The Sorting Methods

Internal Sorting Vs External Sorting

In internal sorting all the data to sort is stored in the core memory [RAM] at all times while sorting is in progress.

Eg. Bubble Sort, Insertion Sort, Selection Sort, Quick Sort

In external sorting data is stored outside the core memory (like on DASD Devices - disk, tape drives) and only loaded into memory in small chunks. Usually applied in cases while dealing with massive volume of data.

Eg. Merge Sort

Stable Sorts Vs Unstable Sorts

In a stable sort two or more objects with equal keys appear in the same order in sorted output as they appear in the input unsorted array.

Eg. Bubble Sort, Insertion Sort

In an unstable sort two or more objects with equal keys doesn't appear in same order before and after sorting.

Eg. Selection Sort, Quick Sort, Shell Sort

In-Place Sorts Vs Out-Of-Place Sorts

In-Place Sort orders the elements in a list without consuming extra memory and no auxillary data structure. However a small amount of extra storage space is allowed for auxiliary variables.

Eg. Insertion Sort, Shell Sort, Bubble Sort, Quick Sort*

Out-Of-Place [or Not-in-Place] Sorts require extra memory to order the elemnets.

Eg. Merge Sort, Bucket Sort,

Strictly speaking

An algorithm can only have a constant amount of extra space, counting everything including function calls and pointers.

Comparison-based Sorts Vs Distributive Sorts

Comparative Sort considers the element values in a list and makes comaprisons on list locations while ordering the list.

Eg. Insertion Sort, Shell Sort, Bubble Sort

Distributive Sorts uses the information about distibution of data and contents at list locations in ordering the lists.

Eg. Heap Sort, Bucket Sort, Counting Sort

The sorting methods involving splitting of lists into smaller manageable sub-list while ordering the sub-lists on a pre-defined criteria and integrating gradually to providing final solution are called Divide-and-Conquer Sorts.

Eg. Heap Sort, Quick Sort, Merge Sort

INSERTION SORT

The (linear) insertion sort is one of the simplest sorting algorithms.

With a portion of the array already sorted, the remaining records are moved into their proper places one by one. This algorithm uses sequential search to find the final location of each element.

Insertion sort can be viewed as the result of the iterative application of inserting an element in an ordered array.

The running time for sorting a file of size n with the insertion sort is $O(n^2)$. Hence the algorithm is justifiable only for sorting very small files.

The insertion sort is stable: records with equal keys remain in the same relative order after the sort as before.

When the list is already ordered (or have only a few records out-of-order), the insertion sort for a file of size n yields $O(n)$ running time.

[The only sort with this behaviour – BEST Case]

```
Function INSERTION_SORT (A)
  1.    For J := 2  to A.length
  2.        KEY := A[J]
  3.        I := J - 1
  4.        While I > 0 and A(i) > KEY
  5.            A(I+1) = A(I)
  6.            I := I - 1
  7.        A(I+1) = KEY
```

BUBBLE SORT [SINK SORT | SIFT SORT] – Donald Knuth

The bubble sort algorithm sorts an array by interchanging adjacent records that are in the wrong order.

The algorithm makes repeated passes through the array probing all adjacent pairs until the file is completely in order.

Every complete pass sets at least one element into its final location (in an upward pass the maximum is settled, in a downward the minimum). In this way, every pass is at least one element shorter than the previous pass.

The bubble sort is a simple stable sorting algorithm, but it is inefficient. Its running time of $O(n^2)$ is unacceptable even for medium-sized files.

For files with very few elements out of place, the double-direction bubble sort (or cocktail shaker sort) can be very efficient. If only k of the n elements are out of order, the running time of the double-direction sort is O(kn).

```
Function BUBBLE_SORT (A)
  1.    For I := 2  to A.length
  2.        For J := 1 to A.Length - I + 1
  3.            If A(J) > A(J+1)
  4.                Call SWAP( A(J), A(J+1) )
```

A bubble sort is often considered the most inefficient sorting method since it must exchange items before the final location is known. These “wasted” exchange operations are very costly.

