UNIT II

CHAPTER 2 Searching and Sorting

University Prescribed Syllabus

Searching and sorting : Need of searching and sorting, Concept of internal and external sorting, sort stability, Searching methods: Linear and binary search algorithms, Fibonacci Series.
Sorting methods : Bubble, insertion, Quick, Merge, shell and comparison of all sorting methods. Analyze Insertion sort, Quick Sort, binary search, hashing for Best, Worst and Average case.
Case Study : Study and Analyze Selection sort, bucket sort, radix sort.

Syllabus Topic : Need of Searching and Sorting

2.1 NEED OF SEARCHING AND SORTING

- Nowadays, the most important aspect in this modern world is nothing but the information. Number of times we need to search different types of information for various purposes.

> ❑ *Definition of Searching : Searching is an operation which finds the element with its location in given list.*

- Now consider a message is reached to a person regarding his due of any previous MSEB bill. Here the person has to search for the bill details. If the bills are stored in a sorted format then definitely he will get the details quickly otherwise searching will be a very tedious task and headache for him.
- We come across such experience many times in our day to day life. Number of times we don't get the documents easily and quickly at the time when required.
- All this happen because we don't keep the data in sorted format. Hence sorting is very important.

> ❑ *Definition of Sorting : Sorting is a technique which is used to rearrange the data elements of a list in a standard form of ascending or descending order, which may be numerical, lexicographical, or any user-defined order.*

- The importance of sorting lies in the fact that it is possible to optimize data search to vary high level if the data is available in sorted format.
- Another important use of sorting is that data can be represented in a better readable format.

☞ **Applications of Sorting**

In real life scenarios, there are some examples of sorting as follows :

1. **Telephone Directory :** In a telephone directory, the telephone numbers of people are stored in sorted manner based on their names. It helps us to search any record easily.
2. **Dictionary :** In dictionary, all the words are stored in sorted manner based on their alphabetical order. It helps us to search any word easily and fast.

Unit II In Sem.

Syllabus Topic : Searching – Searching Methods

2.2 SEARCHING METHODS

GQ. 2.2.1 *Write short note on Searching Methods.* **(3 Marks)**

- In search operation if element is found in given list then search is successful otherwise not.
- There are three important types of searching methods.
  (1) Sequential or Linear Search
  (2) Binary Search

Syllabus Topic : Sequential / Linear Search

2.3 SEQUENTIAL / LINEAR SEARCH

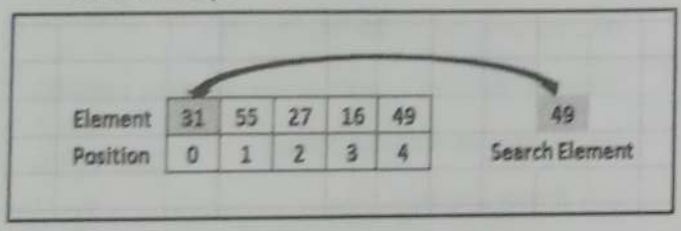
GQ. 2.3.1 *Explain linear search with example.* **(3 Marks)**

- **Concept :** This is simple method of searching.
- In this method, the element is searched in sequential manner; hence this search is called as sequential search.
- This algorithm can be applied on both sorted and unsorted list.
- **Example :** Now consider an array arr[] having 5 elements which are not sorted :

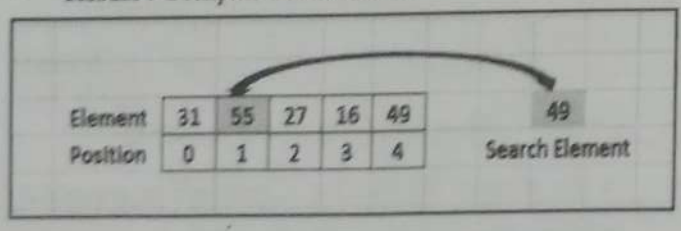
```
int arr[] = {31,55,27,16,49}
```

- We have to search 49.

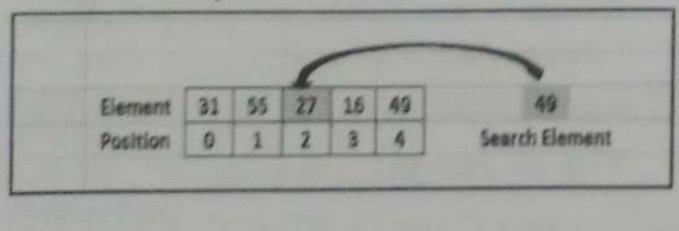
- In unsorted array, the 0^{th} location number is compared with element you want to search, if both are equal then the number is found otherwise element is compared with 1^{st} location element and so on.
- In this method, the element to search is compared with all the array elements from first position. When it is matched with any of element, then search is stopped and prints the element with its position.
- If list is unsorted, continue above step until end of list is reached or number is found.
- If list is sorted, continue above step until number is found or an element is found which is greater than element being searched.
- We will see step by step process of linear search in unsorted array.



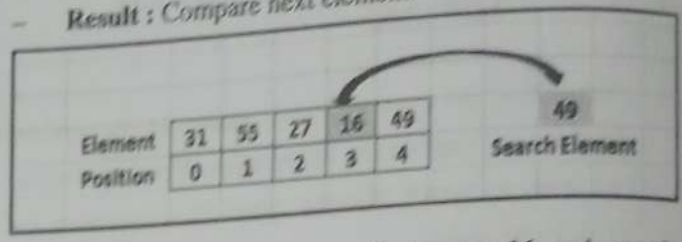
- **Comparison between :** 0^{th} element **31** and search element **49.**
- **Conclusion :** 31 != 49
- **Result :** Compare next element



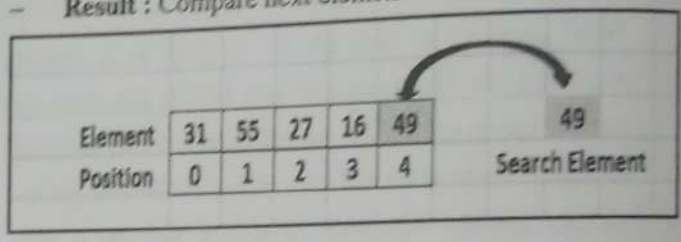
- **Comparison between :** 1^{st} element 55 and search element **49.**
- **Conclusion :** 55 != 49
- **Result :** Compare next element.



- **Comparison between :** 2^{nd} element 27 and search element **49.**
- **Conclusion :** 27 != 49
- **Result :** Compare next element.



- **Comparison between :** 3^{rd} element **16** and search element **49.**
- **Conclusion :** 16 != 49
- **Result :** Compare next element.



- **Comparison between :** 4^{th} element 49 and search element **49.**
- **Conclusion :** 49 = 49
- **Result :** Print the element and position and stop the search.

2.3.1 Algorithm of Linear Search

UQ. 2.3.2 Write pseudo C algorithm for :
(i) Linear search **SPPU - Dec.16, 3 Marks**

1. Accept element from user which you want to search.
2. Compare the search element with the 0^{th} location element in the list.
3. If both are equal then search is successful. Stop the search
4. If both are not equal then next element in the list is compared with search element.
5. Repeat step 3 and 4 until search element is compared with last element.
6. If none of element matches with element to search, then display message number is not found.

2.3.2 Complexity Analysis of Linear Search

GQ. 2.3.3 State complexity of linear search. **(2 Marks)**

Total numbers of comparisons' are N,

Time complexity

- Best case = O(1)
- Average case = n(n+1)/2n = O(n)
- Worst case = O(n)

Space complexity

O(1)

2.3.3 Program on Linear Search

GQ. 2.3.4 *Write a program to implement linear search for 10 elements in an array.* **(6 Marks)**

```
#include <iostream.h>
int main()
{
 int a[100], search, i, n;
 cout<<"\n Enter the size of array : ";
 cin>>n;
 cout<<"\n Enter "<<n<<" numbers : ";
 for (i = 0; i < n; i++)
  cin>>a[i];

cout<<"\n Enter the number you want to search : ";
 cin>>search;
 for (i = 0; i < n; i++)
 {
  if (a[i] == search)
  {
   cout<<search <<" is present at location: "<<i+1;
   break;
  }
 }
 if (i == n)
 cout<<search<<" is not present in array";
 return 0;
}
```

Accepts array elements.

Compare given element with array elements

Up to end if element not found, prints element not found

Output

```
C:\linear_search.exe
Enter the size of array : 5
Enter 5 numbers : 31 55 27 16 49
Enter the number you want to search : 49
49 is present at location: 5.
```

Explanation

- Here program first accepts the size of array and array elements from user.
- Then accept a number which to be searched.
- If given number is matched with array element then print number is found and its location.
- If none of element matches with element to search, then display message number is not found.

2.3.4 Advantages of Sequential Search

1. The linear search is simple (easy for understand).
2. This algorithm is applied on both sorted and unsorted list.

2.3.5 Disadvantages of Sequential Search

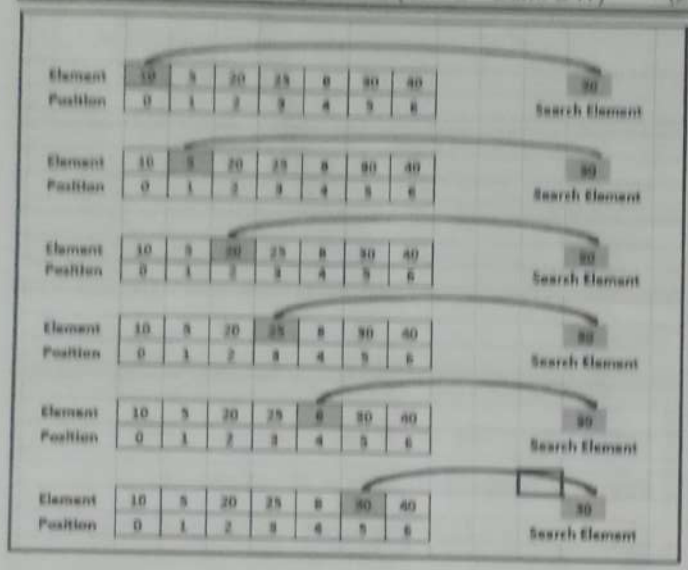
1. This search is not efficient when list is large.
2. Maximum number of comparisons are N. (when list contain n elements).

2.3.6 Examples on Sequential Search

GQ. 2.3.5 *Find position of element 30 using linear search algorithm in given sequence.*

10, 5, 20, 25, 8, 30, 40 **(3 Marks)**

Unit II In Se



30 found at location (5+1) = 6.

Syllabus Topic : Binary Search

2.4 BINARY SEARCH

GQ. 2.4.1 *Explain binary search with appropriate example.* **(4 Marks)**

- **Concept** : Binary search is very fast searching technique.
- This algorithm can be applied only on sorted list.
- Mid = $\frac{\text{Lower + Upper}}{2}$ this formula is used to find the mid of array.
- In this algorithm, given element is compared with middle element of the list, if these two are equal then prints element is found (search is successful). Otherwise, the list is divided into two parts. First part contains elements from first to mid-1, another part contains elements from mid+1 to last element in list.
- If given element is less than middle element then continue searching in first part otherwise in the second part.
- Repeat above steps till element is found or the division of part gives only one element.

Example

Now consider an array arr[7] having 5 elements

- int arr[ ] = {5,17,21,25,47,73,92}
- Search element 17.

	L			M			U
Element	5	17	21	25	47	73	92
Position	0	1	2	3	4	5	6

Here Lower = 0 and Upper = 6

$$\text{Mid} = \frac{\text{Lower + Upper}}{2} = \frac{0+6}{2} = 3$$

Checks 25 = = 17 no, here 17 < 25 hence search is continued in first part of list.

Then

$$\text{Upper} = \text{mid} - 1 = 3 - 1 = 2$$

Again calculate mid

$$\text{Mid} = \frac{\text{Lower + Upper}}{2} = \frac{0+2}{2}$$

$$\text{Mid} = 1$$

	L	M	U				
Element	5	17	21	25	47	73	92
Position	0	1	2	3	4	5	6

Here 17==17 the search is successful.

