COMPUTER SCIENCE

B. Sc. I (CBCS)
Semester-II
2023-2024

1CS2 : Data Structure & OOPS

Unit-III : Searching & Sorting

PROF. V. V. AGARKAR
Assistant Professor & Head
Department of Computer Science

Shri. D. M. Burungale Science & Arts College, Shegaon, Dist. Buldana

UNIT – III

> *Syllabus:* **Sorting and Searching:** Definition and concept. **Searching Techniques:** Linear Search, Binary Search and Indexed Sequential Search. **Sorting Techniques:** Bubble Sort, Selection Sort, Insertion Sort, Radix Sort, Merge Sort and Quick Sort.

Searching

Searching is the process of looking one piece of data that has been stored within a whole group of data. Basically a search algorithm is an algorithm which accepts an argument '***a***' and tries to find the corresponding data where the match of '***a***' occurs in a file or in a table. Following are different types of searching techniques:

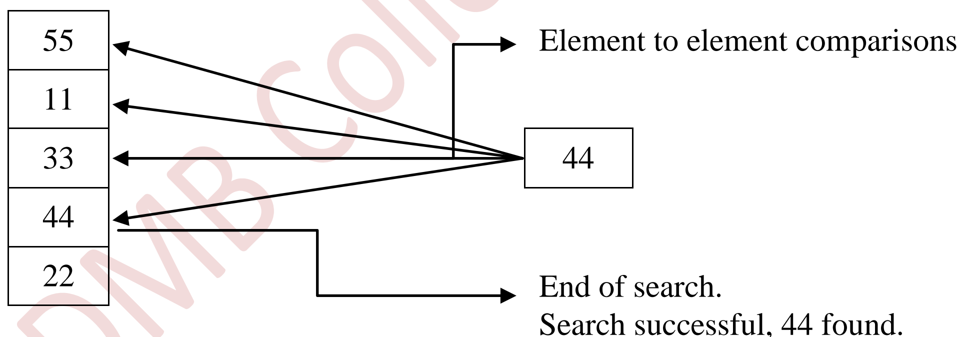
1. Linear or Sequential Search

The simplest of all the searching techniques is Linear or Sequential Search. As the name suggests, the expected element is searched sequentially in the list, one by one, from the first element to last element until a match occurs.

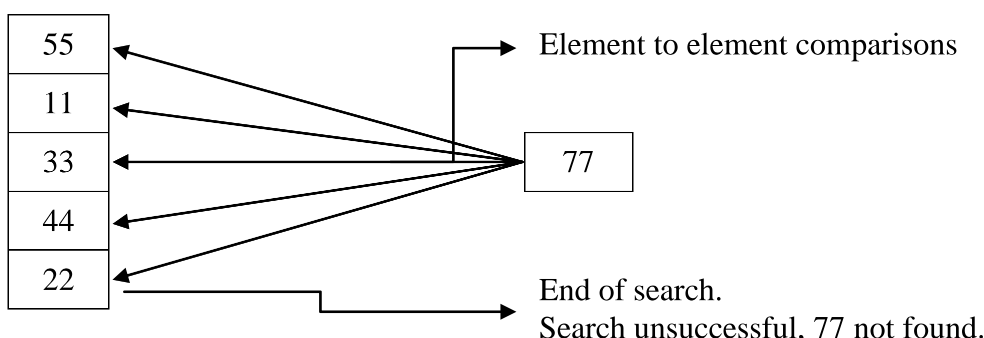
Linear search is not the most efficient way to search, however, it is very simple to implement. Linear search can be applied on sorted or unsorted linear data structure.