In particular, if during a pass there are no exchanges, then it is evident that the list must be sorted. A bubble sort can be modified to stop early if it finds that the list has become sorted.

This means that for lists that require just a few passes, a bubble sort may have an advantage in that it will recognize the sorted list and stop.

```
Function BUBBLE_SORT_IMPROVED (A)
  1.    For I := 2  to A.length
  2.        SWAPPED := FALSE
  3.        For J := 1 to A.Length – I + 1
  4.            If A(J) > A(J+1)
  5.                Call SWAP( A(J), A(J+1) )
  6.                SWAPPED := TRUE
  7.        If SWAPPED = FALSE
  8.            Break
```

SELECTION SORT

The Selction Sort involves finding the smallest element of the list A and exchanging it with the element in A[1].

It proceeds next with finding the second smallest element of A, and exchange it with A[2]. This process continues for the first n–1 elements of A resulting in a sorted list.

The Selection sort algorithm is based on the idea of finding the minimum or maximum element in an unsorted array and then putting it in its correct position in a sorted array.

The selection sort improves on the bubble sort by making only one exchange for every pass through the list.

On each pass, the largest remaining item is selected and then placed in its proper location.

The selection sort is a combination of searching and sorting. During each pass, the unsorted element with the smallest (or largest) value is moved to its proper position in the array. The number of times the sort passes through the array is one less than the number of items in the array.

```
Function SELECTION_SORT (A)
  1.    For I := 1 to A.length - 1
  2.        MIN_POS := I
  3.        For J := I+1 to A.Length
  4.            If A(J) < A(MIN_POS)
  5.                MIN_POS := J
  6.        IF MIN_POS <> I
  7.            Call SWAP( A(I), A(MIN_POS) )
```

SHELL SORT

The ShellSort algorithm was introduced by Donald L. Shell in 1959. It uses insertion sort on periodic subsequences of the input to produce a faster sorting algorithm.

The shell sort, sometimes called the "diminishing increment sort" improves on the insertion sort by breaking the original list into a number of smaller sublists, each of which is sorted using an insertion sort.

It is an in-place comparison sort. It can either be seen as a generalization of sorting by exchange (bubble sort) or sorting by insertion (insertion sort). The method starts by sorting elements far apart from each other and progressively reducing the gap between them.

Shell sort, compares and swaps elements which are far from each other, hence is not a stable sort. [this makes it faster]

Running Time Analysis of Shell Sort depends on the Gap Sequence [pre-stored].

Default Gap Sequence [Shell's Sequence] := $\lfloor N/2^K \rfloor$, K > 0 [Last Gap := 1]

```
Function SHELL_SORT (A)
  1.    For GAP := A.length/2  downto 1 Step GAP/2
  2.        For I := GAP+1 to A.Length
  3.            KEY := A(I)
  4.            For J := I downto GAP AND A(J – GAP) > KEY Step J-GAP
  5.                A(J) := A(J – GAP)
  6.            A(J) := KEY
```

Shell Sort

-- performs more operations and has higher cache miss ratio

-- has little code and does not use call stack

[used in qsort() in C STL for embedded systems instead of quick sort]

-- Implementation of Shellsort is present in Linux Kernel.