2.4.1 Algorithm of Binary Search

UQ. 2.4.2 *Write pseudo C algorithm for : Binary search.* **SPPU - Dec. 16, Dec. 18, 4 Marks**

1. Accept element from user which to be searched.
2. Search element is compared with mid element of the list, if these two are equal then the element is found (search is successful).
3. Otherwise, the list is divided into two parts. First part contains first element to mid-1 element, another part contain mid+1 to last element in list.

4. If search element is less than mid then continue searching in first part otherwise in the second part.
5. Repeat step 2, 3 and 4 until element is found or the division of part gives only one element.

2.4.2 Analysis of Binary Search

GQ. 2.4.3 *Give complexity for the following : Binary search.* **(1 Mark)**

- If list contain N elements then N/2 comparisons are required.
- Then the complexity of Binary search is **O(N/2).**

2.4.3 Program on Binary Search

UQ. 2.4.4 *Write a C++ function for the binary search.* **SPPU - Dec.16, 4 Marks**

```
#include <iostream.h>
int main()
{
Binary_search();
  return 0;
}
void Binary_search() // function for binary search
{
  int k, lower, upper, mid, n, search, arr[50];
  cout<<"\n Enter size of array : ";
  cin>>n;
  cout<<"\n Enter "<<n<<elements : ";
  for (k = 0; k < n; k++)
  {
    cin>>arr[k];
  }
  cout<<"\n Enter element you want to search : ";
  cin>>search;
    lower = 0;
  upper = n - 1;
  mid = (lower+upper)/2;
  while (lower <= upper) {
if (arr[mid] < search)
    lower = mid + 1;
else if (arr[mid] == search)
  {
cout<<search<<" found at location : "<< mid+1;
    break;
  }
  else
    upper = mid - 1;
  mid = (lower + upper)/2;
 }
 if (lower > upper)
  cout<<search<<" Not found in the list ";
}
}
```

Accepts array elements

Setting lower, upper and mid positions

If given element is greater than middle element then continue searching in second part.

If given element is equal to mid element then print number is found.

Output

```
C:\binary_search.exe
Enter size of array : 7
Enter 7 elements : 5 17 21 25 47 73 92
Enter element you want to search : 17
17 found at location : 2
```

Unit II In Sem.

2.5.2 PROGRAM TO IMPLEMENT FIBONACCI SEARCH

GQ. 2.5.2 *Write a program to implement Fibonacci Search.* **(6 Marks)**

```
#include <iostream.h>
 // Utility function to find minimum of two elements
int findminimum(int x, int y) { return (x<=y)? x : y; }

/* Returns index of x if present, else returns -1 */
int SearchFibo(int myarray[], int x, int n)
{
  /* Initialize fibonacci numbers */
  int fibMMm2 = 0;  // (m – 2)'th Fibonacci No.
  int fibMMm1 = 1;  // (m – 1)'th Fibonacci No.
  int fibM = fibMMm2 + fibMMm1; // m'th Fibonacci

  /* fibM is going to store the smallest Fibonacci
    Number greater than or equal to n */
  while (fibM < n)
  {
    fibMMm2 = fibMMm1;
    fibMMm1 = fibM;
    fibM  = fibMMm2 + fibMMm1;
  }
  // Marks the eliminated range from front
  int offset = – 1;

  /* while there are elements to be inspected. Note that
    we compare myarray[fibMm2] with x. When fibM
becomes 1,
    fibMm2 becomes 0 */
  while (fibM > 1)
  {
    // Check if fibMm2 is a valid location
    int i = findminimum(offset+fibMMm2, n – 1);
    /* If x is greater than the value at index fibMm2,
      cut the subarray array from offset to i */
    if (myarray[i] < x)
    {
      fibM  = fibMMm1;
      fibMMm1 = fibMMm2;
      fibMMm2 = fibM – fibMMm1;
      offset = i;
    }
    /* If x is greater than the value at index fibMm2,
      cut the subarray after i+1  */
    else if (myarray[i] > x)
    {
      fibM  = fibMMm2;
      fibMMm1 = fibMMm1 – fibMMm2;
      fibMMm2 = fibM – fibMMm1;
    }

    /* element found. return index */
    else return i;
  }

  /* comparing the last element with x */
  if(fibMMm1 && myarray[offset+1]==x)return offset+1;

  /*element not found. return -1 */
  return -1;
}
```

```
/* driver function */
int main()
{
  int myarray[] = {10, 22, 35, 40, 45, 50, 80, 82,
          85, 90, 100};
  int n = sizeof(myarray)/sizeof(myarray[0];
  int x = 85;
  cout<<"Found at index: "<<SearchFibo(myarray, x, n));

}
```

Output

```
Found at index: 8
```

Explanation

To understand the algorithm we will consider below example :

i	1	2	3	4	5	6	7	8	9	10	11	12	13
ar[i]	10	22	35	40	45	50	80	82	85	90	100	–	–

- **Consideration :** 1-based indexing. X which is the target is 85.
- Length of array myarray is = 11 (n).
- Smallest Fibonacci number which is equal to or greater than 11 is 13. As per our illustration:

  fibMm2 = 5

  fibMm1 = 8

  fibM = 13.

- Additional implementation detail is the offset variable which is initialized to 0. The eliminated range is marked by it starting from the front. It will be updated time to time.
- Now as the offset value is an index and there is elimination of all indices including it and below it, there is only makes sense to add something to it.
- Since approximately one-third of the array is marked by the fibMm2, and also the indices it marks are sure to be valid ones, it is possible to add fibMm2 to offset and verify the element which is at index i = min(offset + fibMm2, n).

Unit II In Sem.

fibMm2	fibMm1	fibM	offset	1 = min (offset + fibLn)	arr[1]	Consequence
5	8	13	0	5	45	Move one down, reset offset
3	5	8	5	8	81	Move one down, reset offset
2	3	5	8	10	90	Move two down
1	1	2	8	9	85	Return 1

- **Example :** [p, q, r] is sorted alphabetically, [1, 2, 3] is a list of integers sorted in ascending order, and [3, 2, 1] is a list of integers sorted in descending order.

2. Efficiency of Sorting Algorithm

GQ. 2.7.1 *Explain efficiency of sorting algorithm.* **(2 Marks)**

- **Concept :** In the process of sorting algorithm, the efficiency is always considered as an important issue. To add value to the sorting algorithms, we have to efficiently sort the records.

> *Definition of Efficiency of sorting algorithm : Efficiency of sorting algorithm is nothing but the time taken by the sorting algorithm for sorting purpose.*

- Hence the efficiency of sorting is denoted in terms of time complexity. As we have seen in previous chapter, Big -oh notation is used to represent time complexity.
- In general for various algorithms, time complexities available are $O(n^2)$ and O(nlogn).
- The time complexity $O(n^2)$ is basically related with sorting techniques such as insertion sort, shell sort and bubble sort while the time complexity O(nlogn)is related with sorting techniques such as quick sort and merge sort.
- The bubble sort is considered as the slowest while the quick sort is considered as fastest sorting algorithm.
- The number of records to be sorted also affects the efficiency.

3. Number of Passes

In the process of sorting elements in some specific sequence, the elements are arranged in number of ways.

> *Definition of Passes : Passes are the phases in which elements are shifted to get the exact position.*

For example : 1, 3, 2, 5, 4

Pass I : 1, 3, 2, 4, 5

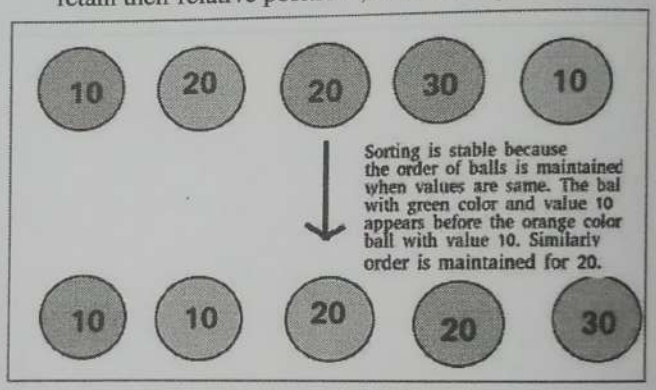
Pass II : 1, 2, 3, 4, 5

- Here we can observe that two passes are used to sort the elements in ascending order.
- In first pass, elements 4 and 5 are placed at their proper place while in the second pass elements 2 and 3 are placed at their proper place.

4. Sort Stability

UQ. 2.7.2 *Explain sort stability.* **SPPU - May 17, 2 Marks**

- A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.
- Informally, stability means that equivalent elements retain their relative positions, after sorting.



Syllabus Topic : Sorting Methods

2.8 SORTING METHODS

GQ. 2.8.1 *State any four sorting techniques.* **(2 Marks)**

- **Concept :** As we have already seen that Sorting is a technique which is used to rearrange the data elements of a list in a standard form of ascending or descending order, which may be numerical, lexicographical, or any user-defined order.
- After data storage, various operations are performed on data as per requirement such as insertion, deletion or modification in the records. After every such operation, we need to sort the data.

- Data structure provides various techniques to sort the data in required order :

  (1) Bubble Sort (2) Selection Sort

  (3) Insertion Sort (4) Merge Sort

  (5) Quick sort (6) Bucket Sort

Syllabus Topic : Bubble sort

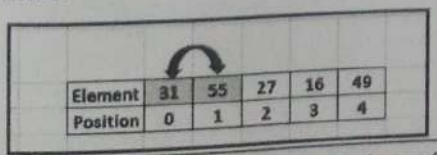
2.9 BUBBLE SORT

GQ. 2.9.1 *Give the principle of bubble sort.* **(2 Marks)**

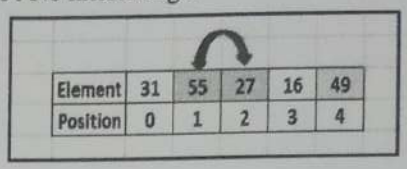
- **Concept :** There are few **principals** of bubble sort.
- **Principals :** Bubble sort algorithm divides the input list into two parts :
  - Sorted sublist.
  - Unsorted sublist.
- Initially, the sorted sublist is empty and the unsorted sublist is the entire input list.
- In Bubble sort, every element is compared with its next element and then swapping is performed if first element is greater than next element.
- After completion of first iteration largest element in given list is placed in sorted sublist.
- After completion of second iteration second largest element in given list is placed in sorted sublist and so on.
- The process is repeated in unsorted list until all the elements get sorted.
- **Example :** Now consider an array arr[5] having 5 elements :

```
int arr[] = {31,55,27,16,49}
```

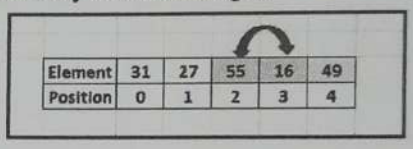
☞ **Iteration I**



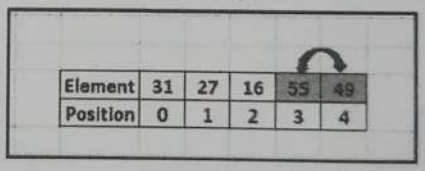
- **Comparison between :** 0th element **31** and 1st element **55.**
- **Conclusion :** 31 < 55
- **Result :** No interchange.



- **Comparison between :** 1st element 55 and 2nd element 27.
- **Conclusion :** 55 > 27
- **Result :** They are interchanged.

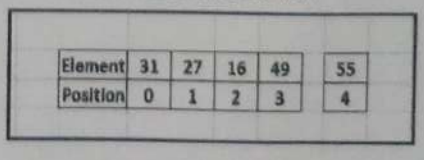


- **Comparison between :** 2nd element **55** and 3rd element **16.**
- **Conclusion :** 55 > 16
- **Result :** They are interchanged.



- **Comparison between :** 3rd element 55 and 4th element 49.
- **Conclusion :** 55 > 49
- **Result :** They are interchanged.

After completion of first iteration, given list is divided into two sublists, first one is unsorted sublist and second is sorted sublist as follows :



Unit II In Sem.