Example :-

1] Search `44` in the list : `55, 11, 33, 44, 22`



2] Search `77` in the list : `55, 11, 33, 44, 22`



DMB College, Shegaon

Algorithm for linear search is as follows:

Algorithm

```
LINEAR-SEARCH(A, N, ITEM)
 Here A is an array with N elements. This algorithm
 searches the element ITEM in array A.
 1.  [Initialize]
     Set FLAG := 1.
 2.  Repeat step 3 For K = 1 to N
 3.        IF A[K] = ITEM Then
               Set FLAG := 0
               PRINT "Search is successful" and
               Element is found at location K.
               Exit.
           End If
 4. If FLAG = 1 Then
        PRINT "Search is unsuccessful".
    End If
 5. Exit.
```

DMB College, Shegaon

2. Binary Search

This search is applicable only to ***ordered* or *sorted lists*** (ascending or descending data).

In binary search, first selects the middle element in the list and compares its value to that of the key value. Because, the list is sorted, if the key value is less than the middle value then the key must be in the first half of the list. Likewise, if the value of the key item is greater than that of the middle value in the list, then key lies in the second half of the list.

Now, knowing that key must be in one half of the array or the other, the binary search examines the mid value of the half in which the key must reside. The algorithm thus narrows the search area by half at each step until it has either found the key data or the search fails.

Example :-

Consider the array of 7 elements and search **23** in the array.

A[0]	*A[1]*	*A[2]*	*A[3]*	*A[4]*	*A[5]*	*A[6]*
12	14	16	19	23	27	31

Prof. V. V. AGARKAR
D.M.Burungale Science & Arts College, Shegaon

Low ... ***High***

A[0]	*A[1]*	*A[2]*	*A[3]*	*A[4]*	*A[5]*	*A[6]*
12	**14**	**16**	**19**	**23**	**27**	**31**

1. The element 12 and 31 are at positions low and high, respectively.
2. Calculate mid value, which is as

   Mid = (low+high)/2
   = (0+6)/2
   = 3

   Therefore, A[3] =19 is a middle element.

Low ... mid (A[3]) ... ***High***

A[0]	*A[1]*	*A[2]*	*A[3]*	*A[4]*	*A[5]*	*A[6]*
12	**14**	**16**	**19**	**23**	**27**	**31**

3. The key value 23 is compared with mid element i.e. 19. The key value 23 is greater than 19, so key is on the right half. Hence, *low = mid+1 = 3+1 = 4*.

Low (A[4]) ... ***High*** (A[6])

A[0]	*A[1]*	*A[2]*	*A[3]*	*A[4]*	*A[5]*	*A[6]*
12	**14**	**16**	**19**	**23**	**27**	**31**

4. Calculate mid value of the second half, which is as

   Mid = (low+high)/2
   = (4+6)/2
   = 5

   Therefore, A[5]=27 is a middle element.

Low (A[4]) ... mid (A[5]) ... ***High*** (A[6])

A[0]	*A[1]*	*A[2]*	*A[3]*	*A[4]*	*A[5]*	*A[6]*
12	**14**	**16**	**19**	**23**	**27**	**31**

5. The key value 23 is compared with mid element i.e. 27. The key value 23 is less than 27, so key is on the left half. Hence, *high = mid-1 = 5-1 = 4*.

Low ***High*** (A[4])

A[0]	*A[1]*	*A[2]*	*A[3]*	*A[4]*	*A[5]*	*A[6]*
12	**14**	**16**	**19**	**23**	**27**	**31**

DMB College, Shegaon

6. Calculate mid value, which is as

   Mid = (low+high)/2
   = (4+4)/2
   = 4

   Therefore, A[4]=23 is a middle element.

7. The key value 23 is compared with mid element i.e. 23. The key value 23 is equal to mid value 23, so key is found at position 4. ***Search successful***.

