Data Structure & Algorithms (DSA) : CCT203

Semester - III, B.Tech. CSE (Cyber Security)

UNIT IV

GC Code: dyb4gii

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

Course Objectives

1. CO1: To impart to students the basic concepts of data structures and algorithms.
2. CO2: To familiarize students on different searching and sorting techniques.
3. CO3: To prepare students to use linear (stacks, queues, linked lists) and nonlinear (trees, graphs) data structures.
4. CO4: To enable students to devise algorithms for solving real-world problems.

Text Books & Reference Books

Text Books:

1. Ellis Horowitz, Sartaj Sahni & Susan Anderson-Freed, Fundamentals of Data Structures in C, Second Edition, Universities Press, 2008.
2. Mark Allen Weiss; Data Structures and Algorithm Analysis in C; Second Edition; Pearson Education; 2002.
3. G.A.V. Pai; Data Structures and Algorithms: Concepts, Techniques and Application; First Edition; McGraw Hill; 2008.

Reference Books:

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein; Introduction to Algorithms; Third Edition; PHI Learning; 2009.
2. Ellis Horowitz, Sartaj Sahni and Sanguthevar Rajasekaran; Fundamentals of Computer Algorithms; Second Edition; Universities Press; 2008.
3. A. K. Sharma; Data Structures using C, Second Edition, Pearson Education, 2013.

Course Outcomes

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

On completion of the course the student will be able to

1. Recognize different ADTs and their operations and specify their complexities.
2. Design and realize linear data structures (stacks, queues, linked lists) and analyze their computation complexity.
3. Devise different sorting (comparison based, divide-and-conquer, distributive, and tree-based) and searching (linear, binary) methods and analyze their time and space requirements.
4. Design traversal and path finding algorithms for Trees and Graphs.

UNIT IV Sorting and Searching

Sorting:

- Different approaches to sorting
- Properties of different sorting algorithms (insertion, Shell, quick, merge, heap, counting)
- Performance analysis and comparison.

Searching:

- Necessity of a robust search mechanism
- Searching linear lists (linear search, binary search)
- Complexity analysis of search methods

Sorting

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- Sorting means arranging the elements of an array so that they are placed in some relevant order which may be either ascending or descending
- 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} ;
- There are two types of sorting:
  - **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

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

Sorting on Multiple Keys

- Many a times, when performing real-world applications, it is desired to sort arrays of records using multiple keys. This situation usually occurs when a **single key is not sufficient** to uniquely identify a record.
- For example, in a big organization we may want to sort a list of employees on the basis of their departments first and then according to their names in alphabetical order.
- Customers' addresses can be sorted based on the name of the city and then the street.
- In a library, the information about books can be sorted alphabetically based on titles and then by authors' names, etc.
- Data records can be sorted based on a property. Such a component or property is called a **sort key.**
- A sort key can be defined using **two or more sort keys.**
- In such a case, the first key is called the **primary sort key**, the second is known as the **secondary sort key**, etc.

Example: Sorting on Multiple Keys

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- Consider,

Name	Department	Salary	Phone Number
Janak	Telecommunications	1000000	9812345678
Raj	Computer Science	890000	9910023456
Aditya	Electronics	900000	7838987654
Huma	Telecommunications	1100000	9654123456
Divya	Computer Science	750000	9350123455

- Take department as the primary key and name as the secondary key, then the sorted order of records can be given as:

Name	Department	Salary	Phone Number
Divya	Computer Science	750000	9350123455
Raj	Computer Science	890000	9910023456
Aditya	Electronics	900000	7838987654
Huma	Telecommunications	1100000	9654123456
Janak	Telecommunications	1000000	9812345678

Practical Considerations for Internal Sorting

- Here, the logic to sort the records will be same and only the implementation details will differ
- When analysing the performance of different sorting algorithms, the practical considerations would be the following:
  - Number of sort key comparisons that will be performed
  - Number of times the records in the list will be moved
  - Best case performance
  - Worst case performance
  - Average case performance
  - Stability of the sorting algorithm where stability means that equivalent elements or records retain their relative positions even after sorting is done

Bubble Sort

6 5 3 1 8 7 2 4

Bubble Sort

- In bubble sorting, 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.
- This procedure of sorting is called bubble sorting because elements 'bubble' to the top of the list. Note that at the end of the first pass, the largest element in the list will be placed at its proper position (i.e., at the end of the list).
- If the elements are to be sorted in descending order, then in first pass the smallest element is moved to the highest index of the array

Ex: Bubble Sort

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

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

- Pass 1:

30, 52, 29, 87, 63, 27, 19, 54	(a) Compare 30 and 52. Since 30 < 52, no swapping is done.
30, **52, 29,** 87, 63, 27, 19, 54	(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.
30, 29, 52, **87, 63**, 27, 19, 54	(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.

After Pass 1: 30, 29, 52, 63, 27, 19, 54, (87):Largest Element is placed at the highest index of array

Ex: Bubble Sort

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

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

- Pass 2:

30, 29, 52, 63, 27, 19, 54, (87)   (a) Compare 30 and 29. Since 30 > 29, swapping is done.

29, **30, 52**, 63, 27, 19, 54, (87)   (b) Compare 30 and 52. Since 30 < 52, no swapping is done.

30, 29, **52, 63**, 27, 19, 54, (87)   (c) Compare 52 and 63. Since 52 < 63, no swapping is done.

30, 29, 52, **63, 27,** 19, 54, (87)   (d) Compare 63 and 27. Since 63 > 27, swapping is done.

29, 30, 52, 27, **63, 19,** 54, (87)   (e) Compare 63 and 19. Since 63 > 19, swapping is done.

29, 30, 52, 27, 19, **63, 54**, (87)   (f) Compare 63 and 54. Since 63 > 54, swapping is done.

After pass 2: 29, 30, 52, 27, 19, 54, (63, 87) : Second Largest Element is placed at the second highest index of array

Ex: Bubble Sort

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

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

- Pass 3:

29, 30, 52, 27, 19, 54, (63, 87) (a) Compare 29 and 30. Since 29 < 30, no swapping is done.

29, **30, 52**, 27, 19, 54, (63, 87) (b) Compare 30 and 52. Since 30 < 52, no swapping is done.

29, 30, **52, 27**, 19, 54, (63, 87) (c) Compare 52 and 27. Since 52 > 27, swapping is done.

29, 30, 27, **52, 19**, 54, (63, 87) (d) Compare 52 and 19. Since 52 > 19, swapping is done.

29, 30, 27, 19, **52, 54**, (63, 87) (e) Compare 52 and 54. Since 52 < 54, no swapping is done.

After pass 3: 29, 30, 27, 19, 52, (54, 63, 87)
: Third Largest Element is placed at the third highest index of array

Ex: Bubble Sort

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

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

- Pass 4:

29, 30, 27, 19, 52, (54, 63, 87) (a) Compare 29 and 30. Since 29 < 30, no swapping is done.

29, **30, 27**, 19, 52, (54, 63, 87) (b) Compare 30 and 27. Since 30 > 27, swapping is done.

29, 27, **30, 19,** 52, (54, 63, 87) (c) Compare 30 and 19. Since 30 > 19, swapping is done.

29, 27, 19, **30, 52,** (54, 63, 87) (d) Compare 30 and 52. Since 30 < 52, no swapping is done.

After pass 3: 29, 27, 19, 30, (52, 54, 63, 87) **:** Fourth Largest Element is placed at the fourth highest index of array

Ex: Bubble Sort

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

- Pass 5:

29, 27, 19, 30, (52, 54, 63, 87) (a) Compare 29 and 27. Since 29 > 27, swapping is done.