GQ. 2.9.5 *Write a program to sort an array of ten elements with bubble sort.* **(7 Marks)**

```
#include <iostream.h>
int main()
{
 int arr[10], n, i, j, temp;
   cout<<"Enter 10 elements : ";
 for (i = 0; i < 10; i++)
 cin>>arr[i];
  for (i = 0 ; i < ( 10 - 1 ); i++)
 {
  for (j = 0 ; j < 10 - i ; j++)
  {
   if (arr[j] > arr[j+1])
   {
    temp    = arr[j];
    arr[j]  = arr[j+1];
    arr[j+1] = temp;
   }
  }
 }

 cout<<"Sorted elements are : ";
 for ( i = 0 ; i < 10 ; i++ )
 {
   cout<< arr[i];
 }

}
```

Prints sorted elements

Output

```
C:\dd1.exe
Enter 10 elements : 1 8 7 9 6 5 3 4 2 10
Sorted elements are : 1 2 3 4 5 6 7 8 9 10
```

Syllabus Topic : Insertion Sort

2.10 INSERTION SORT

GQ. 2.10.1 *Explain in brief insertion sort.* **(4 Marks)**

- **Concept :** In this sorting algorithm, the first iteration starts with comparison of 1st position element with the 0th position element.
- If the 1st position element is found less than the 0th position element, then 1st position element is inserted at 0th location and 0th position element is shifted one position to right.
- In the every iteration, we will compare every element with all its previous elements and swapped with the element having less value.
- **Example :** Now consider an array arr[5] having 5 elements :

```
int arr[ ] = {31,55,27,16,49}
```

☞ **Iteration I**

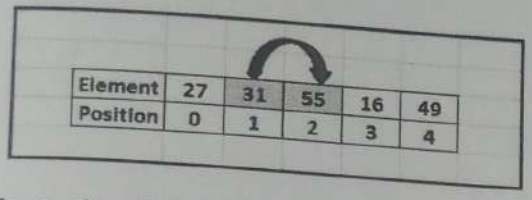
Element	31	55	27	16	49
Position	0	1	2	3	4

- **Comparison between :** 1st element 55 and 0th element 31.
- **Conclusion :** 55 > 31
- **Result :** No interchange.

☞ **Iteration II**

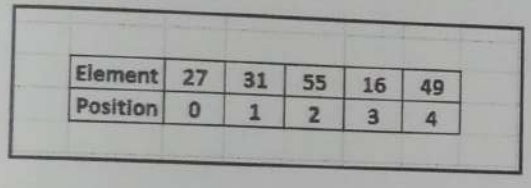
Element	31	55	27	16	49
Position	0	1	2	3	4

- **Comparison between :** 2nd element 27 and 0th element 31.
- **Conclusion :** 27 < 31
- **Result :** From 0th position element 31, all elements are shifted one position right and 27 is inserted at 0th location.



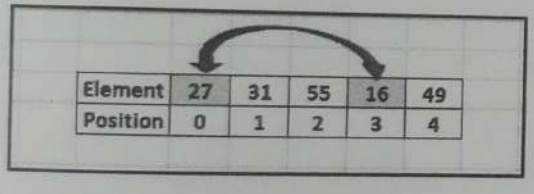
Element	27	31	55	16	49
Position	0	1	2	3	4

- **Comparison between :** 2nd element 55 and 1st element 31.
- **Conclusion :** 55 > 31
- **Result :** No interchange.
- After completion of second iteration elements will be as follows :



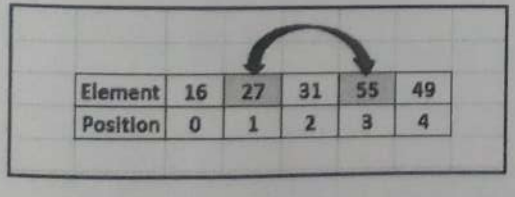
Element	27	31	55	16	49
Position	0	1	2	3	4

☞ **Iteration III**



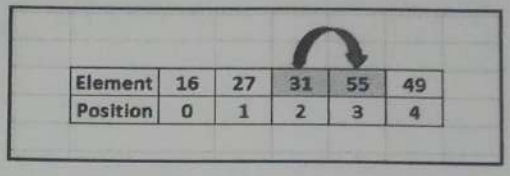
Element	27	31	55	16	49
Position	0	1	2	3	4

- **Comparison between :** 3rd element **16** and 0th element **27.**
- **Conclusion :** 16 < 27
- **Result :** From 0th position element 27, all elements are shifted one position right and 16 is inserted at 0th location.



Element	16	27	31	55	49
Position	0	1	2	3	4

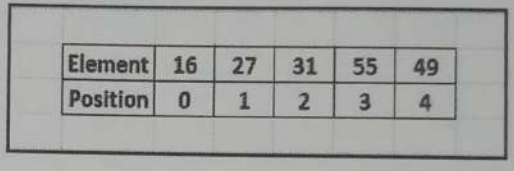
- **Comparison between :** 3rd element **55** and 1st element **27.**
- **Conclusion :** 55 > 27
- **Result :** No action



Element	16	27	31	55	49
Position	0	1	2	3	4

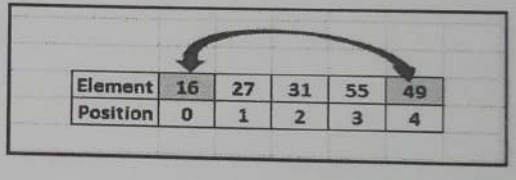
- **Comparison between :** 3rd element **55** and 2nd element **31.**
- **Conclusion :** 55 > 31
- **Result :** No action

At the end of iteration III, the list is:



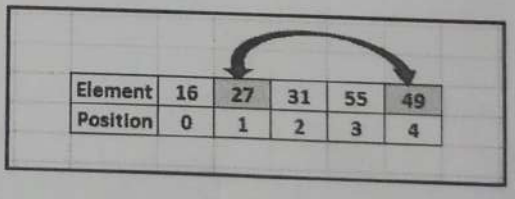
Element	16	27	31	55	49
Position	0	1	2	3	4

☞ **Iteration IV**



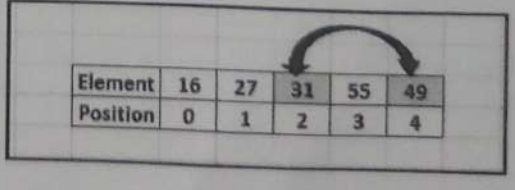
Element	16	27	31	55	49
Position	0	1	2	3	4

- **Comparison between :** 4th element **49** and 0th element **16.**
- **Conclusion :** 49 > 16
- **Result :** No action



Element	16	27	31	55	49
Position	0	1	2	3	4

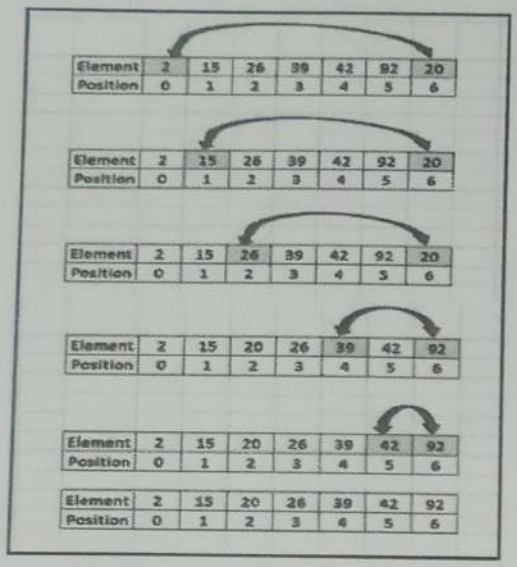
- **Comparison between :** 4th element **49** and 1st element **27.**
- **Conclusion :** 49 > 27
- **Result :** No action



Element	16	27	31	55	49
Position	0	1	2	3	4

Unit II In Sem.

Iteration VI



2.10.4 Analysis of Insertion Sort

Total numbers of comparisons' are :

$= (n-1) + (n-2) + ...3+2+1$

$= n(n-1)/2$

$= O(n^2)$

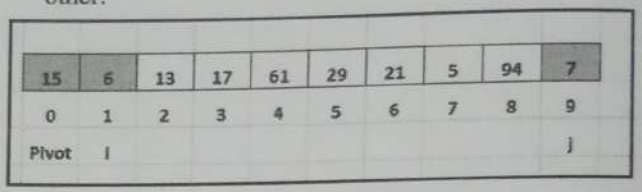
Space complexity: O(1)

Syllabus Topic : Quick Sort

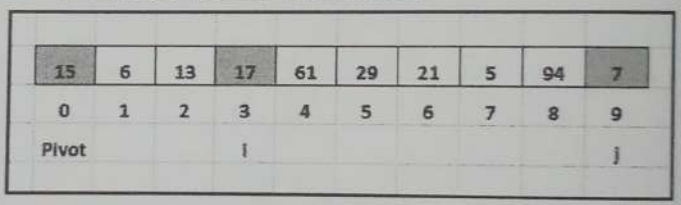
2.11 QUICK SORT

GQ. 2.11.1 *Explain quick sort.* **(2 Marks)**

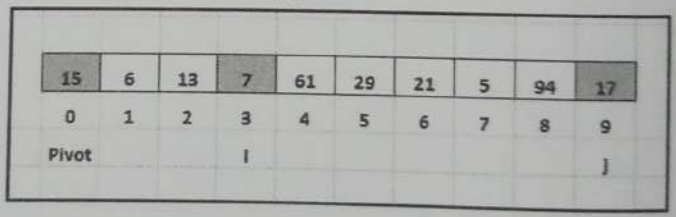
- As name suggest, **quick sort** is considered as the fastest sorting algorithm.
- In first iteration we set the first element (0^{th} location element) as pivot element. In this iteration we have to place pivot at its correct position.
- Here we require two index variables, i and j. The variable i points to 1^{st} location element and j points to $(n-1)^{th}$ location element.
- The working of i variable is to search an element which is greater than pivot element. Here i will be incremented by 1 till greater element than pivot is not found.
- The working of j variable is to search an element which is less than pivot element. Here j will be decremented by 1 till small element than pivot is not found.
- When these two elements are found, they are swapped.
- This process will be repeated until i and j crossed each other.



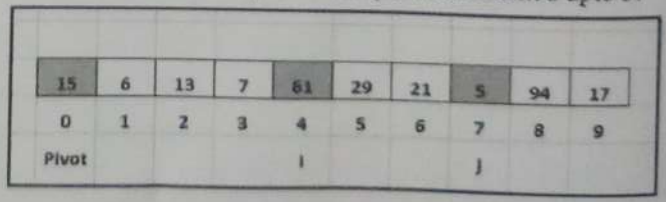
- Here the first element 15 is pivot element, i points to 6 and j points to 7.
- Now i try to find element greater than 15 (pivot), hence it will be incremented upto 17. j try to find element less than 15, as the current element 7 is less than 15 , it will not decremented.



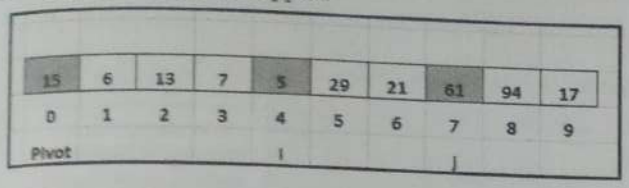
- 17 and 7 will be swapped.



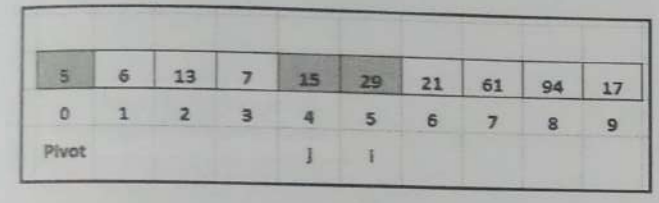
- Now again the i will be incremented to find element greater than pivot and j will be decremented to find element less than pivot.
- i is incremented upto 61 and j is decremented upto 5.



- 61 and 5 will get swapped.



- Now again i will be incremented and j will be decremented.
- i will be set to 29 and j will be set to 5.
- But at this situation, i and j cross each other.
- Hence now value of pivot element is swapped with value of j.



- Here **5** becomes the pivot element. We can observe that array is divided into two parts: all the elements before 15 are less than it and all the elements next to it are greater than it.
- Now same procedure is applied for two sub arrays. This will be repeated until all the arrays left with single element. At that situation all the elements will be in sorted form.

