the array and place it in the second position. Repeat this procedure until the entire array is sorted. Therefore,

In **Pass 1**, find the position POS of the smallest value in the array and then swap **ARR[POS]** and **ARR[0].** Thus, **ARR[0]** is sorted.

In **Pass 2**, find the position POS of the smallest value in sub-array of N–1 elements. Swap **ARR[POS]** with **ARR[1].** Now, **ARR[0]** and **ARR[1]** is sorted.

In **Pass N–1,** find the position POS of the smaller of the elements **ARR[N–2]** and **ARR[N–1].** Swap **ARR[POS]** and **ARR[N–2]** so that **ARR[0], ARR[1], ..., ARR[N–1]** is sorted.

Selection Sort-Example

Sort the values in the array using Selection Sort:

39	9	81	45	90	27	72	18

PASS	POS	ARR[0]	ARR[1]	ARR[2]	ARR[3]	ARR[4]	ARR[5]	ARR[6]	ARR[7]
1	1	9	39	81	45	90	27	72	18
2	7	9	18	81	45	90	27	72	39
3	5	9	18	27	45	90	81	72	39
4	7	9	18	27	39	90	81	72	45
5	7	9	18	27	39	45	81	72	90
6	6	9	18	27	39	45	72	81	90
7	6	9	18	27	39	45	72	81	90

Selection Sort-Algorithm

SELECTION SORT(ARR, N)

Step 1: Repeat **Steps 2 and 3** for K = 0 to N-1

Step 2: CALL SMALLEST(ARR, K, N, POS)

Step 3: SWAP A[K] with ARR[POS]

[END OF LOOP]

Step 4: EXIT

SMALLEST (ARR, K, N, POS)

Step 1: [INITIALIZE] SET SMALL = ARR[K]

Step 2: [INITIALIZE] SET POS = K

Step 3: Repeat for J = K+1 to N-1

IF SMALL > ARR[J]

SET SMALL = ARR[J]

SET POS = J

[END OF IF]

[END OF LOOP]

Step 4: RETURN POS

Selection Sort-Algorithm Explanation

In the algorithm, during the **Kth** pass, find the position **POS** of the smallest elements from ARR[K], ARR[K+1], ..., ARR[N].

To find the smallest element, use a variable SMALL to hold the smallest value in the sub-array ranging from ARR[K] to ARR[N].

Then, swap ARR[K] with ARR[POS]. This procedure is repeated until all the elements in the array are sorted.

Merge Sort

This algorithm works by using a divide-and-conquer strategy - divide, conquer and combine algorithmic paradigm.

The merge sort algorithm works as follows:

- **Divide** means partitioning the n-element array to be sorted into two sub-arrays of n/2 elements. If **A** is an array containing zero or one element, then it is already sorted. However, if there are more elements in the array, divide **A** into two sub-arrays, **A1** and **A2**, each containing about half of the elements of **A**.
- **Conquer** means sorting the two sub-arrays recursively using merge sort.
- **Combine** means merging the two sorted sub-arrays of size n/2 to produce the sorted array of n elements.

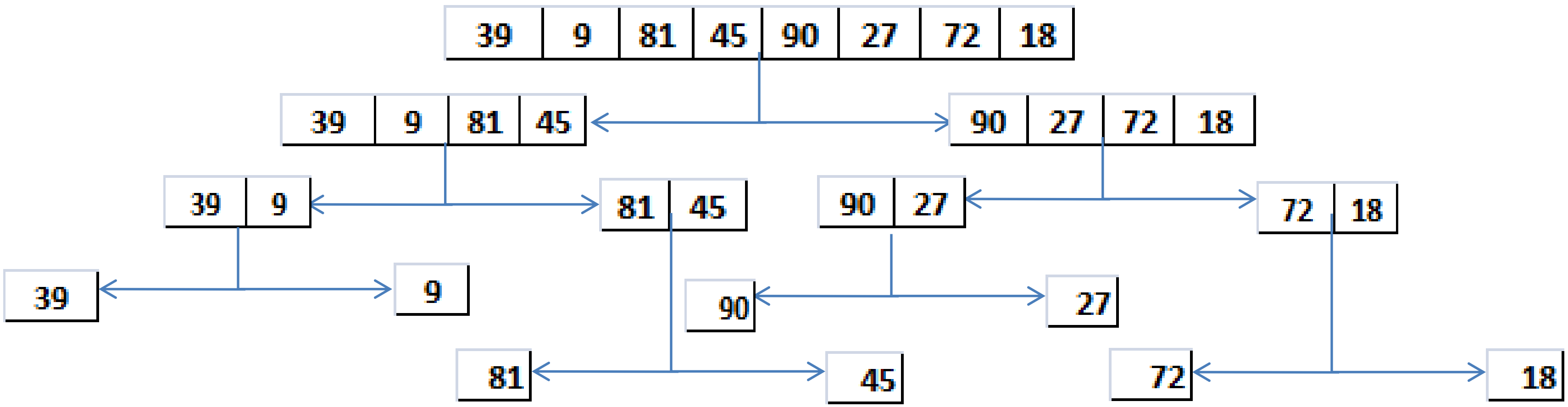
Merge Sort-Technique

Merge sort works as follows:

- If the array is of length 0 or 1, then it is already sorted.
- Otherwise, divide the unsorted array into two sub-arrays of about half the size.
- Use merge sort algorithm recursively to sort each sub-array.
- Merge the two sub-arrays to form a single sorted list.

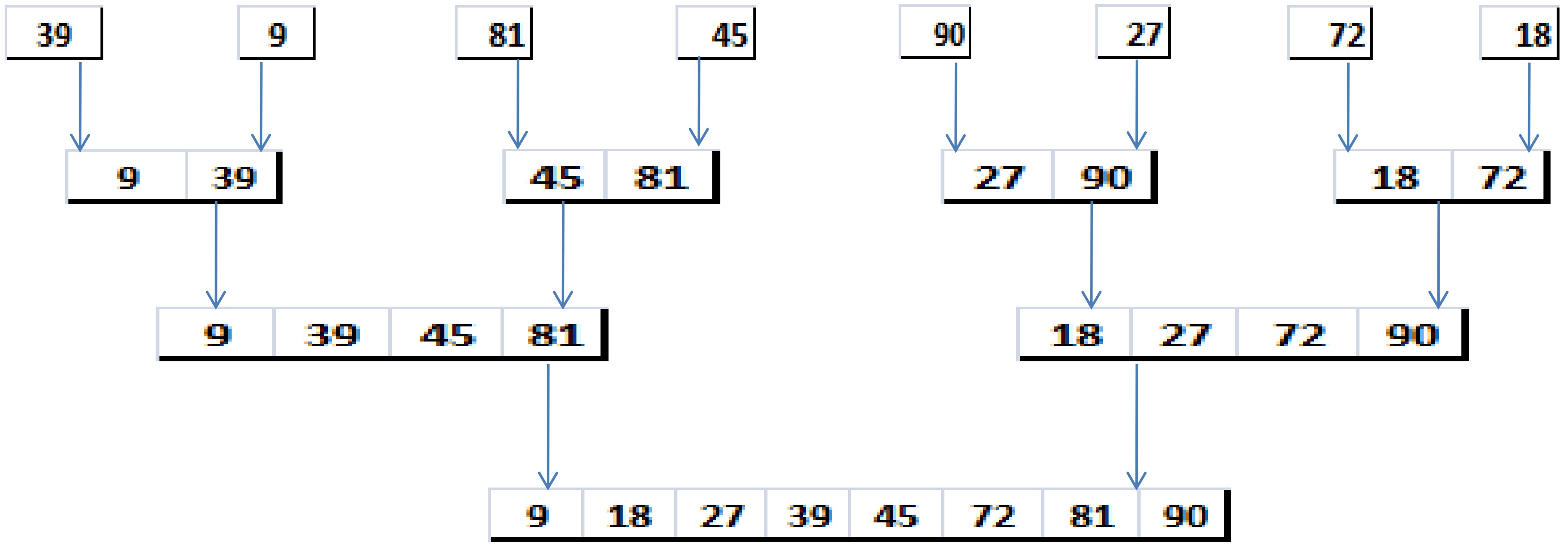
Merge Sort-Example

Sort the values in the array using Merge sort:



Divide and Conquer the array

Merge Sort-Example Continued



Combine the elements to form a sorted array

Merge Sort-Example Continued

- The merge sort algorithm uses a function merge which combines the sub-arrays to form a sorted array.

- While the merge sort algorithm recursively divides the list into smaller lists, the merge algorithm conquers the list to sort the elements in individual lists.

- Finally, the smaller lists are merged to form one list.

Merge Sort-Example Continued

								TEMP							
9	39	45	81	18	27	72	90	9							
BEG, I			MID	J			END	INDEX							

- Compare ARR[I] and ARR[J], the smaller of the two is placed in TEMP at the location specified by INDEX and subsequently the value I or J is incremented.