27, **29, 19**, 30, (52, 54, 63, 87) (b) Compare 29 and 19. Since 29 > 19, swapping is done.

27, 19, **29, 30,** (52, 54, 63, 87) (c) Compare 29 and 30. Since 29 < 30, no swapping is done

After pass 5 : 27, 19, 29, (30, 52, 54, 63, 87) **:** Fifth Largest Element is placed at the fifth highest index of array

Ex: Bubble Sort

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

- Pass 6:

27, 19, 29, (30, 52, 54, 63, 87)   (a) Compare 27 and 19. Since 27 > 19, swapping is done.

19, **27, 29**, (30, 52, 54, 63, 87)   (b) Compare 27 and 29. Since 27 < 29, no swapping is done.

After pass 6: 19, 27, (29, 30, 52, 54, 63, 87)
Sixth Largest Element is placed at the sixth highest index of array

- Pass 7:

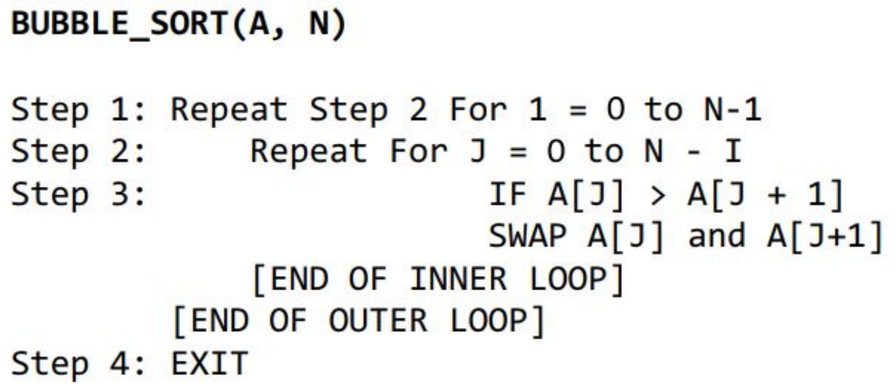
19, 27, (29, 30, 52, 54, 63, 87)   (a) Compare 19 and 27. Since 19 < 27, no swapping is done.

After pass 7: 19, 27, 29, 30, 52, 54, 63, 87. **Observe that the list is completely sorted.**

Algorithm for bubble sort

In Bubble Sort,

- The outer loop is for the total number of passes which is N–1.
- The inner loop will be executed for every pass.
- However, the frequency of the inner loop will decrease with every pass because after every pass, one element will be in its correct position.
- Therefore, for every pass, the inner loop will be executed N–I times, where N is the number of elements in the array and I is the count of the pass.



```
BUBBLE_SORT(A, N)

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

Complexity of Bubble Sort

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- Total Passes = N - 1
- Total No of Comparisons:
  - 1st Pass: N-1
  - 2nd Pass: N-2
  - 3rd Pass: N-3, and so on.

Hence,

- $f(n) = (n - 1) + (n - 2) + \ldots.. + 3 + 2 + 1$
- $f(n) = n(n - 1)/2$
- $\mathbf{f(n) = n^2/2 + O(n) = O(n^2)}$
- So, the time required to execute bubble sort is proportional to **n^2** where n is the total number of elements in the array.

```
BUBBLE_SORT(A, N)

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

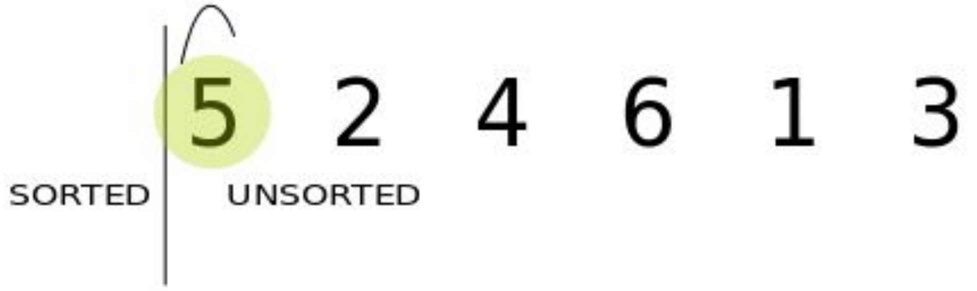
The nested loop is a dependant quadratic loop.

```
for(i=0;i<10;i++)
        for(j=0; j<=i;j++)
                statement block;
```

Complexity of such loops, **$f(n)=n(n + 1)/2 = n^2$**

Write a program to enter n numbers in an array. Redisplay the array with elements being sorted in ascending order.

Insertion Sort

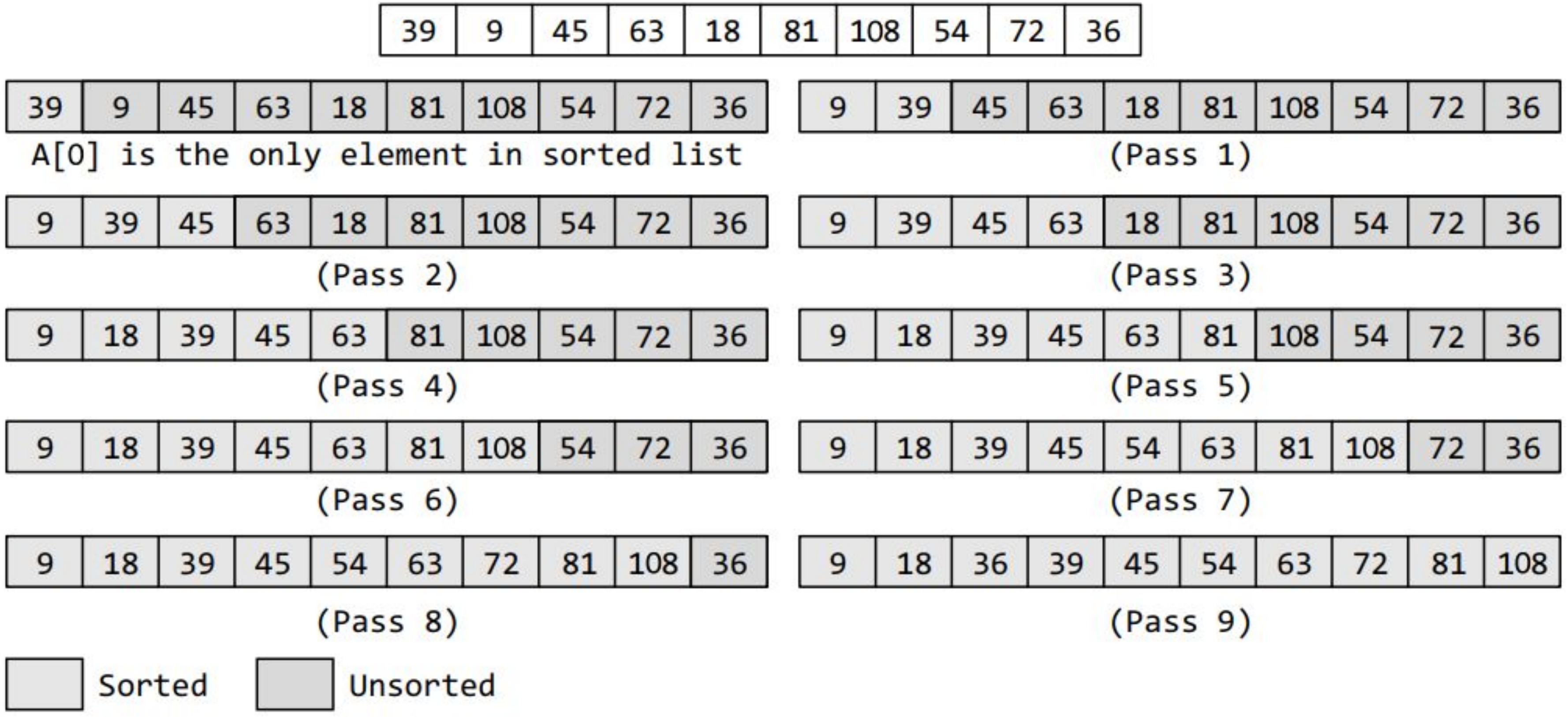


Insertion Sort

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.