2.11.1 Algorithm of Quick Sort

GQ. 2.11.2 *Write an algorithm to implement Quick sort.* **(4 Marks)**

1. Lowest index element set as pivot element.
2. Take two index variable, i and j. i points to 1st location element and j points to (n-1)th location element.
3. Index variable i is in search of element which is greater than pivot element. Here i will incremented by 1 till greater element is not found.
4. Index variable j is in search of element which is less than pivot element. Here j will be decremented by 1 till small element is not found.
5. If these two elements are found, they are swapped.
6. The process ends when these two variables are crossed or meet (In above example they are crossed). Then value at index j is swapped with pivot and list is divided into 2 sublists.
7. Above steps are repeated on these two sub arrays (sublists) until all sub arrays contain only 1 element.

Unit II In Sem.

2.11.2 Analysis of Quick Sort

GQ. 2.11.3 *Discuss time and space complexity of Quick sort.* **(2 Marks)**

- **Best case :** O(n log n)
- **Worst case :** $O(n^2)$
- Quick sort is recursive algorithm.
- Time taken by quick sort = running time of right partition + running time of left partition + time for partitioning
- Here first two terms are for recursive call and last term is for partition.
- Array contains n elements. Suppose that left partition contains k elements then running time of left partition = T(k).
- Therefore running time of right partition= T(n-k-1).
- For partition, needs scanning of array linearly = cn

$T(n) = T(k) + T(n - k - 1) + cn.$ (Here c is constant)

$T(n) = T(n - 1) + cn.$

Space complexity : O(log(n))

2.11.3 Advantages and Disadvantages of Quick Sort

GQ. 2.11.4 *State advantages and disadvantages of quick sort.* **(4 Marks)**

Advantages

1. This sorting algorithm is useful when list is small as well as long.
2. Best case time complexity is O(n log n).

Disadvantage

1. Quick sort algorithm is not stable.

2.11.4 Program on Quick sort

UQ. 2.11.5 *Write program to implement Quick Sort.)* SPPU - Dec.18, 4 Marks

```
#include <iostream.h>

void quicksort(int [10],int,int);

int main(){
 int x[20],size,i;

 cout<<"Enter size of the array: ";
 cin>>size;

 cout<<"Enter "<<size<<" elements: ";
 for(i=0;i<size;i++)
  cin>>x[i];
 quicksort(x,0,size-1;

 cout<<"Sorted elements: ";
 for(i=0;i<size;i++)
  cout<<x[i];

}

void quicksort(int x[10],int first,int last){
  int pivot,j,temp,i;

  if(first<last){
     pivot=first;
     i=first;
     j=last;
while(i<j)
{
while(x[i]<=x[pivot]&&i<last)
i++;
while(x[j]>x[pivot])
j--;
if(i<j){
temp=x[i];
x[i]=x[j];
x[j]=temp;
        }
      }
      temp=x[pivot];
      x[pivot]=x[j];
      x[j]=temp;
      quicksort(x,first,j-1);
      quicksort(x,j+1,last);
  }
}
```

i is incremented to find larger than pivot and j is decremented to find smaller than pivot

Pivot and j positions elements are swapped

Output

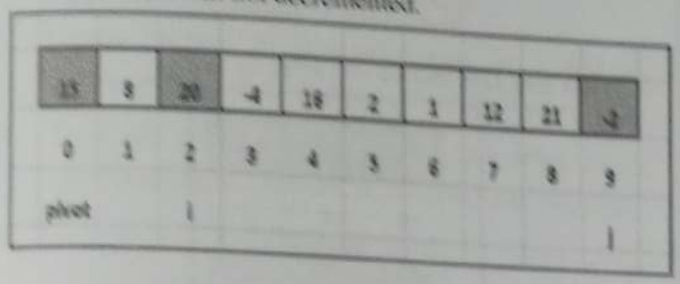
```
C:\q.exe
Enter size of the array: 5
Enter 5 elements: 3 1 2 5 4
Sorted elements:  1 2 3 4 5
```

2.11.5 Examples of Quick Sort

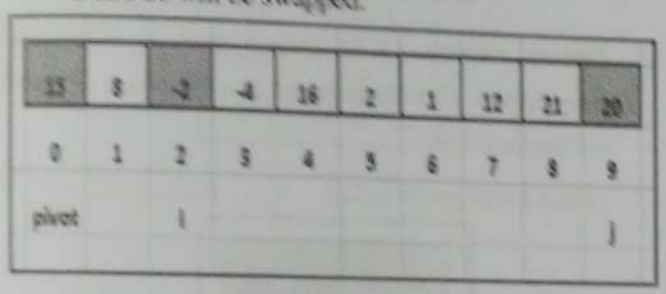
GQ. 2.11.6 *Sorts the given list using quick sort: 15, 08, 20, – 4, 16, 02, 01, 12, 21, -2.* **(4 Marks)**

15	8	20	-4	16	2	1	12	21	-2
0	1	2	3	4	5	6	7	8	9
pivot	i								j

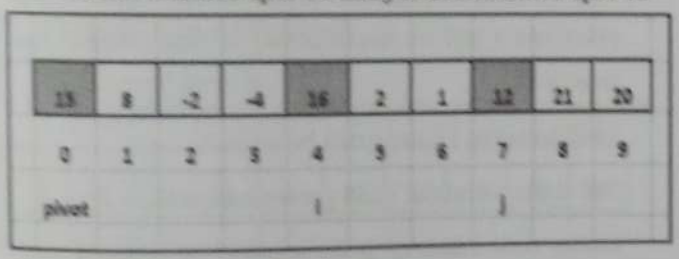
- Here the first element 15 is pivot element, i points to 8 and j points to – 2.
- Now i try to find element greater than 15 (pivot), hence it will be incremented upto 20. j try to find element less than 15, as the current element -2 is less than 15, it will not decremented.



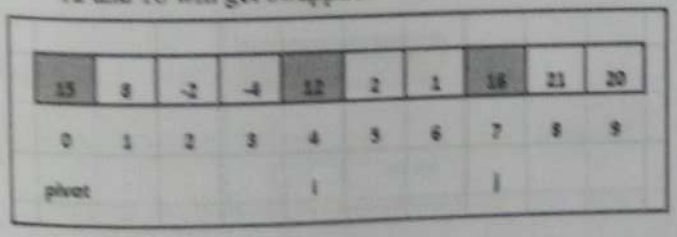
- – 2 and 20 will be swapped.



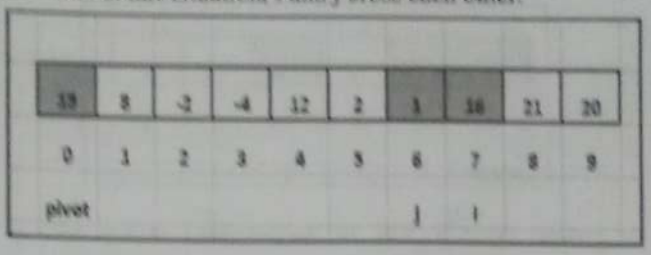
- Now again the i will be incremented to find element greater than pivot and j will be decremented to find element less than pivot.
- i is incremented upto 16 and j is decremented upto 12



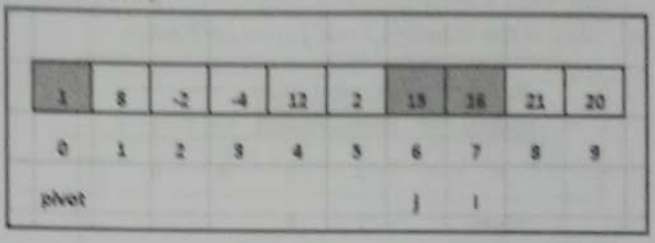
- 12 and 16 will get swapped.



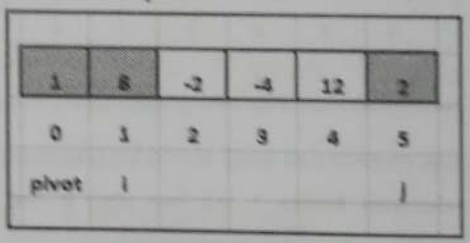
- Now again i will be incremented and j will be decremented.
- i will be set to 16 and j will be set to 1.
- But at this situation, i and j cross each other.



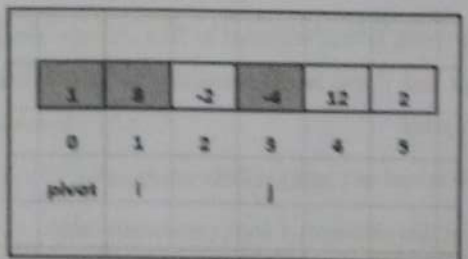
- Hence now value of pivot element is swapped with value of j.



- We can observe that array is divided into two parts: all the elements before 15 are less than it and all the elements next to it are greater than it.
- Now same procedure is applied for two sub arrays. Our first sub array is :

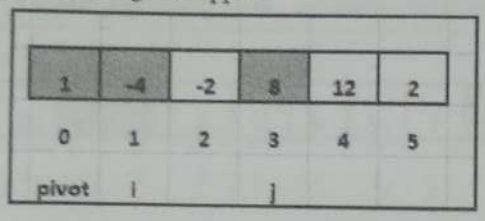


- Here the first element 1 is pivot element, i points to 8 and j points to 2.
- Now i will be incremented to find element greater than pivot and j will be decremented to find element less than pivot.
- Current element 8 is greater than 1, i will not incremented. j is set to – 4.

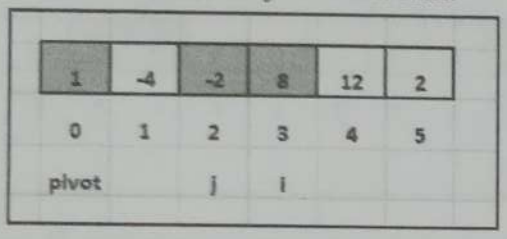


Unit II In Sem.

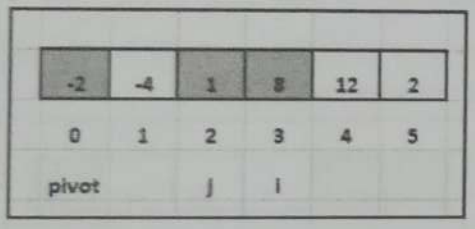
- 8 and – 4 will get swapped.



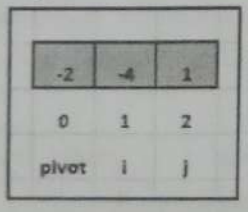
- Now again i will be incremented and j will be decremented
- i will be set to 8 and j will be set to -2.
- But at this situation, i and j cross each other.



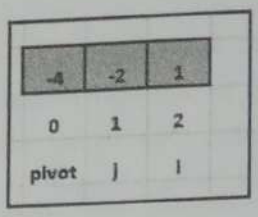
- Hence now value of pivot element is swapped with value of j.



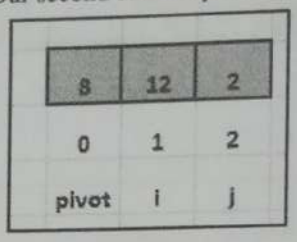
- We can observe that array is divided into two parts: all the elements before 1 are less than it and all the elements next to it are greater than it.
- Now same procedure is applied for two sub arrays. Our first sub array is:



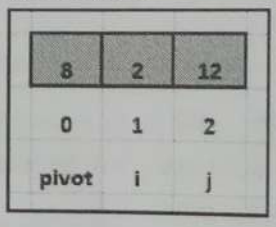
- Now i will be incremented to find element greater than pivot and j will be decremented to find element less than pivot
- i will be set to 1 and j will be set to – 4.
- But at this situation, i and j cross each other.
- Hence now value of pivot element is swapped with value of j.



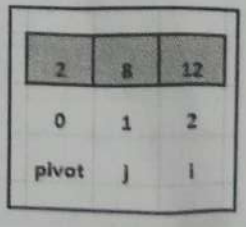
- Now take Our second sub array is:



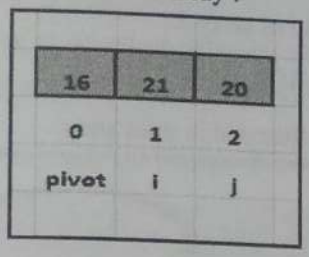
- Current element 12 is greater than 8, i will not incremented. Current element 2 is less than 8, j will not decremented.
- 12 and 2 will get swapped.