								TEMP							
9	39	45	81	18	27	72	90	9	18						
BEG	I		MID	J			END		INDEX						

9	39	45	81	18	27	72	90	9	18	27					
BEG	I		MID		J		END			INDEX					

9	39	45	81	18	27	72	90	9	18	27	39				
BEG	I		MID			J	END				INDEX				

9	39	45	81	18	27	72	90	9	18	27	39	45			
BEG		I	MID			J	END					INDEX			

9	39	45	81	18	27	72	90	9	18	27	39	45	72		
BEG			I, MID			J	END						INDEX		

9	39	45	81	18	27	72	90	9	18	27	39	45	72	81	
BEG			I, MID				J END							INDEX	

- When I is greater than MID, copy the remaining elements of the right sub-array in TEMP.

9	39	45	81	18	27	72	90	9	18	27	39	45	72	81	90
BEG			MID	I			J END								INDEX

UNIVERSITY OF ENGINEERING & MANAGEMENT, KOLKATA

Merge Sort-Algorithm

```
MERGE (ARR, BEG, MID, END)
Step 1: [INITIALIZE] SET I = BEG, J = MID + 1, INDEX = 0
Step 2: Repeat while (I <= MID) AND (J<=END)
                IF ARR[I] < ARR[J]
                        SET TEMP[INDEX] = ARR[I]
                        SET I = I + 1
                ELSE
                        SET TEMP[INDEX] = ARR[J]
                        SET J = J + 1
                [END OF IF]
        SET INDEX = INDEX + 1
        [END OF LOOP]
Step 3: [Copy the remaining elements of right sub-array, if any]
        IF I > MID
                Repeat while J <= END
                        SET TEMP[INDEX] = ARR[J]
                        SET INDEX = INDEX + 1, SET J = J + 1
                [END OF LOOP]
[Copy the remaining elements of left sub-array, if any]
        ELSE
                Repeat while I <= MID
                        SET TEMP[INDEX] = ARR[I]
                        SET INDEX = INDEX + 1, SET I = I + 1
                [END OF LOOP]
        [END OF IF]
Step 4: [Copy the contents of TEMP back to ARR] SET K= INDEX
Step 5: Repeat while K < INDEX
                SET ARR[K] = TEMP[K]
                SET K = K + 1
        [END OF LOOP]
Step 6: END
```

Merge Sort-Algorithm Continued

MERGE_SORT(ARR, BEG, END)

```
Step 1: IF BEG < END
    SET MID = (BEG + END)/2
    CALL MERGE_SORT (ARR, BEG, MID)
    CALL MERGE_SORT (ARR, MID + 1, END)
    MERGE (ARR, BEG, MID, END)
[END OF IF]
Step 2: END
```

Quick Sort

- This algorithm (also known as partition exchange sort) works by using a divide-and-conquer strategy to divide a single unsorted array into two smaller sub-arrays. The quick sort algorithm works as follows:

– Select an element pivot from the array elements.

– Rearrange the elements in the array in such a way that all elements that are less than the pivot appear before the pivot and all elements greater than the pivot element come after it (equal values can go either way).

– After such a partitioning, the pivot is placed in its final position. This is called the partition operation.

– Recursively sort the two sub-arrays thus obtained. (One with sub-list of values smaller than that of the pivot element and the other having higher value elements.)

– The base case of the recursion occurs when the array has zero or one element because in that case the array is already sorted.

Quick Sort-Technique

Quick sort works as follows:

- 1. Set the index of the first element in the array to loc and left variables. Also, set the index of the last element of the array to the right variable.

  That is, **loc = 0, left = 0,** and **right = n–1** (where n in the number of elements in the array)

- 2. Start from the element pointed by right and scan the array from right to left, comparing each element on the way with the element pointed by the variable **loc.**

  That is, a[loc] should be less than a[right].

  (a) If that is the case, then simply continue comparing until right becomes equal to **loc**. Once right = **loc**, it means the pivot has been placed in its correct position.

  (b) However, if at any point, we have a[loc] > a[right], then interchange the two values and jump to Step 3.

  (c) Set **loc** = right

Quick Sort-Technique

- 3. Start from the element pointed by left and scan the array from left to right, comparing each element on the way with the element pointed by **loc**.

  That is, a[loc] should be greater than a[left].