Ex: Insertion Sort



39	9	45	63	18	81	108	54	72	36

39	9	45	63	18	81	108	54	72	36

A[0] is the only element in sorted list

9	39	45	63	18	81	108	54	72	36

(Pass 1)

9	39	45	63	18	81	108	54	72	36

(Pass 2)

9	39	45	63	18	81	108	54	72	36

(Pass 3)

9	18	39	45	63	81	108	54	72	36

(Pass 4)

9	18	39	45	63	81	108	54	72	36

(Pass 5)

9	18	39	45	63	81	108	54	72	36

(Pass 6)

9	18	39	45	54	63	81	108	72	36

(Pass 7)

9	18	39	45	54	63	72	81	108	36

(Pass 8)

9	18	36	39	45	54	63	72	81	108

(Pass 9)

▢ Sorted ▨ Unsorted

Algorithm for insertion sort

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

In Insertion Sort,

- Step 1 executes a for loop which will be repeated for each element in the array.
- In Step 2, we store the value of the Kth element in TEMP.
- In Step 3, we set the Jth index in the array.
- In Step 4, a for loop is executed that will create space for the new element from the unsorted list to be stored in the list of sorted elements.
- Finally, in Step 5, the element is stored at the (J+1)th location.

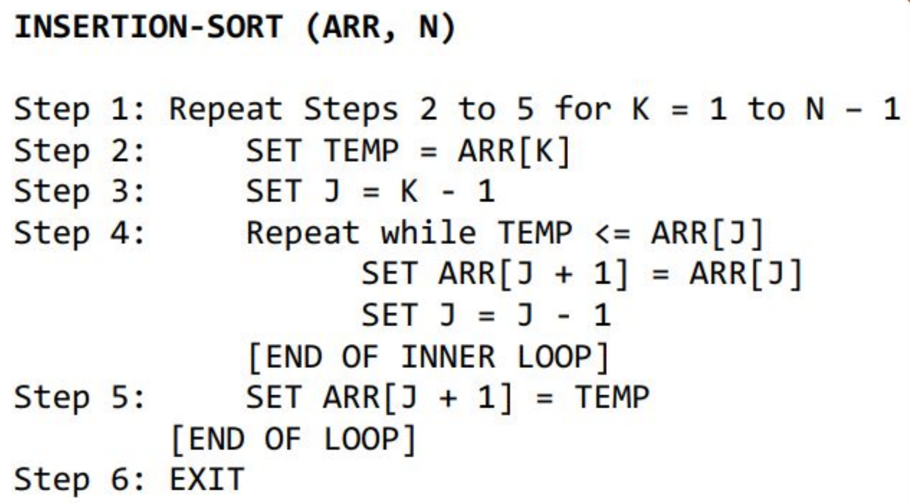
```
INSERTION-SORT (ARR, N)

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
```

Complexity of Insertion Sort

- Total Passes = N - 1
- Total No of Comparisons:
  - 1st Pass: 1
  - 2nd Pass: 2
  - 3rd Pass: 3, and so on.

Hence,

- $f(n) = (n – 1) + (n – 2) + ... + 3 + 2 + 1$
- $f(n) = n (n – 1)/2$
- $\mathbf{f(n) = n^2/2 + O(n) = O(n^2)}$
- So, the time required to execute insertion sort is proportional to $\mathbf{n^2}$ where n is the total number of elements in the array.



```
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
```

Advantages of Insertion Sort

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- It is **easy to implement** and **efficient to use** on small sets of data.
- It **performs better** than algorithms like selection sort and bubble sort. Insertion sort algorithm is simpler than shell sort, with only a small trade-off in efficiency. It is over twice as fast as the bubble sort and almost 40 percent faster than the selection sort.
- It requires **less memory space** (only O(1) of additional memory space)

Write a program to sort an array using insertion sort algorithm.

Shri Ramdeobaba College of Engineering and Management, Nagpur

Selection Sort

Selection Sort

RCOEM

Shri Ramdeobaba College of Engineering and Management, Nagpur

- Selection sort has a quadratic running time complexity of $O(n^2)$, hence, making it inefficient to be used on large lists.
- But, it is noted for its simplicity and also has performance advantages over more complicated algorithms in certain situations.

Example: Selection Sort

- Example: Sort the array given below 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

Technique: Selection Sort

Consider an array ARR with N elements. Selection sort works as follows:

- 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.

Algorithm for Selection Sort

```
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(ARR, N)

Step 1: Repeat Steps 2 and 3 for K = 1
        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
```

Complexity: Selection Sort

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- In Pass 1, selecting the element with the smallest value calls for scanning all n elements;thus, n–1 comparisons are required in the first pass. Then, the smallest value is swapped with the element in the first position.
- In Pass 2, selecting the second smallest value requires scanning the remaining n – 1 elements and so on. Therefore, $= (n - 1) + (n - 2) + \ldots + 2 + 1 = n(n - 1) / 2 =$ **$O(n^2)$** comparisons

- **Advantages of Selection Sort**
  - It is simple and easy to implement.
  - It can be used for small data sets.
  - It is 60 percent more efficient than bubble sort.

Write a C program to sort an array using selection sort algorithm

RCOEM

Shri Ramdeobaba College of Engineering and Management, Nagpur

Merge Sort

Merge Sort

- Merge sort uses the **divide, conquer,** and **combine** algorithmic paradigm.
- **Divide:**
  - **Partitioning the n-element array** to be sorted into **two sub-arrays of n/2 elements.**
  - If A is an array with **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:**
  - It means **sorting the two sub-arrays recursively** using **merge sort.**
- **Combine:**
  - It means **merging the two sorted subarrays** of **size n/2** to produce the **sorted array** of n elements

Concept and Steps: Merge Sort

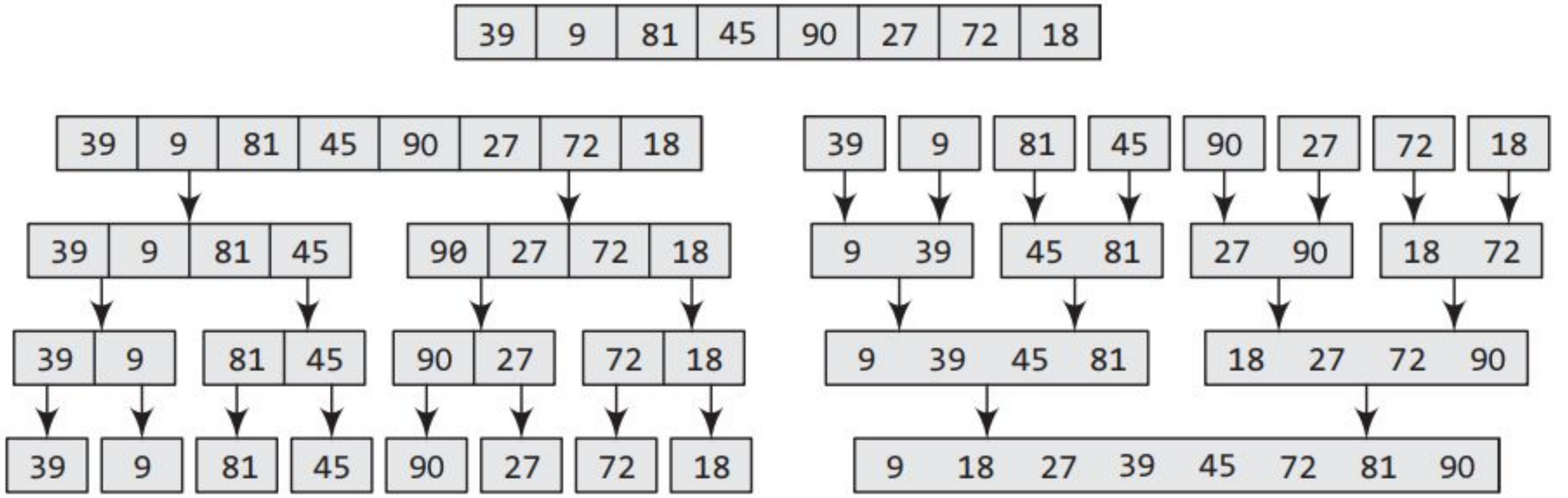
RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- Two main concepts to **improve its performance** (running time):
  - A smaller list takes fewer steps and thus less time to sort than a large list.
  - As number of steps is relatively less, thus less time is needed to create a sorted list from two sorted lists rather than creating it using two unsorted lists.