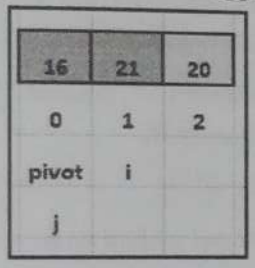
- Now i will be incremented to find element greater than pivot and j will be decremented to find element less than pivot
- i will be set to 12 and j will be set to 2.
- But at this situation, i and j cross each other.
- Hence now value of pivot element is swapped with value of j.



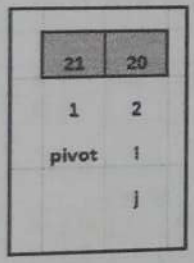
- Now take Our second sub array :



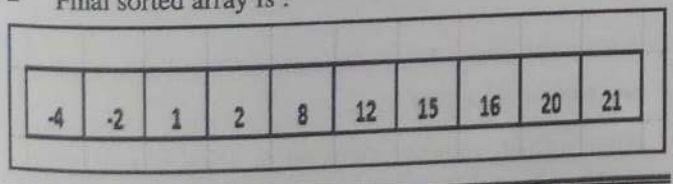
- Current element 21 is greater than 16, i will not incremented. j will be decremented and it set to 16.
- But at this situation, i and j cross each other. Hence now value of pivot element is swapped with value of j.



Now list contain



- 20 and 21 will get swapped.
- Final sorted array is :



Syllabus Topic : Merge Sort

2.12 MERGE SORT

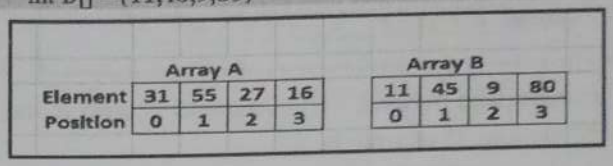
GQ. 2.12.1 *Explain Merge Sort.* **(4 Marks)**

- Merging is nothing but combining of two sorted arrays.
- This takes two arrays as input and produce third array as output in sorted format.

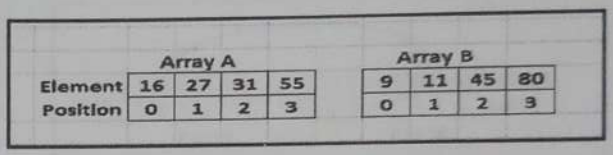
Example

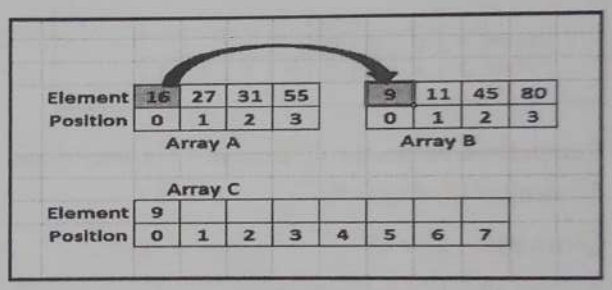
- Now consider two arrays having 4 elements :

```
int A[] = {31,55,27,16}
int B[]={11,45,9,80}
```

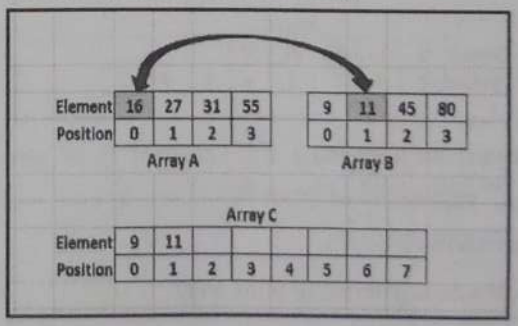


- Array A and B are sorted using any sorting algorithm.



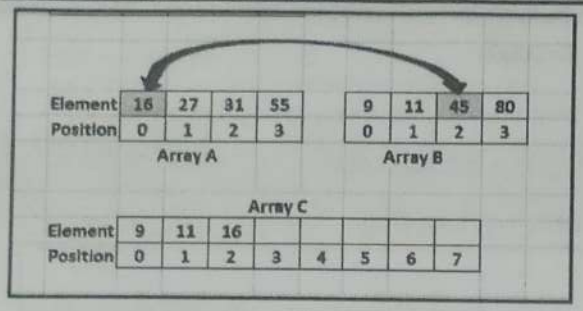


- **Comparison between :** 0^{th} element 16 of array A and 0^{th} element 9 of array B.
- **Conclusion :** $31 > 9$
- **Result :** 9 is inserted in third array.

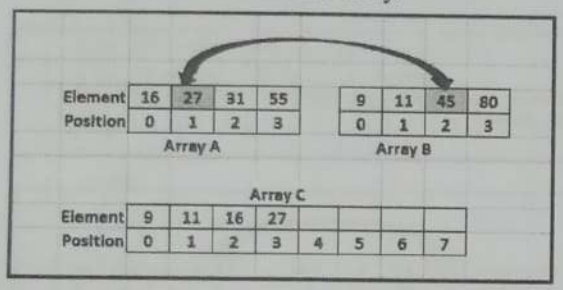


- **Comparison between :** 0^{th} element 16 of array A and 1^{st} element 11 of array B.
- **Conclusion :** $16 > 11$
- **Result :** 11 is inserted in third array.

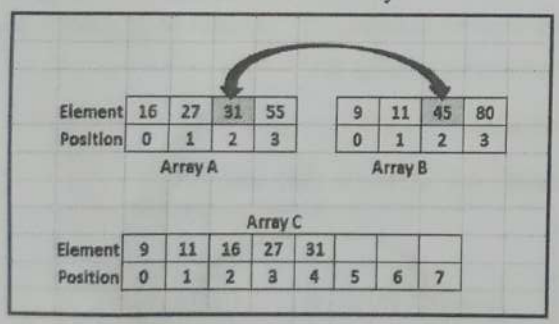
Unit II In Sem.



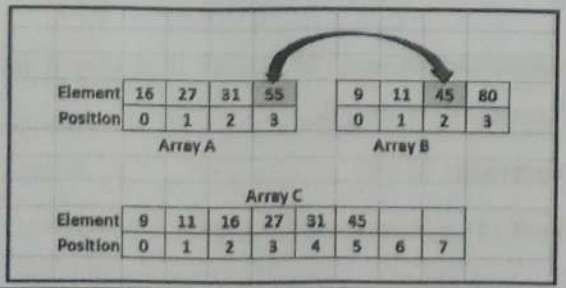
- **Comparison between :** 0^{th} element 16 of array A and 2^{nd} element 45 of array B.
- **Conclusion :** $16 < 45$
- **Result :** 16 is inserted in third array.



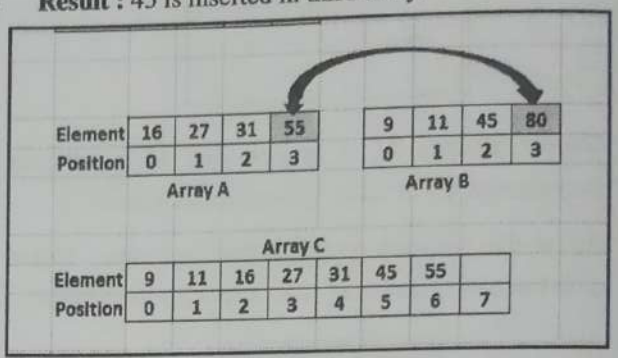
- **Comparison between :** 1^{st} element 27 of array A and 2^{nd} element 45 of array B.
- **Conclusion :** $27 < 45$
- **Result : 27** is inserted in third array.



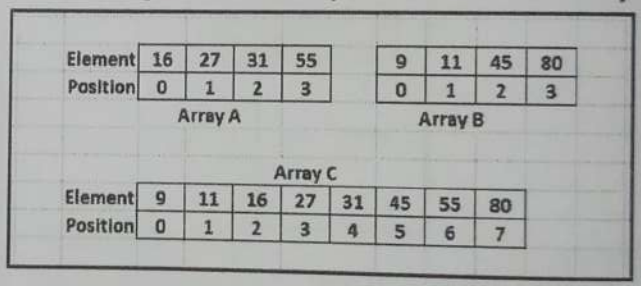
- **Comparison between :** 2^{nd} element 31 of array A and 2^{nd} element 45 of array B.
- **Conclusion :** $31 < 45$
- **Result :** 31 is inserted in third array.



- **Comparison between :** 3^{rd} element 55 of array A and 2^{nd} element 45 of array B.
- **Conclusion :** $55 > 45$
- **Result :** 45 is inserted in third array.



- **Comparison between :** 3^{rd} element 55 of array A and 3^{rd} element 80 of array B.
- **Conclusion :** $55 < 80$
- **Result :** 55 is inserted in third array.
- After this, elements in array A are exhausted, then the remaining elements in array B are added to third array.



2.12.1 Algorithm of Merge Sort

1. Accept two arrays from user and sort these arrays using any sorting algorithm.
2. After sorting, compare elements in array A with array B, If element in array A is less than array B then element in array A is inserted in array C and index variable of array A and array C are incremented by 1.
3. If Element in array A is greater than array B then element in array B is inserted in array C and index variable of array B and array C are incremented by 1.
4. The same procedure is repeated until end of one of the array is reached, then the remaining elements from another array are added to third array

2.12.2 Analysis of Merge Sort

Total number of passes required by merge sort (Time Complexity) : log n efficiency = O (n log n)

2.12.3 Program on Merge Sort

GQ. 2.12.2 *Write program to sort array elements using merge sort.* **(4 Marks)**

```
#include <iostream.h>
#include <conio.h>
int main( )
{
   int A[10];
   int B[10];
   int C[20];
   int i,j,k,n,m,l,temp;
   cout<<"\n Merge sort.\n" ;
   cout<<"\n Enter the size of array A :";
   cin>>n;
   cout<< "\n Enter elements of array A :" ;
   for ( i = 0 ; i < n ; i++ )
   {
      cin>> A[i] ;
   }
   cout<<"\n Enter the size of array B :";
   cin>>m;

   cout<< "\n Enter elements of array B :" ;
   for ( i = 0 ; i < m ; i++ )
   {
      cin>> B[i] ;
   }
```

Accepts first array elements

Accepts second array elements

```
   for ( k = 0 ; k <= n-2 ; k++ )
   {
      for ( l = k + 1 ; l <= n-1 ; l++ )
       {
         if ( A[k] > A[l] )
         {
            temp = A[k] ;
            A[k] = A[l];
            A[l] = temp;
         }
      }
   }
   for ( k = 0 ; k <= m-2 ; k++ )
   {
     for ( l = k + 1 ; l <= m-1 ; l++ )
      {
        if ( B[k] > B[l] )
        {
           temp = B[k] ;
           B[k] = B[l];
           B[l] = temp;
        }
          }
   }
   for ( i = j = k = 0 ; i < n+m ; )
   {
      if ( A[j] <= B[k] )
       C[i++] = A[j++] ;
      else
       C[i++] = B[k++] ;
```

Sort array A using selection sort.

Sort array B using selection sort.

If Element in array A is less than array B then element in array A is inserted in third array, else element in array B is inserted.

Unit II In Sem.

```
        if ( j == n || k == m )
            break;
        }
        for (; j < n ; )
            C[i++] = A[j++] ;

        for (; k < m ; )
        C[i++] = B[k++] ;

    cout<< "\n Array after sorting :" ;

for ( i = 0 ; i < n+m ; i++ )
    {
    cout<< "\t"<<C[i] ;
    }
    cout<<"\n\n";
    return 0;
}
```

If elements in array B are exhausted, then the remaining elements in array A are added to third array.

If elements in array A are exhausted, then the remaining elements in array B are added to third array.