Algorithm

```
BINARY-SEARCH(A, N, ITEM)
 Here A is an array with N elements. This algorithm
 searches the element ITEM in array A. Here low is a
 low bound and high is a high bound of an array.
  1. [Initialize]
     Set FLAG := 1
     Set low := 0 and
     Set high := n-1.
  2. Repeat while (low <= high)
          mid = (low + high)/2
          If ITEM = A[mid] Then
               PRINT "record is found at position", mid+1
               Set FLAG := 0
               EXIT.
          Else
               If ITEM < A[mid] then
                    high = mid-1.
               Else
                    low = mid+1.
               End if
          End if
3. If FLAG = 1 Then
        PRINT "Search is unsuccessful".
    End If
4. EXIT.
```

DMB College, Shegaon

3. Indexed Sequential Search

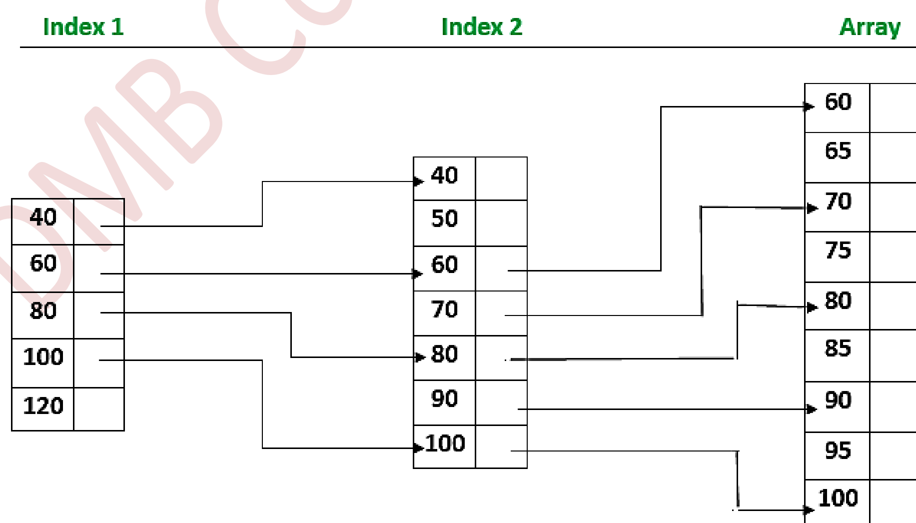
Indexed Sequential Search is a searching algorithm used to find a particular element in a list or array. It's a linear search method, which means it sequentially checks each element of the list until a match is found or the end of the list is reached.

However, unlike a simple sequential search where you start from the beginning and check each element one by one, indexed sequential search involves creating an index structure that contains references to elements at certain intervals. This index structure helps in reducing the number of comparisons required to find an element.

Working of Indexed Sequential Search:

1. Create an index for the list or array. This index typically consists of references to elements at certain intervals. For example, if you have an array of 100 elements, you might create an index containing references to elements at every 10th position (index 0, index 10, index 20, and so on).
2. Search the index to determine the rough location of the element you're searching for. For instance, if you're looking for element X and the index tells you it should be between index 20 and index 30, you know you need to search only that portion of the list.
3. Perform a sequential search within the determined range. Start from the lower bound of the range and check each element sequentially until you find the desired element or reach the upper bound of the range.

Indexed sequential search can be more efficient than a simple sequential search, especially for large lists, because it reduces the number of comparisons required to find an element. However, constructing and maintaining the index may add overhead, particularly if the list is frequently modified. Additionally, the effectiveness of indexed sequential search depends on the distribution of data within the list and the chosen interval for the index.



[Fig. : Indexed Sequential Search]

DMB College, Shegaon

Sorting

Sorting refers to the operation of arranging data in some given order, such as increasing or decreasing, with numerical data, or alphabetically, with character data.

Let *A* be a list of *n* elements $A_1, A_2, A_3, \ldots, A_n$ in memory. *Sorting A* refers to the rearranging the contents of *A* so that they are increasing in order, that is so that:

$$A_1 \leq A_2 \leq A_3 \leq \ldots \leq An$$

For example, suppose an array DATA contains 8 elements as follows:

DATA: 77, 33, 44, 11, 88, 22, 66, 55

After sorting, DATA must appear in memory as follows:

DATA: 11, 22, 33, 44, 55, 66, 77, 88

Following are some sorting algorithms:

1. Bubble sort
2. Selection sort
3. Insertion sort
4. Radix sort
5. Merge sort
6. Quick sort

DMB College, Shegaon

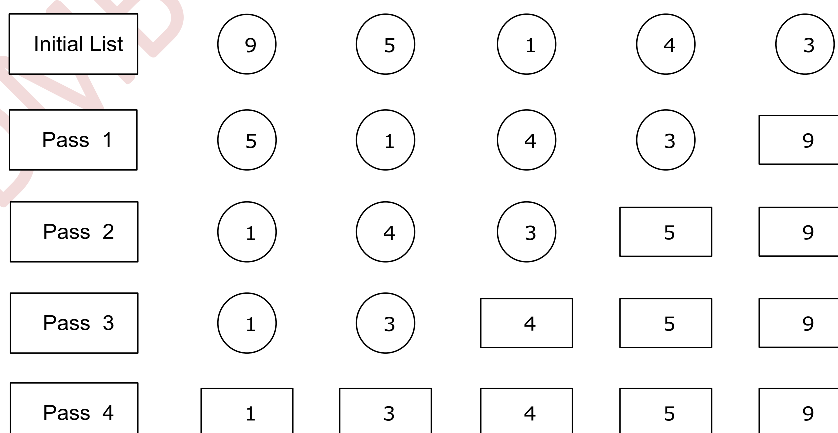
1. Bubble Sort

In bubble sort algorithm, multiple swapping take place in one pass to move or 'bubble' up the greater element to the last of the list.

In this method, two successive elements are compared & swapping is done if the first element is greater than the second element. This process is carried on till the list is sorted.

Example :-

Let us consider the elements 9, 5, 1, 4, and 3 for sorting under bubble sort. Following illustrates the bubble sort:



[Fig : Bubble Sort **]**

1. In pass1, first element 9 is compared with its next element 5. The first element 9 is greater than its next element 5. Hence it is swapped. Now again 9 is compared with next element 1. The next element 1 is smaller than 9 hence swapping is done. The same procedure is repeated with the 4 and 3 and at last the list as 5, 1, 4, 3, 9 after first pass. After the first pass the largest element 9 is bubbled up and it is fixed at the top of the list.
2. In pass 2, the elements of pass 1 are compared. The first element 5 is compared with its next element 1. The elements 5 and 1 are swapped because first element 5 is greater than 1. Next, 5 is compared with element 4. Again, 5 and 4 are swapped. The process is repeated until all successive elements are compared and if necessary swapped. The final list at the end of this pass is 1, 4, 3, 5, 9. After the second pass the largest elements 9 and 5 are fixed at the top of the list.
3. In pass 3, the first element 1 is compared with the next element 4. The element 1 is smaller than 4 and hence no swapping is done. Next, 4 is compared with 3 and swapping is done. The final list at the end of this pass is 1, 3, 4, 5, 9. After the third pass the largest elements 9, 5, and 4 are fixed at the top of the list.
4. In pass 4, the first element 1 is compared with the next element 4. The element 1 is smaller than 4 and hence no swapping is done. At last, the sorted list obtained is 1, 3, 4, 5, 9.

Following is an algorithm for bubble sorting:

Algorithm

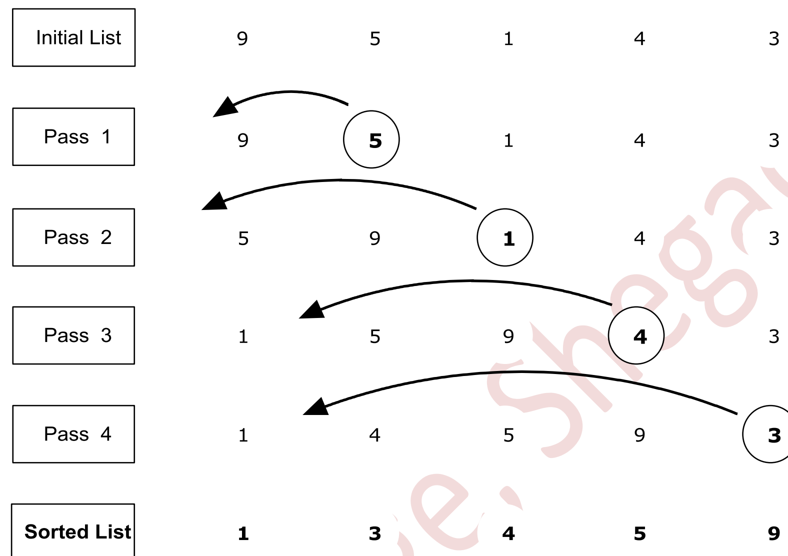
```
BUBBLE-SORT(DATA, N)
Here DATA is an array with N elements. This algorithm
sorts the elements in DATA.
1. Repeat Steps 2 and 3 for I = 1 to N-1.
2.      Set J := 1.  [Initializes pass pointer]
3.      Repeat while J <= N - I
             (a) If DATA[J] > DATA[J+1] then
                     Interchange DATA[J] and DATA[J+1].
                 [End of If structure]
             (b) Set J := J+1.
        [End of step 3 inner loop]
   [End of step 1 outer loop]
4. Exit.
```

DMB College, Shegaon

2. Insertion Sort

Insertion sort is well suited for sorting small data sets or for the insertion of new elements into a sorted sequence. In insertion sort the element is inserted at appropriate place. In insertion sort greater numbers are shifted towards the end of the array and smaller are to beginning.

Example :- Let us consider the elements 9, 5, 1, 4, and 3 for sorting under insertion sort. Following Fig. illustrates the insertion sort:



[Fig. : Insertion Sort]

- Here initial list is: 9 5 1 4 3
- In the **first pass** second element 5 is compared with first element 9. Element 5 is less than 9 and therefore it is inserted before 9 i.e. 9 is shifted at second place and at first place 5 is inserted.

  After first pass list is: 5 9 1 4 3
- In the **second pass** third element 1 is compared with its predecessors 5 & 9, and elements 5, 9, and 1 are placed appropriately.

  After second pass list is: 1 5 9 4 3
- In the **third pass** forth element 4 is compared with its predecessors 1, 5 & 9, and elements 1, 5, 9, and 4 are placed appropriately.

  After third pass list is: 1 4 5 9 3
- In the **forth pass** last element 3 is compared with its predecessors 1, 4, 5 & 9, and elements 1, 4, 5, 9, and 3 are placed appropriately.

  Sorted list is: 1 3 4 5 9

DMB College, Shegaun

Algorithm

```
INSERTION(A, N)
 Here A is an array with N elements. This algorithm sorts
 the elements in array A.

 1. Repeat Steps 2 to 4 for K = 2 to N.
 2.    Set TEMP := A[K] and
       Set PTR := K - 1.
 3.    Repeat while TEMP < A[PTR]
          (a) Set A[PTR+1] := A[PTR]   [Moves elements forward]
          (b) Set PTR := PTR-1.
       [End of loop]
 4.    Set A[PTR+1] := TEMP    [Returns element in proper place]
    [End of step 1 loop]
 5. Exit.
```

3. Selection Sort

Selection sort first find the smallest element in the list and exchange it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and continue in this way until the entire list is sorted.

Example :- Consider the elements 9, 5, 1, 4, and 3 for sorting under selection sort method. Following Fig. illustrates selection sort:

Initial List	9	5	1	4	3
Pass 1	(9)	5	(1)	4	3
Pass 2	1	(5)	9	4	(3)
Pass 3	1	3	(9)	(4)	5
Pass 4	1	3	4	(9)	(5)
Pass 4	1	3	4	5	9

[Fig. : Selection Sort]

Prof. V. V. AGARKAR
D.M.Burungale Science & Arts College, Shegaon
DMB College, Shegaon