- The basic steps of a merge sort algorithm are 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.

Example: Merge Sort

- Sort the array given below using merge sort



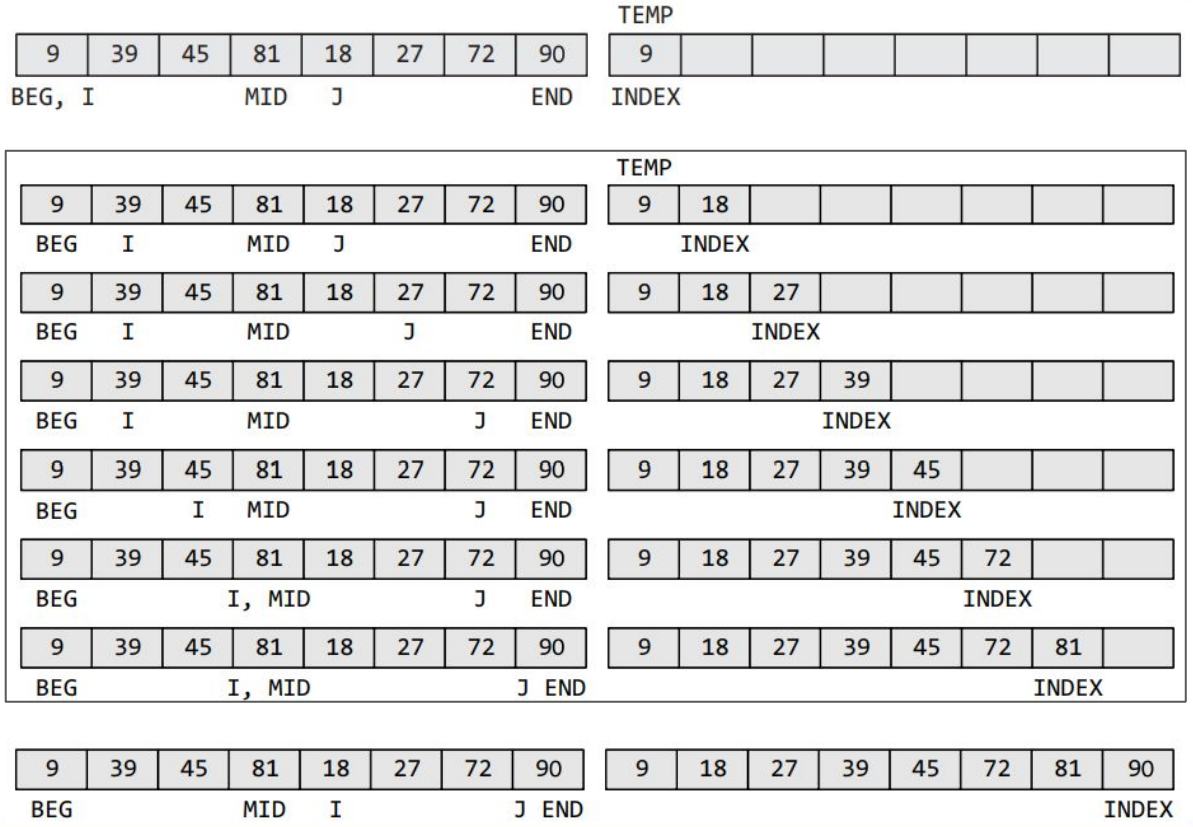
(Divide and Conquer the array)

(Combine the elements to form a sorted array)

Understand: Merge Sort

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- Consider two sub-lists each containing four elements. {09, 39, 45, 81} & {18, 27, 72, 90}
- Compare ARR[I] and ARR[J], the smaller value is placed in TEMP at the location specified by INDEX and subsequently the value I or J is incremented
- 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							
BEG, I			MID	J			END	INDEX							
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	
9	39	45	81	18	27	72	90	9	18	27	39	45	72	81	90
BEG			MID	I			J END								INDEX

Algorithm: Merge Sort

```
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
```

```
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=0
Step 5: Repeat while K < INDEX
               SET ARR[K] = TEMP[K]
               SET K = K + 1
      [END OF LOOP]
Step 6: END
```

Complexity: Merge Sort

- The running time in the average case and the worst case can be given as **O(n log n).**
- Although merge sort has an optimal time complexity, it needs an additional space of **O(n)** for the temporary array TEMP.

Write a C program to implement merge sort.

Quick Sort

Quick Sort

- Like merge sort, this algorithm 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:

  1. Select an **element pivot** from the array elements.

  2. 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.**

  3. 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.)