Prints sorted elements

Output

```
Merge sort.
Enter the size of array A :4
Enter elements of array A :31 55 27 16
Enter the size of array B :4
Enter elements of array B :11 45 9 80
Array after sorting :  9    11    16    27    31    45    55
80
```

Explanation

- Here program accepts elements for arrays A and B.
- Sort these two arrays using selection sort.
- After sorting, compare elements in array A with array B.
- If element in array A is less than element of array B then it is inserted in array C and index variable of array A and C are incremented by 1.
- Else element in array B is inserted in array C and index variable of array B and C are incremented by 1.
- The same procedure is repeated until the end of one of the array is reached.
- Then the remaining elements from another array are added to third array.
- Finally prints the sorted elements.

2.12.4 Advantages and Disadvantages of Merge Sort

GQ. 2.12.3 *State advantages and disadvantages of merge sort.* **(4 Marks)**

Advantages

1. It can be used for internal and external sorting.
2. It is a stable sorting algorithm.
3. It very efficient method.

Disadvantage

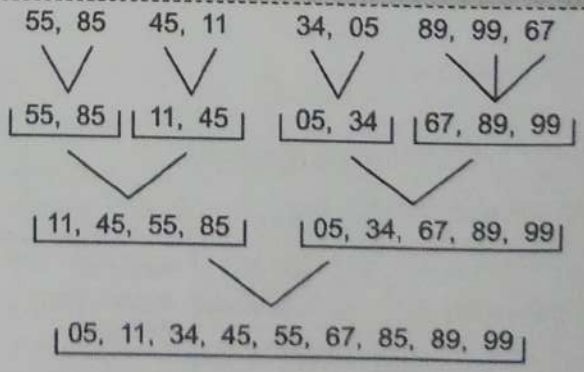
It needs an extra memory for storing of merged data.

2.12.5 Examples on Merge Sort

GQ. 2.12.4 *Sort the following numbers using merge sort:*

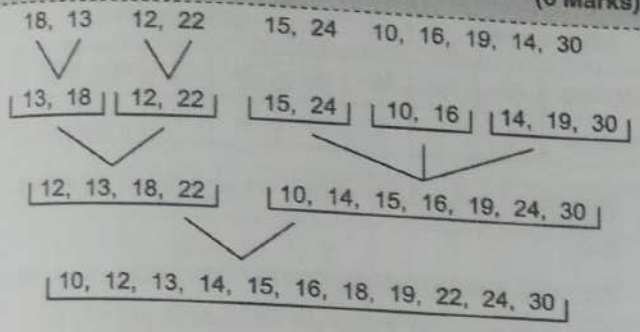
55, 85, 45, 11, 34, 05, 89, 99, 67.

(6 Marks)



GQ. 2.12.5 *Explain Merge sort using the following example:*

18, 13, 12, 22, 15, 24, 10, 16, 19, 14, 30. **(6 Marks)**



2.12.6 Analysis of Merge Sort

GQ. 2.12.6 *What is time-complexity of merge sort ?* **(1 Mark)**

Total number of passes required by merge sort (Time Complexity) : log n efficiency = O (n log n)

Space complexity: O(n)

Syllabus Topic : Shell Sort

2.13 SHELL SORT

GQ. 2.13.1 *Explain shell sort.* **(2 Marks)**

- Shell sort uses insertion sort to sort given array.
- In this sort, we require value of gap or interval.
- Initial value of gap = number of elements in array/2;

Initial list

Element	56	28	95	19	79	33	46	57
Position	0	1	2	3	4	5	6	7

Gap(d_1) = 8/2 = 4

- The list contains 8 elements and gap = 4, hence there are 4 sublists created as follows:
- Here gap=4 , hence mark on 1st element and 1+4=5th element, this is first sublist. {56 , 79}

Sublist 1

Element	56	28	95	19	79	33	46	57
Position	0	1	2	3	4	5	6	7

- For second sublist mark on 2nd element and 2 + 4 = 6th element, this is second sublist. {28 , 33}

Sublist 2

Element	56	28	95	19	79	33	46	57
Position	0	1	2	3	4	5	6	7

- For third sublist mark on 3rd element and 3+4=7th element, this is third sublist. {95 , 46}

Sublist 3

Element	56	28	95	19	79	33	46	57
Position	0	1	2	3	4	5	6	7

- For Fourth sublist mark on 4th element and 4+4 = 8th element, this is third sublist. {19, 57}

Sublist 4

Element	56	28	95	19	79	33	46	57
Position	0	1	2	3	4	5	6	7

- Now we compare and swap the elements if require.

Sorted Sublist 1

Element	56	28	95	19	79	33	46	57
Position	0	1	2	3	4	5	6	7

Sorted Sublist 2

Element	56	28	95	19	79	33	46	57
Position	0	1	2	3	4	5	6	7

Sorted Sublist 3

Element	56	28	46	19	79	33	95	57
Position	0	1	2	3	4	5	6	7

Sorted Sublist 4

Element	56	28	95	19	79	33	46	57
Position	0	1	2	3	4	5	6	7

Unit II In Sem.

After completion of first step array looks as follow:

Element	56	28	46	19	79	33	95	57
Position	0	1	2	3	4	5	6	7

- Now gap i.e $d_2=d_1/2=4/2=2$.
- Then two sublists are created, first is {56, 46, 79, 95} and second is {28, 19, 33, 57}.

Sublist 1

Element	56	28	46	19	79	33	95	57
Position	0	1	2	3	4	5	6	7

Sublist 2

Element	56	28	46	19	79	33	95	57
Position	0	1	2	3	4	5	6	7

Sorted sublist 1

Element	46	28	56	19	79	33	95	57
Position	0	1	2	3	4	5	6	7

Sorted sublist 2

Element	56	19	46	28	79	33	95	57
Position	0	1	2	3	4	5	6	7

☞ **After sorting**

Element	46	19	56	28	79	33	95	57
Position	0	1	2	3	4	5	6	7

- Now gap i.e $d_3= d_2/2= 2/2=1$, we sort the array using gap.
- Select all elements starting from 0^{th} location, compare them with elements within the distance of gap i.e. 1.

Element	19	28	33	46	56	57	79	95
Position	0	1	2	3	4	5	6	7

- Finally the list is sorted.

Element	19	28	33	46	56	57	79	95
Position	0	1	2	3	4	5	6	7

2.13.1 Examples on Shell Sort

GQ. 2.13.2 *Sort the given list using shell sort : 08, 03, 02, 11, 05, 14, 00, 02, 09, 04, 20.* **(4 Marks)**

Element	08	03	02	11	05	14	00	02	09	04	20
Position	0	1	2	3	4	5	6	7	8	9	10

$Gap(d_1) = 11/2 = 5$

- The list contains 11 elements and gap = 5, hence there are 6 sublists created as follows:
- Here gap=5 , hence mark on 1^{st} element and $1+5=6^{th}$ element, this is first sublist. {08 , 14}

☞ **Sublist 1**

Element	08	03	02	11	05	14	00	02	09	04	20
Position	0	1	2	3	4	5	6	7	8	9	10

- For second sublist mark on 2^{nd} element and $2 + 5 = 7^{th}$ element, this is second sublist. {03 , 00}

☞ **Sublist 2**

Element	08	03	02	11	05	14	00	02	09	04	20
Position	0	1	2	3	4	5	6	7	8	9	10

- For third sublist mark on 3rd element and 3+5=8th element, this is third sublist. {02 , 02}

Sublist 3

Element	08	03	02	11	05	14	00	02	09	04	20
Position	0	1	2	3	4	5	6	7	8	9	10

- For Fourth sublist mark on 4th element and 4+5 = 9th element, this is third sublist. {11, 09}

Sublist 4

Element	08	03	02	11	05	14	00	02	09	04	20
Position	0	1	2	3	4	5	6	7	8	9	10

- For Fifth sublist mark on 5th element and 5+5 = 10th element, this is third sublist. {05, 04}

Sublist 5

Element	08	03	02	11	05	14	00	02	09	04	20
Position	0	1	2	3	4	5	6	7	8	9	10

- For Sixth sublist mark on 6+5 = 11th element, this is third sublist. {20}

Sublist 6

Element	08	03	02	11	05	14	00	02	09	04	20
Position	0	1	2	3	4	5	6	7	8	9	10

- Now we compare and swap the elements if require.

Sorted Sublist 1

Element	08	03	02	11	05	14	00	02	09	04	20
Position	0	1	2	3	4	5	6	7	8	9	10

Sorted Sublist 2

Element	08	00	02	11	05	14	03	02	09	04	20
Position	0	1	2	3	4	5	6	7	8	9	10

Sorted Sublist 3

Element	08	03	02	11	05	14	00	02	09	04	20
Position	0	1	2	3	4	5	6	7	8	9	10

Sorted Sublist 4

Element	08	03	02	09	05	14	00	02	11	04	20
Position	0	1	2	3	4	5	6	7	8	9	10

Sorted Sublist 5

Element	08	03	02	11	04	14	00	02	09	05	20
Position	0	1	2	3	4	5	6	7	8	9	10

Sorted Sublist 6

Element	08	03	02	11	05	14	00	02	09	04	20
Position	0	1	2	3	4	5	6	7	8	9	10

- After completion of first step array looks as follow:

Element	08	00	02	09	04	14	03	02	11	05	20
Position	0	1	2	3	4	5	6	7	8	9	10

- Now gap i.e $d_2=d_1/2=5/2=2$.
- Then two sublists are created, first is {08, 02, 04, 03, 11, 20} and second is {00, 09, 14, 02, 05}.

Sublist 1

Element	08	00	02	09	04	14	03	02	11	05	20
Position	0	1	2	3	4	5	6	7	8	9	10

Sublist 2

Element	08	00	02	09	04	14	03	02	11	05	20
Position	0	1	2	3	4	5	6	7	8	9	10

Sorted Sublist 1

Element	02	00	03	09	04	14	08	02	11	05	20
Position	0	1	2	3	4	5	6	7	8	9	10

Sorted Sublist 2

Element	08	00	02	02	04	05	03	09	11	14	20
Position	0	1	2	3	4	5	6	7	8	9	10

After sorting

Element	02	00	03	02	04	05	08	09	11	14	20
Position	0	1	2	3	4	5	6	7	8	9	10

- Now gap i.e $d_3= d_2/2= 2/2=1$, we sort the array using gap.
- Select all elements starting from 0^{th} location, compare them with elements within the distance of gap i.e. 1.

Element	00	02	02	03	04	05	08	09	11	14	20
Position	0	1	2	3	4	5	6	7	8	9	10

☞ **Finally the list is sorted**

Element	00	02	02	03	04	05	08	09	11	14	20
Position	0	1	2	3	4	5	6	7	8	9	10

2.13.2 Algorithm of Shell Sort

GQ. 2.13.3 *Write an algorithm to implement shell sort.* **(2 Marks)**

- Initialize the value of gap or interval as d/2.
- List is divided into sublists using gap.
- Sort these sublists using insertion sort.
- Repeat above steps until given list is sorted.

☞ **Analysis of shell sort**

Shell sort Time complexity is depend on gap sequence.

Complexity

- **Best case :** O (n)
- **Worst case :** $O(n^2)$

2.13.3 Program on Shell Sort

UQ. 2.13.4 *Write pseudo C/C++ code to perform Shell Sort.* SPPU - Dec. 18, 6 Marks

```
#include <iostream.h>
void shellsort(int arr[], int num)
{
  int i, j, k, tmp;
  for (i = num / 2; i > 0; i = i / 2)
  {
    for (j = i; j < num; j++)
    {
      for(k = j - i; k >= 0; k = k - i)
      {
        if (arr[k+i] >= arr[k])
          break;
        else
        {
          tmp = arr[k];
          arr[k] = arr[k+i];
          arr[k+i] = tmp;
        }
      }
    }
  }
}
int main()
{
  int arr[30];
  int k, num;
  cout<<"Enter total no. of elements : ";
  cin>>num;
  cout<<"\nEnter "<<num<<"numbers: ";
  for (k = 0 ; k < num; k++)
  {
    cin>>arr[k];
  }
  shellsort(arr, num);
  cout<<"\nSorted Elements : ";
  for (k = 0; k < num; k++)
    cout<<arr[k];
}
```

Output

```
C:\sh.exe
Enter total no. of elements : 5
Enter 5 numbers: 5 2 3 1 4
Sorted Elements : 1 2 3 4 5
```

Unit II In Sem.