Array [Pre-Sort] :	77	99	22	55	66	11	44	33
Gap	**I**	**Key**	**J**	**Array Contents**							
				1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
4	**5**	**66**	**5**	**77**	**99**	**22**	**55**	**66**	**11**	**44**	**33**
			5	**77**	**99**	**22**	**55**	**77**	**11**	**44**	**33**
			1	**66**	**99**	**22**	**55**	**77**	**11**	**44**	**33**
4	**6**	**11**	**6**	**66**	**99**	**22**	**55**	**77**	**11**	**44**	**33**
			6	**66**	**99**	**22**	**55**	**77**	**99**	**44**	**33**
			2	**66**	**11**	**22**	**55**	**77**	**99**	**44**	**33**
4	**7**	**44**	**7**	**66**	**11**	**22**	**55**	**77**	**99**	**44**	**33**
			7	**66**	**11**	**22**	**55**	**77**	**99**	**44**	**33**
4	**8**	**33**	**8**	**66**	**11**	**22**	**55**	**77**	**99**	**44**	**33**
			8	**66**	**11**	**22**	**55**	**77**	**99**	**44**	**55**
			4	**66**	**11**	**22**	**33**	**77**	**99**	**44**	**55**

2	3	22	3	66	11	22	33	77	99	44	55
			3	66	11	66	33	77	99	44	55
			1	22	11	66	33	77	99	44	55
2	4	33	4	22	11	66	33	77	99	44	55
			4	22	11	66	33	77	99	44	55
2	5	77	5	22	11	66	33	77	99	44	55
			5	22	11	66	33	77	99	44	55
2	6	99	6	22	11	66	33	77	99	44	55
			6	22	11	66	33	77	99	44	55
2	7	44	7	22	11	66	33	77	99	44	55
			7	22	11	66	33	77	99	77	55
			5	22	11	66	33	66	99	77	55
			3	22	11	44	33	66	99	77	55

2	8	55	8	22	11	44	33	66	99	77	55
			8	22	11	44	33	66	99	77	99
			6	22	11	44	33	66	55	77	99
1	2	11	2	22	11	44	33	66	55	77	99
			2	22	22	44	33	66	55	77	99
			1	11	22	44	33	66	55	77	99
1	3	44	3	11	22	44	33	66	55	77	99
			3	11	22	44	33	66	55	77	99
1	4	33	4	11	22	44	33	66	55	77	99
			4	11	22	44	44	66	55	77	99
			3	11	22	33	44	66	55	77	99
1	5	66	5	11	22	33	44	66	55	77	99
			5	11	22	33	44	66	55	77	99

```
-----------------------------------------------------------
  1|  6|    55|  6|  11|  22|  33|  44|  66|  55|  77|  99|
   |   |      |  6|  11|  22|  33|  44|  66|  66|  77|  99|
   |   |      |  5|  11|  22|  33|  44|  55|  66|  77|  99|
-----------------------------------------------------------
  1|  7|    77|  7|  11|  22|  33|  44|  55|  66|  77|  99|
   |   |      |  7|  11|  22|  33|  44|  55|  66|  77|  99|
-----------------------------------------------------------
  1|  8|    99|  8|  11|  22|  33|  44|  55|  66|  77|  99|
   |   |      |  8|  11|  22|  33|  44|  55|  66|  77|  99|
-----------------------------------------------------------

Array [ Sorted ]   :  11|  22|  33|  44|  55|  66|  77|  99|
```

BUCKET SORT

The Bucket Sort [or Bin Sort] is a distribution sort where input elements are initially distributed to several buckets based on an interpolation of the element's key. Each bucket is sorted if necessary, and the buckets' contents are concatenated.

Bucket sorts can be stable. Bucket sort can be implemented with comparisons and therefore can also be considered a comparison sort algorithm.

It uses fixed-size buckets and work well for data sets with key values possibly known and relatively small.