1. In pass 1, select first position element 9 and find smallest element in list. The smallest element is 1 at position 3; hence swap 9 and 1, so 1 is placed at first position.
2. In pass 2, select second position element 5 and find second smallest element in list. The second smallest element is 3 at fifth position; hence swap 5 and 3, so 3 is placed at second position.
3. In pass 3, select third position element 9 and find third smallest element in list. The third smallest element is 4 at fourth position; hence swap 9 and 4, so 4 is placed at third position.
4. In pass 4, select fourth position element 9 and find fourth smallest element in list. The fourth smallest element is 5 at fifth position; hence swap 9 and 5, so 5 is placed at fourth position.
5. In last pass the element 9 is placed at last position because it is height element in list.

Algorithm

```
SELECT(A, N, MIN, LOC)
 Here A is an array with N elements. This algorithm sorts
 the elements in array A.

 1. Repeat Steps 2 to 4 for K = 1 to N-1.

 2.        Set MIN := A[K] and
           Set LOC := K.

 3.        Repeat For J = K+1, K+2,..., N
                IF MIN > A[J] Then
                     Set MIN := A[J] and
                     Set LOC := J.
                End If
           [End of step 3 loop]

 4.        Set TEMP := A[K] and
           Set A[K] := A[LOC]and
           Set A[LOC] := TEMP.
    [End of step 1 loop]

 5. Exit.
```

DMB College, Shegaon

4. Merge Sort

Merge sort is one of the most efficient sorting algorithms. It works on the principle of **Divide and Conquer**. Merge sort repeatedly break down a list into several sublists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list.

Merging of two lists done as follows:

The first element of both lists is compared. If sorting in ascending order, the smaller element among two becomes a new element of the sorted list. This procedure is repeated until both the smaller sublists are empty and the newly combined sublist covers all the elements of both the sublists.

Example :

Suppose A is the following list of 14 numbers:

66 33 40 22 55 88 60 11 80 20 50 44 77 30

Each pass of the merge-sort algorithm will start at the beginning of the array A and merge pairs of sorted subarrays as follows:

Pass 1: Merge each pair of elements to obtain the following list of sorted pairs:

33 66 | 22 40 | 55 88 | 11 60 | 20 80 | 44 50 | 30 77

Pass 2: Merge each pair of pairs to obtain the following list of sorted quadruplet:

22 33 40 66 | 11 55 60 88 | 20 44 50 80 | 30 77

Pass 3: Merge each pair of sorted quadruplet to obtain the following two sorted suharrays:

11 22 33 40 55 60 66 88 | 20 30 44 50 77 80

Pass 4: Merge the two sorted subarrays to obtain the single sorted array

11 20 22 30 33 40 44 50 55 60 66 77 80 88

The original array A is now sorted.

5. Quick Sort

The quicksort algorithm uses a recursive "**divide-and-conquer**" approach. The elements to be sorted are partitioned into two subarrays such that all the elements in the "left" subarray are less than or equal to all the elements in the "right" subarray. The subarrays themselves are then recursively partitioned, until we get down to subarrays containing just a single element (which can't be further partitioned). Once all the recursive invocations reach the base case of a single-element subarray, the entire array is sorted.

Recall that partitioning is accomplished by choosing a pivot value, and repeatedly swapping elements such that the left subarray contains only values that are <= the pivot, and the right subarray contains only values that are >= the pivot.

Quicksort is popular because it is not difficult to implement, works well for a variety of different kinds of input data, and is substantially faster than any other sorting methods.

A quick sort first selects a value, which is called the **pivot value**. The role of the pivot value is to assist with splitting the list. The actual position where the pivot value belongs in the final sorted list, commonly called the **split point**, will be used to divide the list for subsequent calls to the quick sort. Although, there are many different ways to choose the pivot value like: choose first element as pivot or choose last element as pivot or choose median of array as pivot or choose random element as pivot.

The three steps of Quicksort are as follows:

Divide:

Rearrange the elements and split the array into two subarrays and an element in between such that each element in the left subarray is less than or equal the middle element and each element in the right subarray is greater than the middle element.