Syllabus Topic : Comparison of All Sorting Methods

2.14 COMPARISON OF ALL SORTING METHODS

2.14.1 Difference between Bubble Sort and Insertion Sort

GQ. 2.14.1 *Compare bubble sort and insertion sort.* **(6 Marks)**

Basis for comparison	Bubble sort	Insertion sort
Basic	Adjacent element is compared and swapped	Inserting an element in the input list in to the correct position in a list.
Best case time complexity	Ω(n)	Ω(n)
Efficiency	Inefficient	Very Efficient
Stable	Yes	No
Method	Exchanging	Insertion
Speed	Slow	Fast as compared to bubble sort.

2.14.2 Difference between Merge Sort and Quick Sort

GQ. 2.14.2 *Differentiate between merge sort and quick sort.* **(2 Marks)**

Parameters	Merge Sort	Quick Sort
Stability	Merge sort algorithm is stable.	Quick sort algorithm is not stable.
Memory	Merge sort require more memory as compare to quick sort.	Quick sort require less memory as compare to merge sort.

Syllabus Topic : Analyze Insertion sort, Quick Sort, Binary Search, Hashing for Best, Worst and Average Case

2.14.3 Analyze Insertion sort, Quick Sort, Binary Search, Hashing for Best, Worst and Average Case

Algorithm	Analysis		
	Best	Average	Worst
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)
Quick Sort	Ω(n log(n))	θ(n log(n))	O(n^2)
Binary Search	O(1)	O(log n)	O(log n)

2.14.4 Selecting Sorting Technique

UQ. 2.14.3 *With example, discuss the criteria for choosing a sorting algorithm based on the input size and time complexity. [Trade-off bubble, insertion and quicksort.* **SPPU - May 19, 6 Marks**

- **Correctness :** This is the most important thing. It worth nothing if your algorithm is super fast and efficient, but is wrong.
- In sorting we need a stable sort. We have to select the stable sort algorithm, even if it is less efficient.
- Next are basically tradeoffs between running time, needed space and implementation time.
- Important elements to take consider when thinking about the trade off mentioned above:
- **Size of the input :** For example: for small inputs, insertion sort is comparatively faster).
- **Location of the input** : Sorting algorithms on disk are different from algorithms on RAM, because disk reads are much less efficient when not sequential. The algorithm which is usually used to sort on disk is a variation of merge-sort.

- How is the data distributed? If the data is likely to be "almost sorted" then bubble-sort can sort it in just 2-3 iterations and better than other algorithms.
- **Type (and range) of the input** - For enumerable data, an integer designed algorithm (like radix sort) might be more efficient then a general case algorithm.
- **Latency requirement** - if we required fast algorithm, quick-sort is always a better option.

Syllabus Topic : Case Study : Study and Analyze Selection Sort, Bucket Sort, Radix Sort

2.15 STUDY AND ANALYZE SELECTION SORT, BUCKET SORT, RADIX SORT

2.15.1 Selection Sort

- **Concept :** Selection sort is considered as the simplest method of sorting.
- The algorithm divides the input list into two parts: sorted sublist and unsorted sublist.
- Initially, the sorted sublist is empty and the unsorted sublist is the entire input list.
- The 0^{th} element is compared with all further elements. If 0^{th} element is found greater than next element, then they are interchanged.
- Hence at the end of first iteration, the smallest element is placed at 0^{th} position.
- Now process is repeated in unsorted part, i.e. from 1^{st} position.
- The 1^{st} element is compared with all next elements. If 1^{st} element is found greater than next element, then they are interchanged.
- Hence at the end of second iteration, the second smallest element is placed at 1^{st} position.
- The process is repeated until all the elements get sorted.

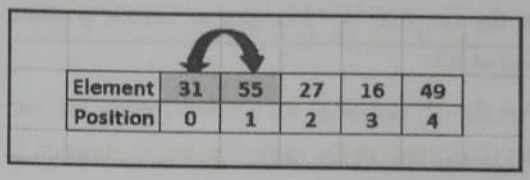
Example

- Now consider an array arr[5] having 5 elements :

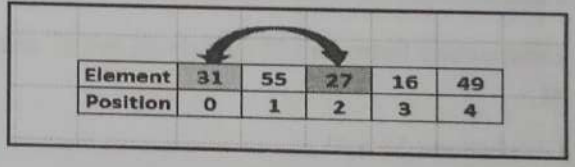
```
int arr[] = {31, 55, 27, 16, 49}
```

- In the first iteration, the element of first position arr[0] is compared with all the remaining elements.
- If the first element is found greater than the compared element, then they are swapped.

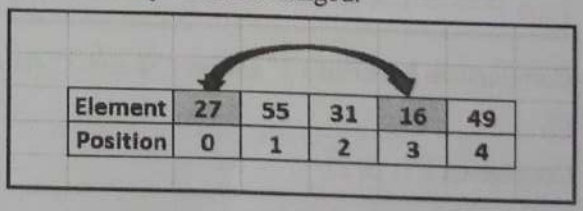
Iteration I



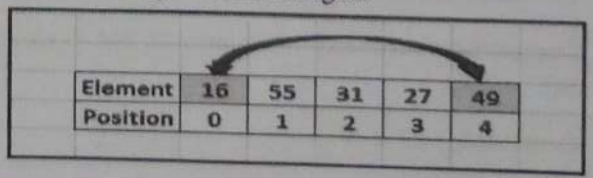
- **Comparison between :** 0^{th} element **31** and first element **55.**
- **Conclusion :** 31 < 55
- **Result :** No interchange



- **Comparison between :** 0^{th} element **31** and 2^{nd} element **27.**
- **Conclusion :** 31 > 27
- **Result :** They are interchanged.



- **Comparison between :** 0^{th} element 27 and 3^{rd} element **16.**
- **Conclusion :** 27 > 16
- **Result :** They are interchanged.



Unit II In Sem.

- **Comparison between :** 0th element **16** and 4th element **49.**
- **Conclusion :** 16 < 49
- **Result :** No interchange

Element	16	55	31	27	49
Position	0	1	2	3	4

- After the completion of first iteration, minimum value in the list that is 16 comes in zeroth position of the sorted list.
- For the 1st position, we start scanning the rest of the list (unsorted) in the same manner in iteration II.

☞ **Iteration II**

Element	16	55	31	27	49
Position	0	1	2	3	4

- **Comparison between :** 1st element **55** and 2nd element **31.**
- **Conclusion :** 55 > 31
- **Result :** They are interchanged.

Element	16	31	55	27	49
Position	0	1	2	3	4

- **Comparison between :** 1st element **31** and 3rd element **27.**
- **Conclusion :** 31 > 27
- **Result :** They are interchanged.

Element	16	27	55	31	49
Position	0	1	2	3	4

- **Comparison between :** 1st element **27** and 4th element **49.**
- **Conclusion :** 27 < 49
- **Result :** No interchange.

Element	16	27	55	31	49
Position	0	1	2	3	4

- After the completion of second iteration, second minimum value in the list that is 27 comes in the first position of the sorted list.
- For the 2nd position, where 55 is present, we start scanning the rest of the list (unsorted) in a same manner in Iteration III.

☞ **Iteration III**

Element	16	27	55	31	49
Position	0	1	2	3	4

- **Comparison between :** 2nd element **55** and 3rd element **31.**
- **Conclusion** : 55 > 31
- **Result :** They are interchanged.

Element	16	27	31	55	49
Position	0	1	2	3	4

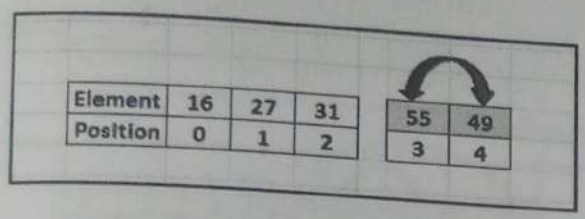
- **Comparison between :** 2nd element **31** and 4th element 49.
- **Conclusion :** 31 < 49
- **Result :** No interchange.

Element	16	27	31	55	49
Position	0	1	2	3	4

- After the completion of third iteration, third minimum value in the list that is 31 comes in the second position of the sorted list.

- For the 3rd position, where 55 is present, we start scanning the rest of the list (unsorted) in a same manner in Iteration IV.

Iteration IV



- **Comparison between :** 3rd element **55** and 4th element **49.**
- Conclusion : 55 > 49
- **Result :** They are interchanged.
- Now the list is sorted completely.

Element	16	27	31	49	55
Position	0	1	2	3	4

- There are 5 elements in above example, hence (n-1) i.e. (5-1) = 4 iterations are required to sort complete list.

2.15.1(A) Algorithm of Selection Sort

1. In first iteration first element is compared with rest of elements. If first element is greater than that then they are swapped.
2. After completion of first iteration smallest element is stored at 0th location.
3. In second iteration second element is compared with rest of (3rd to nth location) elements and process of swapping is repeated.
4. If the list contains n elements, then (n-1) iterations are required.

2.15.1(B) Analysis of Selection Sort

GQ. 2.15.1 *Write complexity of selection sort.* **(2 Marks)**

- In first iteration (n-1) comparisons are required. In second iteration (n-2) comparisons are required and so on.
- From this we conclude that,

$$= (n-1) + (n-2) + ...3+2+1$$

$$= n(n-1)/2$$

$$= O(n^2)$$

Unit II In Sem.

2.15.1(C) Program on Selection Sort in Ascending Order

GQ. 2.15.2 *Write a program in 'C++' language for selection sort.* **(7 Marks)**

```
#include <iostream.h>
int main()
{
  int arr[100], i, n, j, temp;
  cout<<"Enter the number of elements you want be sort: ";
  cin>>n
```

Accepts value of n from user.

```
cout<<" Enter "<<n<<" elements : ";
  for(i=0; i<n ; i++)
  {
      cin>>arr[i];
  }
```

Loop will be executed n times.

```
  for(i=0;i<n;++i)
  {
  for(j=i+1;j<n;++j)
```

```
    {
      if(arr[i]>arr[j])
      {
        temp=arr[i];
        arr[i]=arr[j];
        arr[j]=temp;
      }
    }
}
  cout<<Sorted elements are : ";
  for(j=0;j<n;++j)
  {
    cin<<arr[j]<<" ";
    }
  return 0;
}
```

In unsorted array, if the first position element is greater than next element then they get swapped

Prints the sorted elements in ascending order.

Output

```
C:\cpp\sort.exe
Enter the number of elements you want be sort: 5
Enter 5 elements : 31 55 27 16 49
Sorted elements are : 16  27  31  49  55
```

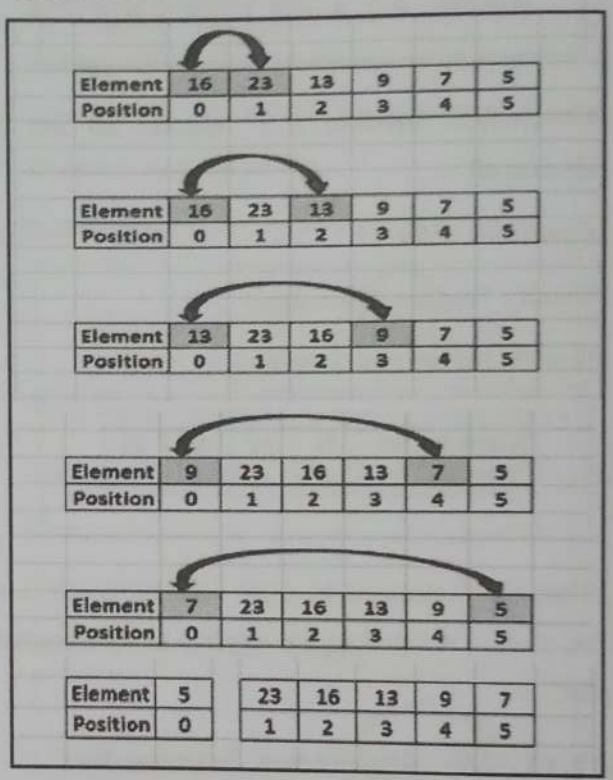
Explanation

- Here program accepts a number from user.
- In unsorted array, if the first position element is greater than next element then they get swapped.
- Above process is repeated until all elements in the list do not get sorted.
- Finally prints the sorted elements.