Bucket sort works as follows:

- Set up an array of initially empty buckets.
- Scatter [Distribute] : Go over the original array, putting each object in its bucket.
- Sort each non-empty bucket.
- Gather [Merge] : Visit the buckets in order and put all elements back into the original array.

```
Function BUCKET_SORT (A)
    1.  POS := 1
    2.  FLAG := 0
    3.  While FLAG <> N
    4.      FLAG := 0
    5.      For K := 0 to N-1
    6.          BUCKET( (A(K)/POS)%10, K ) = A(K)
    7.          If (A(K)/POS)%10 = 0
    8.              FLAG := FLAG + 1
    9.      If FLAG = N
    10.         Return
    11.     For J := 0 to 9 , M := 0
    12.         For I := 0 to N-1
    13.             If BUCKET(J, I) > 0
    14.                 A(M) := BUCKET(J, I)
    15.                 BUCKET(J, I) := 0
    16.                 M := M+1
    17.     POS := POS * 10
```

```
BUCKET SORT (A)
  1.    LEN := length [A]
  2.    For I = 1 to LEN do
  3.        Insert A[i] into list B[A[i]/b] where b is the bucket size
  4.    For i = 0 to n-1 do
  5.    Sort list B with Insertion sort
  6.    Concatenate the lists B[0], B[1], . . B[n-1] together in order.
```

QUICK SORT

Quicksort (partition-exchange sort) is an efficient sorting algorithm, serving as a systematic method for placing the elements of an array in order. It was developed by C. A. R. [Tony] Hoare in 1959 and is a commonly used algorithm for sorting.

It is an in-place, comparison sort and its efficient implementations are not a stable sort.

Quicksort gained widespread adoption, appearing, for example, in Unix as the default library sort subroutine. Hence, it lent its name to the C standard library subroutine qsort and in the reference implementation of Java.

The algorithm uses two indices that start at the ends of the array being partitioned, then move toward each other, until they detect an inversion: a pair of elements, one greater than or equal to the pivot, one lesser or equal, that are in the wrong order relative to each other. The inverted elements are then swapped. When the indices meet, the algorithm stops and returns the final index.

```
Function QUICK_SORT (A, P, R)
  1.    If P < R
  2.        Q := Call Partition(A, P, R)
  3.        Call Quick_Sort(A, P, Q-1)
  4.        Call Quick_Sort(A, Q+1, R)
```

The first activation of `Quick_Sort(A, 0, A.Length-1)`

```
Function PARTITION (A, P, R)
  1.    PIVOT := A(R)
  2.    I := P-1
  3.    For J:= P to R-1
  4.        If A(J) < PIVOT
  5.            I := I+1
  6.            Swap A(I) with A(J)
  7.    Swap A(I+1), A(R) )
  8.    Return I+1
```

```
Array [Pre-Sort]   :  22|  88|  77|  11|  33|  55|  66|  44|

 P | R | I | J | X |Array Contents                       |
   |   |   |   |   |  0 |  1 |  2 |  3 |  4 |  5 |  6 |  7 |
-------------------------------------------------------------
Quick_Sort (A, 0, 7)
  0|  7| -1|  0| 44|  22|  88|  77|  11|  33|  55|  66|  44|
  0|  7|  0|  1| 44|  22|  88|  77|  11|  33|  55|  66|  44|
  0|  7|  0|  2| 44|  22|  88|  77|  11|  33|  55|  66|  44|
  0|  7|  0|  3| 44|  22|  88|  77|  11|  33|  55|  66|  44|
  0|  7|  1|  4| 44|  22|  11|  77|  88|  33|  55|  66|  44|
  0|  7|  2|  5| 44|  22|  11|  33|  88|  77|  55|  66|  44|
  0|  7|  2|  6| 44|  22|  11|  33|  88|  77|  55|  66|  44|
  0|  7|  2|  7| 44|  22|  11|  33|  44|  77|  55|  66|  88|
-------------------------------------------------------------
Quick_Sort (A, 0, 2)
  0|  2| -1|  0| 33|  22|  11|  33|
  0|  2|  0|  1| 33|  22|  11|  33|
  0|  2|  1|  2| 33|  22|  11|  33|
-------------------------------------------------------------
```

```
Quick_Sort (A, 0, 1)
  0|  1| -1|  0| 11|  22|  11|
  0|  1| -1|  1| 11|  11|  22|
-------------------------------------------------------------
Pop Call_Stack     |  11|  22|  33|  44|
-------------------------------------------------------------
Quick_Sort (A, 4, 7)
  4|  7|  3|  4| 88|                    |  77|  55|  66|  88|
  4|  7|  4|  5| 88|                    |  77|  55|  66|  88|
  4|  7|  5|  6| 88|                    |  77|  55|  66|  88|
  4|  7|  6|  7| 88|                    |  77|  55|  66|  88|
-------------------------------------------------------------
Quick_Sort (A, 4, 6)
  4|  6|  3|  4| 66|                    |  77|  55|  66|
  4|  6|  3|  5| 66|                    |  77|  55|  66|
  4|  6|  4|  6| 66|                    |  55|  66|  77|
-------------------------------------------------------------
Pop Call_Stack                         |  55|  66|  77|  88|
-------------------------------------------------------------
Array [ Sorted ]   :  11|  22|  33|  44|  55|  66|  77|  88|
```