Quick Sort

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

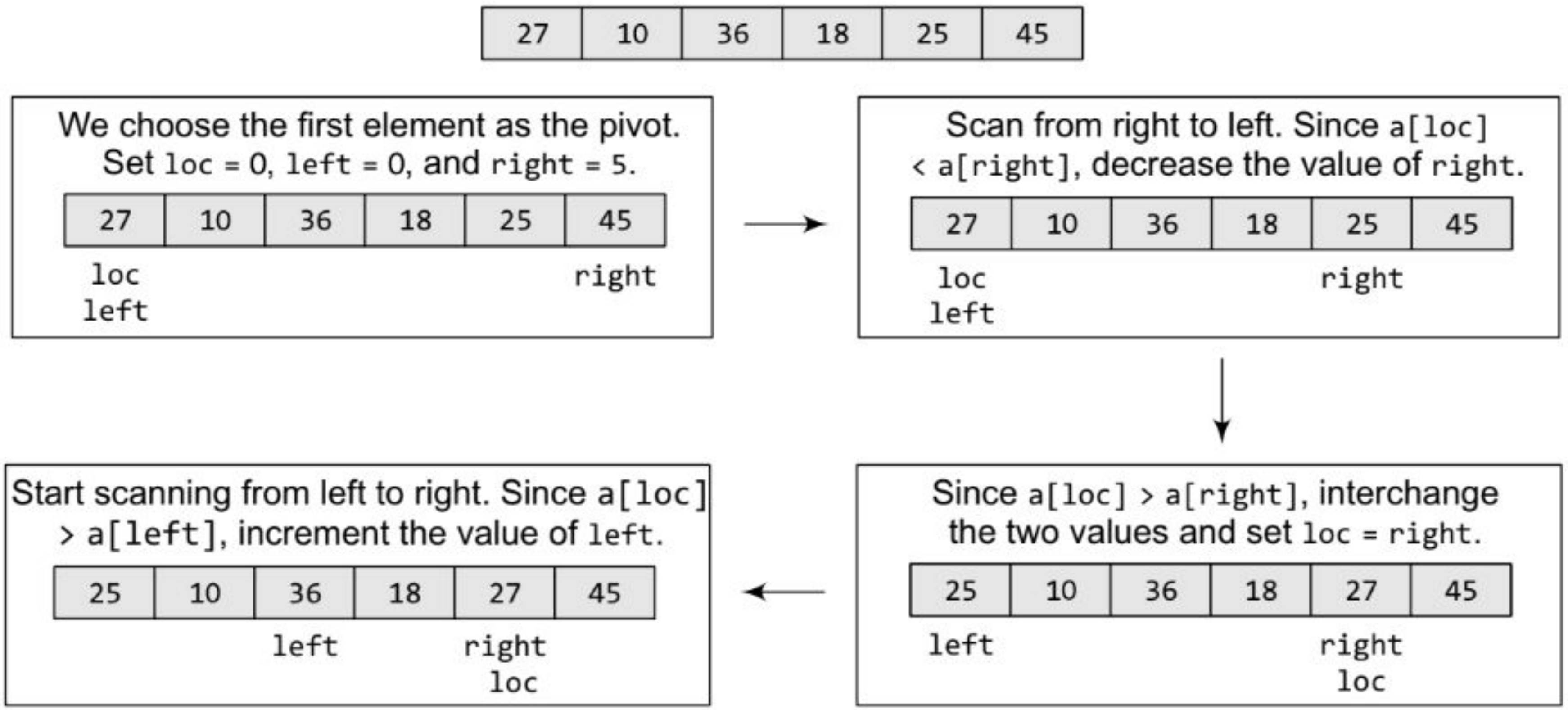
- Like merge sort, the base case of the recursion occurs when the array has zero or one element because in that case the array is already sorted. After each iteration, one element (pivot) is always in its final position. Hence, with every iteration, there is one less element to be sorted in the array.
- Thus, the main task is to find the pivot element, which will partition the array into two halves.
- To understand how we find the pivot element, follow the steps given below. (We take the first element in the array as pivot.)

Quick Sort: Technique

1. Set **loc = 0, left = 0, right = MAX–1**
2. Scan the array from Right to Left, Compare **a[loc] & a[right]**
    a. If **a[loc] < a[right],** continue comparing until right = loc. Once right = loc, it means the pivot has been placed in its correct position.
    b. Else, if, **a[loc] > a[right]**, then **swap the two values** and jump to Step 3.
    c. Set **loc = right**
3. Scan the array from Left to Right, Compare **a[loc] & a[left]**
    a. If **a[loc] > a[left]**, continue comparing until **left = loc**. Once **left = loc**, it means the pivot has been placed in its correct position.
    b. Else, if **a[loc] < a[left]**, then **swap the two values** and jump to Step 2.
    c. Set **loc = left**

Example: Quick Sort

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur



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	

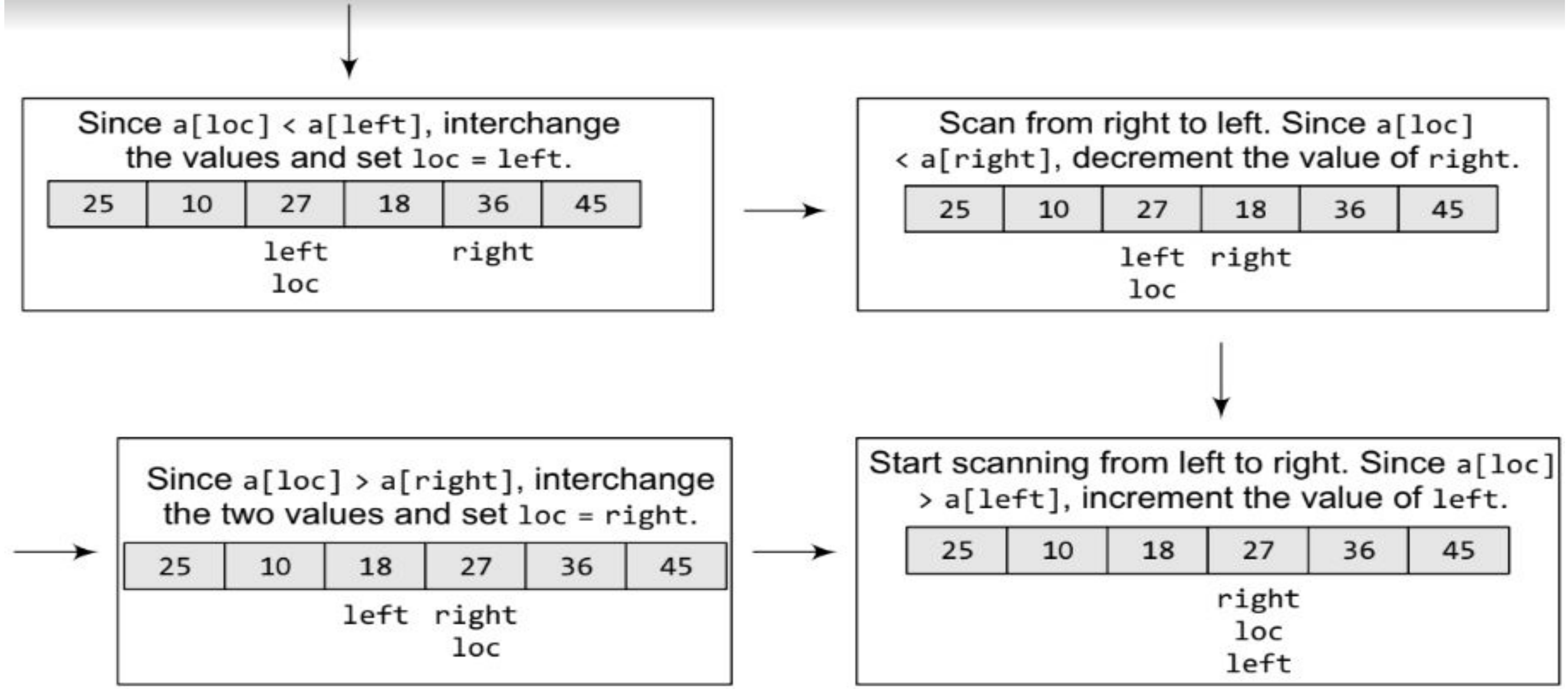
Since a[loc] > a[right], interchange the two values and set loc = right.

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	

Example: Quick Sort



Since a[loc] < a[left], interchange the values and set loc = left.

25	10	27	18	36	45

```
             left      right
             loc
```

Scan from right to left. Since a[loc] < a[right], decrement the value of right.

25	10	27	18	36	45

```
             left right
             loc
```

Since a[loc] > a[right], interchange the two values and set loc = right.

25	10	18	27	36	45

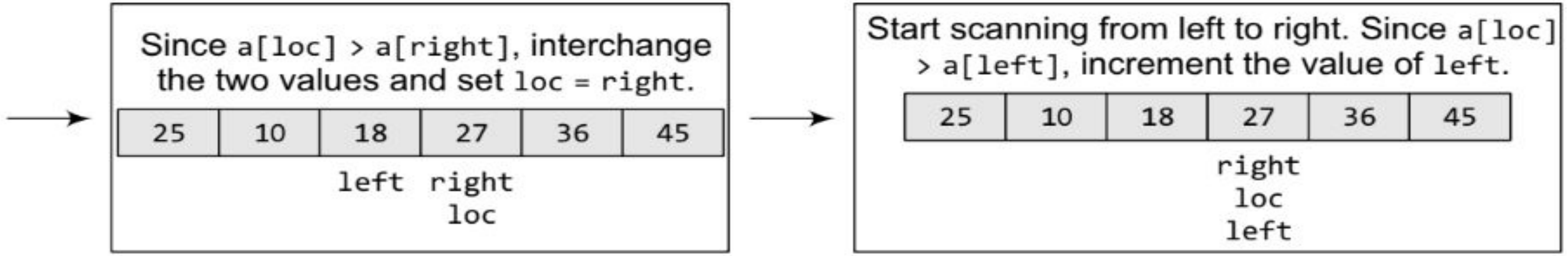
```
             left  right
                   loc
```

Start scanning from left to right. Since a[loc] > a[left], increment the value of left.