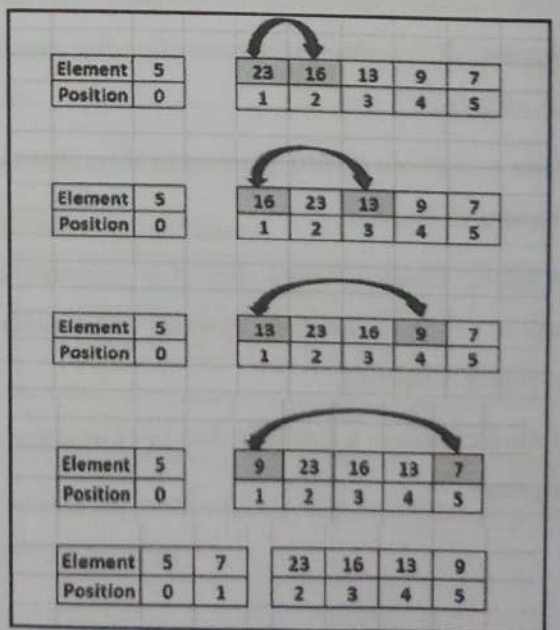
2.15.1(D) Examples on Selection Sort for Given Numbers

GQ. 2.15.3 Sort the following numbers using selection sort. Numbers : 16, 23, 13, 9, 7, 5. **(3 Marks)**

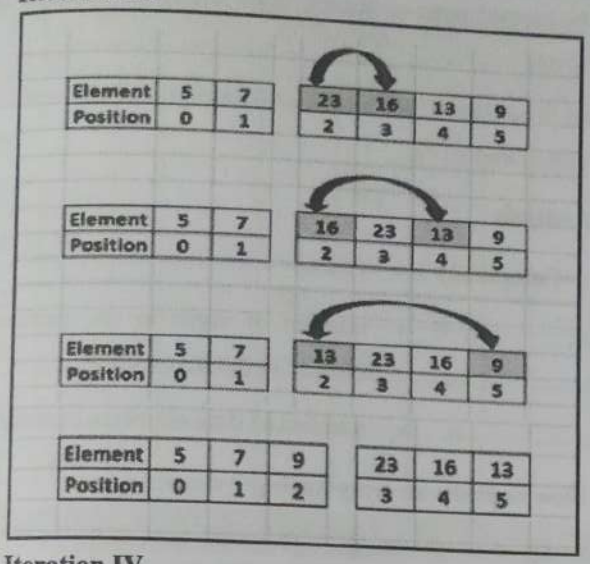
Iteration I



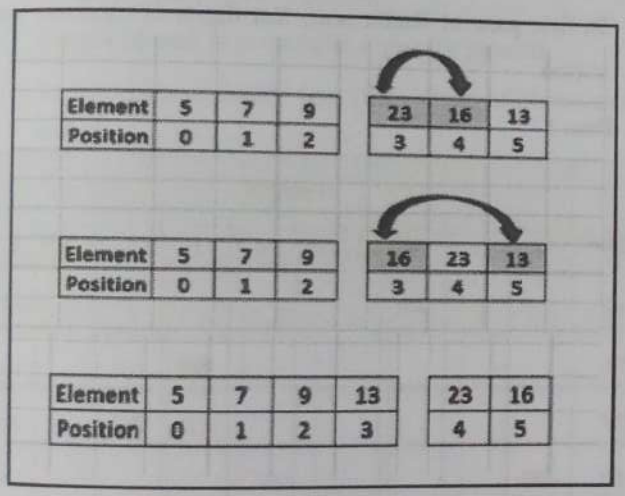
Iteration II



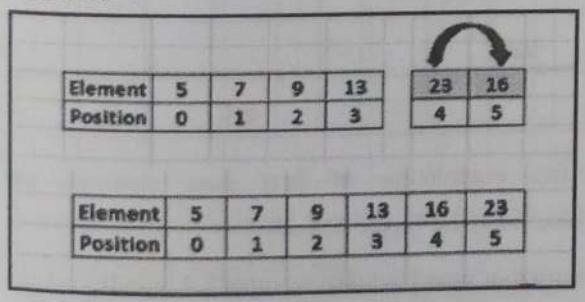
☞ **Iteration III**



☞ **Iteration IV**



☞ **Iteration V**



⏭ 2.15.2 BUCKET SORT

- Bucket sort is a sorting algorithm that works by distributing the elements of an array into a number of buckets.
- Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm.
- Bucket sort can be implemented with comparisons and therefore can also be considered a comparison sort algorithm.
- The computational complexity depends on the algorithm used to sort each bucket, the number of buckets to use, and whether the input is uniformly distributed.

Unit II In Sem.

2.15.2(A) Algorithm of Bucket Sort

1. Set up an array of initially empty "buckets".
2. **Scatter :** Go over the original array, putting each object in its corresponding bucket.

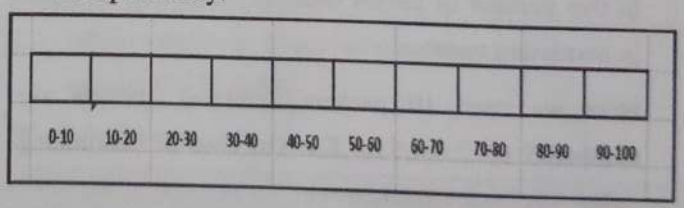
   Sort each non-empty bucket.
3. **Gather :** Visit the buckets in order and put all elements back into the original array.

2.15.2(B) Examples on Bucket Sort

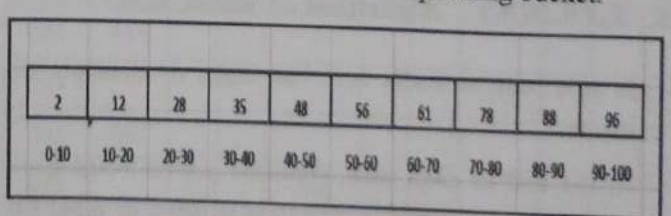
GQ. 2.15.4 *Sort the following elements using bucket sort*

88, 12, 48, 96, 35, 78, 2, 56, 28, 61. **(4 Marks)**

1. Set up an array.



2. Putting each element in its corresponding bucket.



3. Visit the buckets in order and put all elements back into the original array

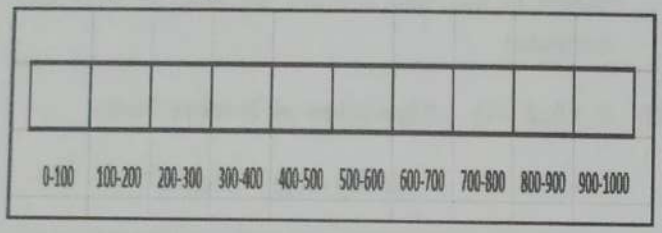
   2, 12, 28, 35, 48, 56, 61, 78, 88, 96.

GQ. 2.15.5 *Sort the following elements using bucket sort*

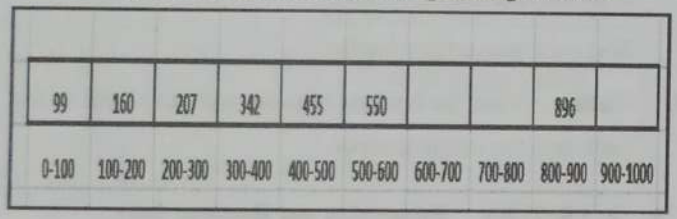
342, 896, 207, 160, 99, 550, 455

(4 Marks)

1. Set up an array



2. Putting each element in its corresponding bucket.



3. Visit the buckets in order and put all elements back into the original array 99, 160, 207, 342, 455, 550, 896.

2.15.3 Radix Sort

- Radix sort is advanced method of sorting.
- In this number of passes depends on number of digits in maximum number.
- Here we need 10 packets (buckets), packets are numbered 0,1,2,3,4,5,6,7,8,9. (In case of descending order 9,8,7,6,5,4,3,2,1,0)

2.15.3(A) Algorithm of Radix Sort

1. In first pass of Radix sort, last digits of elements are sorted.
2. At the end of each pass, elements are collected from bucket 0 to 9, output of this Pass is taken as input for next Pass.
3. In second pass of Radix, ten's digits of elements are sorted.
4. In third pass of Radix sort, 100's digits of elements are sorted and so on.

☞ **Analysis**

Complexity = O(d.n)

where, d = number of digits in the maximum number in the given array.

n = number of data elements in array.

Now consider an array having 7 elements :

int A[] = {342, 896, 207, 160, 101, 550, 455}

- In first pass of Radix sort, last digits of elements are sorted.

☞ **Pass I**

Input	Packets									
	0	1	2	3	4	5	6	7	8	9
342			342							
896							896			
207								207		
160	160									
101		101								
550	550									
455						455				

- After completion of first pass elements are : 160,550,101,342,455,896,207.
- Output of pass I is taken as input for pass II.
- In second pass of Radix sort ten's digits of elements are sorted.

☞ **Pass II**

Input	Packets									
	0	1	2	3	4	5	6	7	8	9
160							160			
550						550				
101	101									
342					342					
455						455				
896										896
207	207									

After completion of second pass, elements are :

101,207,342,550,455,160,896.

- Output of pass II is taken as input for pass III.
- In third pass of Radix sort 100's digits of elements are sorted.

☞ **Pass III**

Input	Packets									
	0	1	2	3	4	5	6	7	8	9
101		101								
207			207							
342				342						
550						550				
455					455					
160		160								
896									896	

Now the array is sorted,

101,160,207,342,455,550,896.

- Repeat steps until number of digits ends.
- Number of passes required by radix sort = Number of digits in maximum number.
- In above example maximum number is 896 and number of digit in it is 3.Since in above example 3 passes are required.

GQ. 2.15.6 *Sort the given number in ascending order using radix sort. Numbers : 348, 14, 614, 5381, 47.* **(3 Marks)**

- Here, digits in maximum number are 4, so we require 4 passes.
- In first pass of Radix sort last digits of elements are sorted

☞ **Pass I**

Input	Packets									
	0	1	2	3	4	5	6	7	8	9
348									348	
14					14					
614					614					
5381		5381								
47								47		

After completion of first pass elements are: 5381, 14, 614, 47 and 348.

☞ **Pass II**

Input	Packets									
	0	1	2	3	4	5	6	7	8	9
5381									5381	
14		14								
614		614								
47					47					
348					348					

After completion of second pass elements are :

14,614,47,348,5381.

☞ **Pass III**

Input	Packets									
	0	1	2	3	4	5	6	7	8	9
014	014									
614							614			
047	047									
348				348						
5381				5381						

Unit II In Sem.

After completion of third pass elements are :

14, 47, 348, 5381, 614.

☞ **Pass IV**

Input	Packets									
	0	1	2	3	4	5	6	7	8	9
0014	0014									
0047	0047									
0348	0348									
5381						5381				
0614	0614									

After completion of fourth pass elements are :

14, 47, 348, 614, 5381.

2.15.3(B) Example on Radix Sort

GQ. 2.15.7 *Sort the following numbers in ascending order using radix sort. 12, 8, 25, 4, 66, 2, 98, 225.* **(3 Marks)**

- Here, digits in maximum number are 3, so we require 3 passes.
- In first pass of Radix sort last digit of a elements are sorted

☞ **Pass I**

Input	Packets									
	0	1	2	3	4	5	6	7	8	9
12			12							
8									8	
25						25				
4					4					
66							66			
2			2							
98									98	
225						225				

After completion of first pass elements are: 12, 2, 4, 25, 225, 66, 8, 98.

☞ **Pass II**

Input	Packets									
	0	1	2	3	4	5	6	7	8	9
12		12								
02	02									
04	04									
25			25							
225			225							
66							66			
08	08									
98										98

After completion of second pass elements are: 2, 4, 8, 12, 25, 225, 66, 98.

☞ **Pass III**

Input	Packets									
	0	1	2	3	4	5	6	7	8	9
002	2									
004	4									
008	8									
012	12									
025	25									
225			225							
066	66									
098	98									

After completion of third pass elements are :

2, 4, 8, 12, 25, 66, 98, 225.

..Chapter Ends

□□□