MERGE SORT

Merge sort is an efficient, general-purpose, comparison-based sorting algorithm. Merge sort is a divide-and-conquer algorithm that was invented by John von Neumann in 1945.

It is a not in place sort and most implementations produce a stable sort.

In the worst case, merge sort does about 39% fewer comparisons than quicksort does in the average case. In terms of moves, merge sort's worst case complexity is O(nlogn)—the same complexity as quicksort's best case, and merge sort's best case takes about half as many iterations as the worst case.

Default sorting algorithm in Perl.

In Java used in Arrays.sort().

In Linux kernel for its linked lists

In Python [TimSort – a Variant of Mergesort]

Also in Java SE, Android Platform, GNU Octave [as standard sort]

```
Function MERGE_SORT (A, P, R)
  1.    If P < R
  2.        Q := FLOOR (P+R)/2
  3.        Call Merge_Sort(A, P, Q)
  4.        Call Merge_Sort(A, Q+1, R)
  5.        Call Merge(A, P, Q, R)
```

The first activation of Merge_Sort(A, 1, N)

```
Function MERGE (A, P, R)
  1.    N1 := Q-P+1
  2.    N2 := R-Q
  3.    Create Arrays L[1.. N1+1] & R[1..N2+1]
  4.    For I := 1 to N1
  5.        L(I) := A(P+I-1)
  6.    For J := 1 to N2
  7.        R(I) := A(Q+J)
```

```
8.    L(N1+1) := INF
9.    R(N2+1) := INF
10.   I := 1, J := 1
11.   For K := P to R
12.       If L(I) <= R(J)
13.           A(K) := L(I)
14.           I := I+1
15.       Else
16.           A(K) := R(J)
17.           J := J+1
```

Array [Pre] :	22	88	77	11	33	55	66	44

P	MID	R	K	Array Contents							
				0	1	2	3	4	5	6	7
0	3	7		22	88	77	11	33	55	66	44
0	1	3		22	88	77	11	33	55	66	44
0	0	1		22	88	77	11	33	55	66	44
0	0	1	0	22	88	77	11	33	55	66	44
0	0	1	1	22	88	77	11	33	55	66	44
0	0	1	2	22	88	77	11	33	55	66	44
2	2	3		22	88	77	11	33	55	66	44
2	2	3	2	22	88	77	11	33	55	66	44
2	2	3	3	22	88	11	11	33	55	66	44
2	2	3	4	22	88	11	77	33	55	66	44

0	1	3	0	22	88	11	77	33	55	66	44
0	1	3	1	11	88	11	77	33	55	66	44
0	1	3	2	11	22	11	77	33	55	66	44
0	1	3	3	11	22	77	77	33	55	66	44
0	1	3	4	11	22	77	88	33	55	66	44

4	5	7		11	22	77	88	33	55	66	44

4	4	5		11	22	77	88	33	55	66	44
4	4	5	4	11	22	77	88	33	55	66	44
4	4	5	5	11	22	77	88	33	55	66	44
4	4	5	6	11	22	77	88	33	55	66	44