25	10	18	27	36	45

```
                   right
                   loc
                   left
```

Quick Sort: Technique


Since a[loc] > a[right], interchange the two values and set loc = right.
25 10 18 27 36 45
left right
loc
Start scanning from left to right. Since a[loc] > a[left], increment the value of left.
25 10 18 27 36 45
right
loc
left

- 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

RCOEM

Shri Ramdeobaba College of Engineering and Management, Nagpur

```
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
```

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

Quick Sort: Complexity

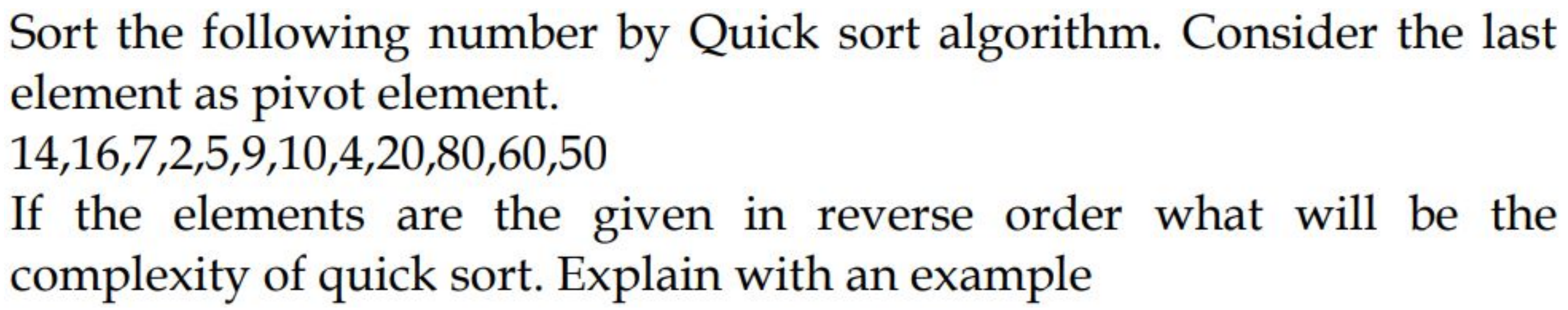
- Practically, the efficiency of quick sort depends on the element which is chosen as the pivot.
- **Its worst-case efficiency = O(n^2).**
- The worst case occurs when the array is already sorted (either in ascending or descending order) and the left-most element is chosen as the pivot.
- However, many implementations randomly choose the pivot element. The randomized version of the quick sort algorithm always has an algorithmic complexity of **O(n log n).**

Pros and Cons of Quick Sort

- 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.

Write a program to implement quick sort algorithm.

Quick Sort Example


Sort the following number by Quick sort algorithm. Consider the last element as pivot element.
14,16,7,2,5,9,10,4,20,80,60,50
If the elements are the given in reverse order what will be the complexity of quick sort. Explain with an example

Solution shared on GC.

Shell Sort

Shell Sort

- It is a generalization of insertion sort.
- While discussing insertion sort, we have observed two things:
  - Insertion sort **works well** when the input data is ‘**almost sorted**’.
  - Insertion sort is quite **inefficient** to use as it moves the values just **one position at a time**.
- Shell sort is considered an **improvement over insertion sort** as it compares elements separated by a gap of several positions.
- In Shell sort, elements are sorted in multiple passes and in each pass, data are taken with smaller and smaller gap sizes.
- However, the last step of shell sort is a plain insertion sort
- But by the time we reach the last step, the elements are already ‘almost sorted’, and hence it provides good performance

Shell Sort: Technique

Step 1:

Arrange the elements of the array in the form of a table and sort the columns (using insertion sort).

Step 2:

Repeat Step 1, each time with smaller number of longer columns in such a way that at the end, there is only one column of data to be sorted

Shell Sort: Example

Sort the elements given below using shell sort. 63, 19, 7, 90, 81, 36, 54, 45, 72, 27, 22, 9, 41, 59, 33

- Arrange the elements of the array in the form of a table and sort the columns.

63	19	7	90	81	36	54	45
72	27	22	9	41	59	33	

Result:

63	19	7	9	41	36	33	45
72	27	22	90	81	59	54	

- The elements of the array can be given as:
  63, 19, 7, 9, 41, 36, 33, 45, 72, 27, 22, 90, 81, 59, 54

- Repeat Step 1 with smaller number of long columns.

Shell Sort: Example

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

63	19	7	9	41
36	33	45	72	27
22	90	81	59	54

Result:

22	19	7	9	27
36	33	45	59	41
63	90	81	72	54

- The elements of the array can be given as:
  22, 19, 7, 9, 27, 36, 33, 45, 59, 41, 63, 90, 81, 72, 54
- Repeat Step 1 with smaller number of long columns

22	19	7
9	27	36
33	45	59
41	63	90
81	72	54

Result:

9	19	7
22	27	36
33	45	54
41	63	59
81	72	90

Shell Sort: Example

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- The elements of the array can be given as:

  9, 19, 7, 22, 27, 36, 33, 45, 54, 41, 63, 59, 81, 72, 90

- Finally, arrange the elements of the array in a single column and sort the column.

- Finally, the elements of the array can be given as:
  7, 9, 19, 22, 27, 33, 36, 41, 45, 54, 59, 63, 72, 81, 90

	Result:
9	7
19	9
7	19
22	22
27	27
36	33
33	36
45	41
54	45
41	54
63	59
59	63
81	72
72	81
90	90

Shell Sort: Algorithm

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

```
Shell_Sort(Arr, n)

Step 1: SET FLAG = 1, GAP_SIZE = N
Step 2: Repeat Steps 3 to 6 while FLAG = 1 OR GAP_SIZE > 1
Step 3:     SET FLAG = 0
Step 4:     SET GAP_SIZE = (GAP_SIZE + 1) / 2
Step 5:     Repeat Step 6 for I = 0 to I < (N - GAP_SIZE)
Step 6:             IF Arr[I + GAP_SIZE] > Arr[I]
                        SWAP Arr[I + GAP_SIZE], Arr[I]
                        SET FLAG = 0
Step 7: END
```

Write a C program to implement shell sort algorithm

Radix Sort

Radix Sort

- Radix sort is a linear sorting algorithm
- for integers, it uses buckets of digits 0 to 9, for alphabets, it uses buckets of 26 letters

Radix Sort: Example

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- Sort the numbers given below using radix sort: 345, 654, 924, 123, 567, 472, 555, 808, 911
- In 1st Pass, the numbers are sorted according to the digit at **ones place**

Number	0	1	2	3	4	5	6	7	8	9
345						345				
654					654					
924					924					
123				123						
567								567		
472			472							
555						555				
808									808	
911		911								

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

Radix Sort: Example

- After 1st pass, the numbers are collected bucket by bucket. The new list thus formed is used as an input for the next pass.
- In the second pass, the numbers are sorted according to the digit at the **tens place**.