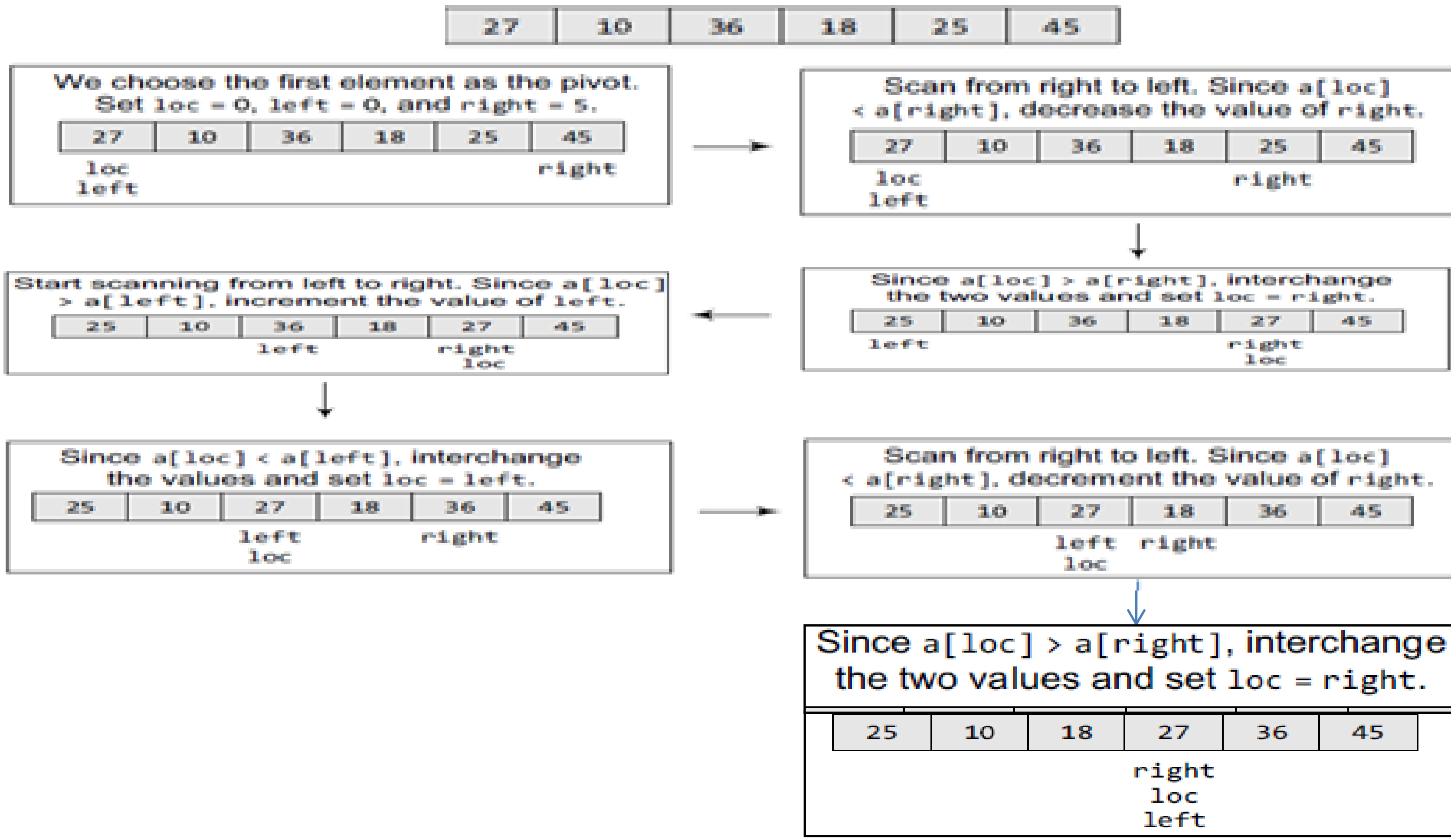
  (a) If that is the case, then simply continue comparing until left becomes equal to **loc**. Once **left = loc**, it means the pivot has been placed in its correct position.

  (b) However, if at any point, we have a[loc] < a[left], then interchange the two values and jump to Step 2.

  (c) Set **loc** = left.

Quick Sort-Example

Sort the values in the array using Quick sort:



Quick Sort-Example Continued

- 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-Algorithm

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-Algorithm Continued

```
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
```

Quick Sort-Pros & Cons

- It is faster than other algorithms such as bubble sort, selection sort, and insertion sort.
- Quick sort can be used to sort arrays of small size, medium size, or large size.
- On the flip side, quick sort is complex and massively recursive.

Quick Sort-Comparison of Algorithms

Algorithm	Average Case	Worst Case
Bubble sort	$O(n^2)$	$O(n^2)$
Bucket sort	$O(n.k)$	$O(n^2.k)$
Selection sort	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n^2)$	$O(n^2)$
Shell sort	–	$O(n \log^2 n)$
Merge sort	$O(n \log n)$	$O(n \log n)$
Heap sort	$O(n \log n)$	$O(n \log n)$
Quick sort	$O(n \log n)$	$O(n^2)$

Sort-Assignments

- Write a program in C to sort an array using bubble sort algorithm.
- Write a program in C to sort an array using insertion sort algorithm.
- Write a program in C to sort an array using selection sort algorithm.
- Write a program in C to sort an array using quick sort algorithm.
- Write a program in C to sort an array using merge sort algorithm.

Thank You