6	6	7		11	22	77	88	33	55	66	44
6	6	7	6	11	22	77	88	33	55	66	44
6	6	7	7	11	22	77	88	33	55	44	44
6	6	7	8	11	22	77	88	33	55	44	66

```
 4|  5|  7|  4|  11|  22|  77|  88|  33|  55|  44|  66|
 4|  5|  7|  5|  11|  22|  77|  88|  33|  55|  44|  66|
 4|  5|  7|  6|  11|  22|  77|  88|  33|  44|  44|  66|
 4|  5|  7|  7|  11|  22|  77|  88|  33|  44|  55|  66|
 4|  5|  7|  8|  11|  22|  77|  88|  33|  44|  55|  66|
-------------------------------------------------------
 0|  3|  7|  0|  11|  22|  77|  88|  33|  44|  55|  66|
 0|  3|  7|  1|  11|  22|  77|  88|  33|  44|  55|  66|
 0|  3|  7|  2|  11|  22|  77|  88|  33|  44|  55|  66|
 0|  3|  7|  3|  11|  22|  33|  88|  33|  44|  55|  66|
 0|  3|  7|  4|  11|  22|  33|  44|  33|  44|  55|  66|
 0|  3|  7|  5|  11|  22|  33|  44|  55|  44|  55|  66|
 0|  3|  7|  6|  11|  22|  33|  44|  55|  66|  55|  66|
 0|  3|  7|  7|  11|  22|  33|  44|  55|  66|  77|  66|
 0|  3|  7|  8|  11|  22|  33|  44|  55|  66|  77|  88|
-------------------------------------------------------
Array [Sorted] :  11|  22|  33|  44|  55|  66|  77|  88|
```

HEAP SORT [JWJ Williams (1964), RW FLOYD (in-place version later 1964)]

Heap sort is a comparison-based sorting algorithm. Heapsort can be thought of as an improved selection sort: like that algorithm, it divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element and moving that to the sorted region.

The improvement consists of the use of a heap data structure rather than a linear-time search to find the maximum.

It is slower than a well-implemented quicksort,but exhibits a more favorable worst-case O(nlogn) runtime. Heapsort is an in-place algorithm, but it is not a stable sort.

A heap is a specialized tree-based data structure that satisfies the heap property: if P is a parent node of C, then the key (the value) of P is either greater than or equal to (in a max heap) or less than or equal to (in a min heap) the key of C. The node at the "top" of the heap (with no parents) is called the root node.

The heap is one maximally efficient implementation of an abstract data type called a priority queue, and in fact priority queues are often referred to as "heaps", regardless of how they may be implemented.

Heaps are also crucial in several efficient graph algorithms such as Dijkstra's algorithm [SSSP] and Prim's algorithm [MST).

A Binary Heap is a Binary Tree with following properties.

- It's a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array.
- A Binary Heap is either Min Heap or Max Heap.

Tree Terminologies

- A rooted binary tree has a root node and every node has at most two children.
- A full binary tree (a proper or plane binary tree) is a tree in which every node has either 0 or 2 children.
- A perfect binary tree is a binary tree in which all interior nodes have two children and all leaves have the same depth or same level.
- In a complete binary tree every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible. It can have between 1 and 2^h nodes at the last level h. A complete binary tree can be efficiently represented using an array.
- A balanced binary tree has the minimum possible maximum height (or depth) for the leaf nodes because, for any given number of leaf nodes, the leaf nodes are placed at the greatest height possible.

Algorithm

The heapsort algorithm involves preparing the list by first turning it into a max heap. The algorithm then repeatedly swaps the first value of the list with the last value, decreasing the range of values considered in the heap operation by one, and sifting the new first value into its position in the heap. This repeats until the range of considered values is one value in length.