Number	0	1	2	3	4	5	6	7	8	9
911		911								
472								472		
123			123							
654						654				
924			924							
345					345					
555						555				
567							567			
808	808									

Radix Sort: Example

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- In the 3rd pass, the numbers are sorted according to the digit at the hundreds place.

Number	0	1	2	3	4	5	6	7	8	9
808									808	
911										911
123		123								
924										924
345				345						
654							654			
555						555				
567						567				
472					472					

- After the third pass, the list can be given as: **123, 345, 472, 555, 567, 654, 808, 911, 924.**

Radix Sort: Complexity

- Assume that there are n numbers that have to be sorted and k is the number of digits in the largest number.
- Hence, the entire radix sort algorithm takes O(k*n) time to execute

Pros and Cons of Radix Sort

- Radix sort is a very simple algorithm. When programmed properly, radix sort is one of the fastest sorting algorithms for numbers or strings of letters.
- Radix sort takes more space than other sorting algorithms
- As the algorithm is dependent on digits or letters. This feature compromises with the flexibility to sort input of any data type. For every different data type, the algorithm has to be rewritten
- Radix sort is a good choice for many programs that need a fast sort, but there are faster sorting algorithms available.

Radix Sort: Algorithm

```
Algorithm for RadixSort (ARR, N)

Step 1: Find the largest number in ARR as LARGE
Step 2: [INITIALIZE] SET NOP = Number of digits in LARGE
Step 3: SET PASS = 0
Step 4: Repeat Step 5 while PASS <= NOP-1
Step 5:             SET I = 0 and INITIALIZE buckets
Step 6:             Repeat Steps 7 to 9 while I<N-1
Step 7:                   SET DIGIT  = digit at PASSth place in A[I]
Step 8:                   Add A[I] to the bucket numbered DIGIT
Step 9:                   INCEREMENT bucket count for bucket numbered DIGIT
                    [END OF LOOP]
Step 10:            Collect the numbers in the bucket
        [END OF LOOP]
Step 11: END
```

Write a C program to implement radix sort algorithm

Questions on Sorting

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- Which technique of searching an element in an array would you prefer to use and in which situation?
- Define sorting. What is the importance of sorting?
- What are the different types of sorting techniques? Which sorting technique has the least worst case?
- Explain the difference between bubble sort and quicksort. Which one is more efficient?
- Sort the elements 77, 49, 25, 12, 9, 33, 56, 81 using (a) insertion sort (b) selection sort (c) bubble sort (d) merge sort (e) quick sort (f) radix sort (g) shell sort
- Quick sort shows quadratic behaviour in certain situations. Justify.
- Write a recursive function to perform selection sort.
- Compare the running time complexity of different sorting algorithms.
- Discuss the advantages of insertion sort.

Questions on Sorting

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- If the following sequence of numbers is to be sorted using quick sort, then show the iterations of the sorting process. 42, 34, 75, 23, 21, 18, 90, 67, 78
- Sort the following sequence of numbers in descending order using heap sort. 42, 34, 75, 23, 21, 18, 90, 67, 78
- A certain sorting technique was applied to the following data set, 45, 1, 27, 36, 54, 90 After two passes, the rearrangement of the data set is given as below: 1, 27, 45, 36, 54, 90 Identify the sorting algorithm that was applied.
- A certain sorting technique was applied to the following data set, 81, 72, 63, 45, 27, 36 After two passes, the rearrangement of the data set is given as below: 27, 36, 80, 72, 63, 45 Identify the sorting algorithm that was applied.
- A certain sorting technique was applied to the following data set, 45, 1, 63, 36, 54, 90 After two passes, the rearrangement of the data set is given as below: 1, 45, 63, 36, 54, 90 Identify the sorting algorithm that was applied.

Programming Questions on Sorting

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- Write a program to implement bubble sort. Given the numbers 7, 1, 4, 12, 67, 33, and 45. How many swaps will be performed to sort these numbers using the bubble sort.
- Write a program to implement a sort technique that works by repeatedly stepping through the list to be sorted.
- Write a program to implement a sort technique in which the sorted array is built one entry at a time.
- Write a program to implement an in-place comparison sort.
- Write a program to implement a sort technique that works on the principle of divide and conquer strategy.
- Write a program to implement partition-exchange sort.
- Write a program to implement a sort technique which sorts the numbers based on individual digits.

Programming Questions on Sorting

- Write a program to sort an array of integers in descending order using the following sorting techniques: (a) insertion sort (b) selection sort (c) bubble sort (d) merge sort (e) quick sort (f) radix sort (g) shell sort
- Write a program to sort an array of floating point numbers in descending order using the following sorting techniques: (a) insertion sort (b) selection sort (c) bubble sort (d) merge sort (e) quick sort (f) radix sort (g) shell sort
- Write a program to sort an array of names using the bucket sort.

Searching

Searching

- Searching means to find whether a particular value is present in an array or not.
- If yes, the searching process gives the location of that value in the array (Element Found)
- If not, the searching process displays an appropriate message (Element not found)
- There are two popular methods for searching the array elements: **linear search** and **binary search.**

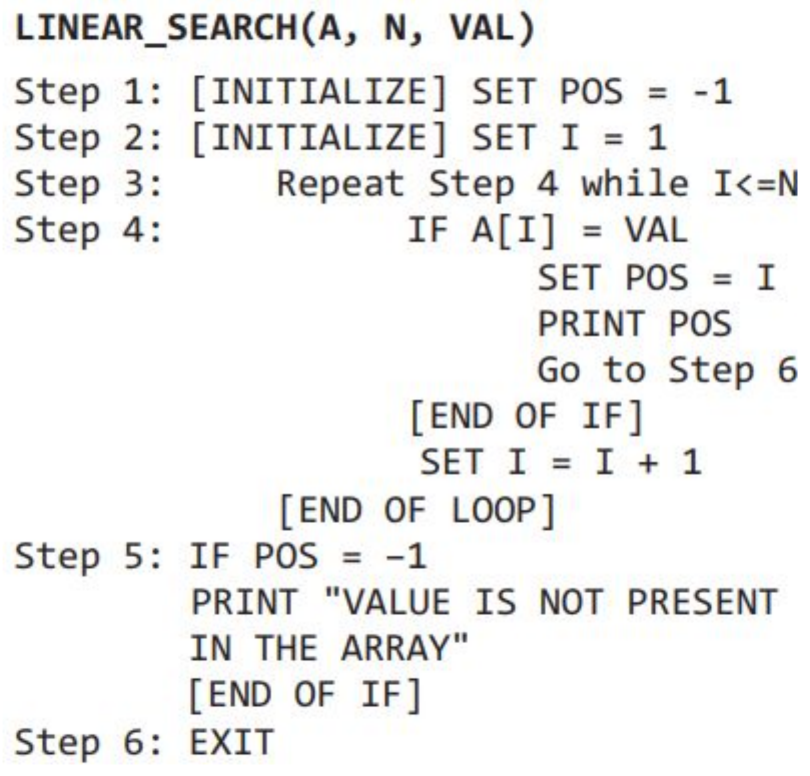
Linear Search

Linear Search

- Linear search, also called as **sequential search.**
- It is a very simple method used for searching an array for a particular value.
- It works by comparing the value to be searched **with every element of the array one by one in a sequence** until a match is found.
- Linear search is mostly used to search an unordered list of elements (array in which data elements are not sorted).
- For example, if an array A[] is declared and initialized as, int A[] = {10, 8, 2, 7, 3, 4, 9, 1, 6, 5}; and the value to be searched is VAL = 7, then searching means to find whether the value '7' is present in the array or not.
- If yes, then it returns the position of its occurrence. Here, POS = 3 (index starting from 0).