Conquer:

Recursively sort the two subarrays.

Combine:

Combine all the sublist together to get sorted list.

Example :

Suppose A is the following list of 12 numbers:

44 33 11 55 77 90 40 60 99 22 88 66

First step is to find pivot value, so here first number in the list 44 is chosen as pivot value.

(44) 33 11 55 77 90 40 60 99 22 88 66

Beginning with the last number. 66, scan the list from right to left, comparing each number with 44 and stopping at the first number less than 44. The number is 22. Interchange 44 and 22 to obtain the list.

(22) 33 11 55 77 90 40 60 99 (44) 88 66

Beginning with 22, next scan the list in the opposite direction, from left to right, comparing each number with 44 and stopping at the first number greater than 44. The number is 55. Interchange 44 and 55 to obtain the list.

22 33 11 (44) 77 90 40 60 99 (55) 88 66

Beginning this time with 55, now scan the list in the original direction, from right to left, until meeting the first number less than 44. It is 40. Interchange 44 and 40 to obtain the list.

22 33 11 (40) 77 90 (44) 60 99 55 88 66

Beginning with 40, scan the list from left to right. The first number greater than 44 is 77. Interchange 44 and 77 to obtain the list

22 33 11 40 (44) 90 (77) 60 99 55 88 66

Beginning with 77, scan the list from right to left seeking a number less than 44. We do not meet such it before meeting 44. This means all numbers have been scanned and compared with 44. Furthermore, all numbers less than 44 now form the sublist of numbers to the left of 44, and all numbers greater than 44 now form the sublist of numbers to the right of 44, as shown below:

22 33 11 40 (44) 90 77 60 99 55 88 66

First Sublist: 22 33 11 40 — Second Sublist: 90 77 60 99 55 88 66

Thus 44 is correctly placed in its final position, and the task of suiting the original list A has now been reduced to the task of sorting each of the above sublists.

The above reduction step is repeated with First sublist and Second sublist and further it is repeated with each sublist containing 2 or more elements.

Hence after all sublist are processed by above procedure recursively, the sorted list will obtained as follows:

11 22 33 40 44 55 60 66 77 88 90 99

6. Radix Sort

Radix sort is a non-comparative sorting algorithm that sorts numbers by processing individual digits of the numbers from the least significant digit (rightmost) to the most significant digit (leftmost). It can be applied to integers, where each integer is treated as a string of digits. Radix sort operates by distributing elements into buckets based on the value of the current digit being examined, and then collecting them back in order. It can also be applied to strings, treating each string as a sequence of characters.

- **Radix Sort Algorithm**

The detailed steps are as follows −

Step 1 − Check whether all the input elements have same number of digits. If not, check for numbers that have maximum number of digits in the list and add leading zeroes to the ones that do not.

Step 2 − Take the least significant digit of each element.

Step 3 − Sort these digits and change the order of elements based on the output achieved.

Step 4 − Repeat the Step 2 for the next least significant digits until all the digits in the elements are sorted.

Step 5 − The final list of elements achieved after k^{th} loop is the sorted output.

DMB College, Shega

Example:

The Input list is:

[170, 45, 75, 90, 802, 24, 2, 66]

Step 1: Find the largest element in the array, which is 802. It has three digits, so we will iterate three times, once for each significant place. Also leading zeros are added to make all the numbers 3 digit long.

170, 045, 075, 090, 802, 024, 002, 066

Step 2: Starting from the rightmost (unit) digit, sort the numbers based on the unit digits:

170, 090, 802, 002, 024, 045, 075, 066

Step 3: Sort the elements based on the tens place digits.

802, 002, 024, 045, 066, 170, 075, 090

Step 4: Sort the elements based on the hundreds place digits.

002, 024, 045, 066, 075, 090, 170, 802

Step 5: The array is now sorted in ascending order.

The final sorted array using radix sort is:

[2, 24, 45, 66, 75, 90, 170, 802]

DMB College, Shegaon