The steps are:

1. Call the buildHeap() function on the list.
2. Swap the first element of the list with the final element. Decrease the considered range of the list by one.
3. Call the Heapify() function on the list to sift the new first element to its appropriate index in the heap.
4. Go to step (2) unless the considered range of the list is one element.

```
Function PARENT(I)
    return FLOOR(I/2)

Function LEFT(I)
    return 2*I

Function RIGHT(I)
    return 2*I + 1
```

Heaps also satisfy the heap property: for every node i other than the root,

```
    A( PARENT(I) ) >= A(I)
```

```
Procedure BUILD_HEAP(A)
1  HEAP_SIZE(A) := Length(A)
2  For I := FLOOR(Length(A)/2) downto 1
3        Call HEAPIFY(A, I)
```

```
Procedure HEAPIFY(A, I)
1 L := LEFT(I)
2 R := RIGHT(I)
3 If L <= HEAP_SIZE(A) AND A(L) > A(I)
4     LARGEST := L
5 ELSE LARGEST := I
6 IF R <= HEAP_SIZE(A) AND A(R) > A(LARGEST)
7     LARGEST := R
8 If LARGEST <> I
9     Exchange A[I] with A[LARGEST]
10          HEAPIFY(A, LARGEST)
```

```
Procedure HEAPSORT(A)
1  BUILD-HEAP(A)
2  For I := Length(A) downto 2
3       Exchange A(1) with A(I)
4       HEAP_SIZE(A) := HEAP_SIZE(A)-1
5       HEAPIFY(A, 1)

List: 4, 1, 3, 2, 16, 9, 10, 14, 8, 7
```

```
void quick_sort(int *arr, int p, int r){
    int loc;
    if(p<r){
        loc = partition(arr,p,r);
        quick_sort(arr,p,loc-1);
        quick_sort(arr,loc+1,r);
    }
}
int partition(int *arr, int p, int r){
    int i, j, x, t;
    i = p - 1; x = arr[r];
    for(j = p; j < r; j++)
        if(arr[j] <= x){
            i++; swap(arr[i],arr[j],t);
        }
    arr[r] = arr[i + 1]; arr[i + 1] = x;
    return(i + 1);
}
```

```
void merge(int arr[],int p,int q,int r){
    int n1,n2,i,j,k,*left,*right;
    n1=q-p+1; n2=r-q;
    left=(int*)calloc(n1+1,sizeof(int));
    right=(int*)calloc(n2+1,sizeof(int));
    for(i=0;i<n1;i++)
        left[i]=arr[p+i];
    for(j=0;j<n2;j++)
        right[j]=arr[q+j+1];
    left[i]=999999999; right[j]=999999999; i=0; j=0;
    for(k=p;k<=r;k++)
        if(left[i]<=right[j])
            arr[k]=left[i++];
        else
            arr[k]=right[j++];
    return;
}
```

```
void merge_sort(int arr[], int p, int r){
    int mid;
    if(p<r){
        mid=(p+r)/2;
        merge_sort(arr,p,mid);
        merge_sort(arr,mid+1,r);
        merge(arr,p,mid,r);
    }
}
```

DATA STRUCTURES & PROGRAM DESIGN [CST213]

Course Coordinator:

DILIPKUMAR A. BORIKAR

M.TECH [IT] IIT Kharagpur, M.B.A. [Mktg & Fin] RTMNU, [Ph.D.]

Assistant Professor, CSE

Email: borikarda@rkne.edu

Teaching Experience:

18 Years [14 Years at KITS Ramtek]

Courses Taught:

Database Systems [Distributed], Algorithms & Data Structures, Software Engineering, Digital Image Processing

Information Processing, Data Analytics, Sentiment Analysis

Recognitions & Awards:

Institite Silver Medal, School of IT, IIT Kharagpur [2009]

HoD [IT] at KITS Ramtek [August 2009–July 2014]