Linear Search: Algorithm

- In Steps 1 and 2, initialize the value of POS and I.
- In Step 3, a while loop is executed that would be executed till I is less than N (total number of elements in the array).
- In Step 4, a check is made to see if a match is found between the current array element and VAL. If a match is found, then the position of the array element is printed, else the value of I is incremented to match the next element with VAL. However, if all the array elements have been compared with VAL and no match is found, then it means that VAL is not present in the array



```
LINEAR_SEARCH(A, N, VAL)
Step 1: [INITIALIZE] SET POS = -1
Step 2: [INITIALIZE] SET I = 1
Step 3:     Repeat Step 4 while I<=N
Step 4:          IF A[I] = VAL
                      SET POS = I
                      PRINT POS
                      Go to Step 6
                 [END OF IF]
                 SET I = I + 1
            [END OF LOOP]
Step 5: IF POS = -1
        PRINT "VALUE IS NOT PRESENT
        IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

Linear Search: Complexity

- Linear search executes in **O(n) time** where n is the number of elements in the array.
- Obviously, the best case of linear search is when VAL is equal to the first element of the array.
- In this case, only one comparison will be made.
- Likewise, the worst case will happen when either VAL is not present in the array or it is equal to the last element of the array.
- In both the cases, n comparisons will have to be made.
- However, the performance of the linear search algorithm can be improved by using a sorted array
- **DIY: Write a program to search an element in an array using the linear search technique**

Binary Search

Binary Search

Take an analogy.

- How do we find words in a dictionary?
- We first open the dictionary somewhere in the middle. Then, we compare the first word on that page with the desired word whose meaning we are looking for.
- If the desired word comes before the word on the page, we look in the first half of the dictionary, else we look in the second half.
- Again, we open a page in the first half of the dictionary and compare the first word on that page with the desired word and repeat the same procedure until we finally get the word.
- The same mechanism is applied in the **binary search**

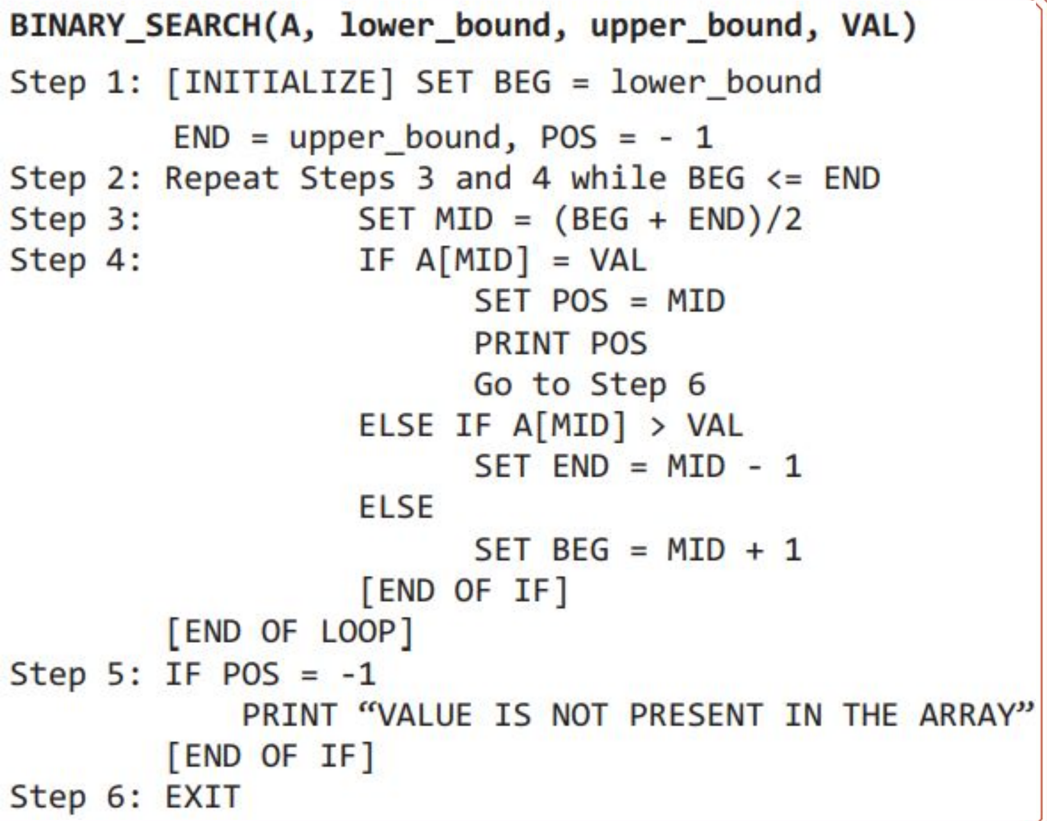
Binary Search: Example

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- Consider an array A[] that is declared and initialized as int A[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; and the value to be searched is VAL = 9.
- The algorithm will proceed in the following manner. **BEG = 0, END = 10, MID = (0 + 10)/2 = 5**
- Now, VAL = 9 and A[MID] = A[5] = 5
- A[5] is less than VAL, therefore, we now search for the value in the second half of the array. So, we change the values of BEG and MID.
- Now, **BEG = MID + 1 = 6, END = 10, MID = (6 + 10)/2 =16/2 = 8**
- VAL = 9 and A[MID] = A[8] = 8
- A[8] is less than VAL, therefore, we now search for the value in the second half of the segment. So, again we change the values of BEG and MID.
- Now, **BEG = MID + 1 = 9, END = 10, MID = (9 + 10)/2 = 9**
- Now, VAL = 9 and A[MID] = 9.

Binary Search: Algorithm

- In this algorithm, we see that BEG and END are the beginning and ending positions of the segment that we are looking to search for the element. MID is calculated as (BEG + END)/2.
- Initially, BEG = lower_bound and END = upper_bound. The algorithm will terminate when A[MID] = VAL. When the algorithm ends, we will set POS = MID. POS is the position at which the value is present in the array.



```
BINARY_SEARCH(A, lower_bound, upper_bound, VAL)
Step 1: [INITIALIZE] SET BEG = lower_bound
            END = upper_bound, POS = - 1
Step 2: Repeat Steps 3 and 4 while BEG <= END
Step 3:             SET MID = (BEG + END)/2
Step 4:             IF A[MID] = VAL
                          SET POS = MID
                          PRINT POS
                          Go to Step 6
                    ELSE IF A[MID] > VAL
                          SET END = MID - 1
                    ELSE
                          SET BEG = MID + 1
                    [END OF IF]
        [END OF LOOP]
Step 5: IF POS = -1
              PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

Binary Search: Technique

- However, if VAL is not equal to A[MID], then the values of BEG, END, and MID will be changed depending on whether VAL is smaller or greater than A[MID].
- (a) If VAL < A[MID], then VAL will be present in the left segment of the array. So, the value of END will be changed as END = MID – 1.
- (b) If VAL > A[MID], then VAL will be present in the right segment of the array. So, the value of BEG will be changed as BEG = MID + 1.
- Finally, if VAL is not present in the array, then eventually, END will be less than BEG. When this happens, the algorithm will terminate and the search will be unsuccessful.

Binary Search: Complexity

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- The complexity of the binary search algorithm can be expressed as **f(n)**, where n is the number of elements in the array.
- The complexity of the algorithm is calculated depending on the number of comparisons that are made.
- In the binary search algorithm, we see that with each comparison, the size of the segment where search has to be made is reduced to half.
- Thus, we can say that, in order to locate a particular value in the array, the total number of comparisons that will be made is given as $\mathbf{2^{f(n)} > n}$ or $\mathbf{f(n) = \log_2 n}$
- **DIY: Write a program to search an element in an array using binary search.**

End of UNIT IV