Program : **B.Tech**

Subject Name: **Analysis and Design of Algorithm**

Subject Code: **CS-402**

Semester: **4th**

Unit-1

ALGORITHM:

An Algorithm is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time. No matter what the input values may be, an algorithm terminates after executing a finite number of instructions. In addition, every algorithm must satisfy the following criteria:

Input: there are zero or more quantities, which are externally supplied; Output: at least one quantity is produced;

Definiteness: each instruction must be clear and unambiguous;

Finiteness: if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps;

Effectiveness: every instruction must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper. It is not enough that each operation be definite, but it must also be feasible.

In formal computer science, one distinguishes between an algorithm, and a program. A program does not necessarily satisfy the fourth condition. One important example of such a program for a computer is its operating system, which never terminates (except for system crashes) but continues in a wait loop until more jobs are entered.

We represent algorithm using a pseudo language that is a combination of the constructs of a programming language together with informal English statements.

DESIGNING ALGORITHMS:

In Computer Science, developing an algorithm is an art or a skill. Before actual implementation of the program, designing an algorithm is very important step.

Steps are:

1. Understand the problem
2. Decision making on
    a. Capabilities of computational devices
    b. Select exact or approximate methods
    c. Data Structures
    d. Algorithmic strategies
3. Specification of algorithms
4. Algorithmic verification
5. Analysis of algorithm
6. Implementation or coding of algorithm

ANALYZING ALGORITHMS:

The efficiency of an algorithm can be decided by measuring the performance of an algorithm.

Performance of a program:

The performance of a program is the amount of computer memory and time needed to run a program. We use two approaches to determine the performance of a program. One is analytical, and the other experimental. In performance analysis we use analytical methods, while in performance measurement we conduct experiments.

Time Complexity:
The time needed by an algorithm expressed as a function of the size of a problem is called the time complexity of the algorithm. The time complexity of a program is the amount of computer time it needs to run to completion.
The limiting behavior of the complexity as size increases is called the asymptotic time complexity. It is the asymptotic complexity of an algorithm, which ultimately determines the size of problems that can be solved by the algorithm.
Space Complexity:
The space complexity of a program is the amount of memory it needs to run to completion. The space need by a program has the following components:
Instruction space: Instruction space is the space needed to store the compiled version of the program instructions.
Data space: Data space is the space needed to store all constant and variable values. Data space has two components:
Space needed by constants and simple variables in program.
Space needed by dynamically allocated objects such as arrays and class instances.
Environment stack space: The environment stack is used to save information needed to resume execution of partially completed functions.
Instruction Space: The amount of instructions space that is needed depends on factors such as:
The compiler used to complete the program into machine code.
The compiler options in effect at the time of compilation
The target computer.

Algorithm Design Goals
The three basic design goals that one should strive for in a program are:
1. Try to save Time
2. Try to save Space
3. Try to save Face

A program that runs faster is a better program, so saving time is an obvious goal. Likewise, a program that saves space over a competing program is considered desirable. We want to “save face” by preventing the program from locking up or generating reams of garbled data.

Basic techniques of designing efficient algorithm
1. Divide-and-Conquer
2. Greedy method
3. Dynamic Programming
4. Backtracking
5. Branch-and-Bound

In this section we will briefly describe these techniques with appropriate examples.
1. **Divide & conquer technique** is a top-down approach to solve a problem.
   The algorithm which follows divide and conquer technique involves 3 steps:
   - Divide the original problem into a set of sub problems.
   - Conquer (or Solve) every sub-problem individually, recursive.
   - Combine the solutions of these sub problems to get the solution of original problem.

2. **Greedy technique** is used to solve an optimization problem. An Optimization problem is one in which, we are given a set of input values, which are required to be either maximized or

minimized (known as objective function) w. r. t. some constraints or conditions. Greedy algorithm always makes the choice (greedy criteria) that looks best at the moment, to optimize a given objective function. That is, it makes a locally optimal choice in the hope that this choice will lead to an overall globally optimal solution. The greedy algorithm does not always guarantee the optimal solution but it generally produces solutions that are very close in value to the optimal.

3. **Dynamic programming** technique is similar to divide and conquer approach. Both solve a problem by breaking it down into a several sub problems that can be solved recursively. The difference between the two is that in dynamic programming approach, the results obtained from solving smaller sub problems are *reused* (by maintaining a table of results) in the calculation of larger sub problems. Thus, dynamic programming is a *Bottom-up* approach that begins by solving the smaller sub-problems, saving these partial results, and then reusing them to solve larger sub-problems until the solution to the original problem is obtained. *Reusing* the results of sub-problems (by maintaining a table of results) is the major advantage of dynamic programming because it avoids the re-computations (computing results twice or more) of the same problem.
   Thus, Dynamic programming approach takes much less time than naïve or straightforward methods, such as divide-and-conquer approach which solves problems in *top-down* method and having lots of re-computations. The dynamic programming approach always gives a guarantee to get an optimal solution.

4. The term **"backtrack"** was coined by American mathematician D.H. Lehmer in the 1950s. Backtracking can be applied only for problems which admit the concept of a "partial candidate solution" and relatively quick test of whether it can possibly be completed to a valid solution. Backtrack algorithms try each possibility until they find the right one. It is a depth-first-search of the set of possible solutions. During the search, if an alternative doesn't work, the search backtracks to the choice point, the place which presented different alternatives, and tries the next alternative. When the alternatives are exhausted, the search returns to the previous choice point and try the next alternative there. If there are no more choice points, the search fails.

5. **Branch-and-Bound** (B&B) is a rather general optimization technique that applies where the greedy method and dynamic programming fail.
   B&B design strategy is very similar to backtracking in that a state-space- tree is used to solve a problem. Branch and bound is a systematic method for solving optimization problems. However, it is much slower. Indeed, it often leads to exponential time complexities in the worst case. On the other hand, if applied carefully, it can lead to algorithms that run reasonably fast on average. The general idea of B&B is a BFS-like search for the optimal solution, but not all nodes get expanded (i.e., their children generated). Rather, a carefully selected criterion determines which node to expand and when, and another criterion tells the algorithm when an optimal solution has been found. Branch and Bound (B&B) is the most widely used tool for solving large scale NP-hard combinatorial optimization problems.

The following table-1.1 summarizes these techniques with some common problems that follow these techniques with their running time. Each technique has different running time (…time complexity).

Design strategy	Problems that follows
Divide & Conquer	• Binary search • Multiplication of two n-bits numbers • Quick Sort • Heap Sort • Merge Sort
Greedy Method	• Knapsack (fractional) Problem • Minimum cost Spanning tree ○ Kruskal‟s algorithm ○ Prim‟s algorithm • Single source shortest path problem ○ Dijkstra‟s algorithm
Dynamic Programming	• All pair shortest path-Floyed algorithm • Chain matrix multiplication • Longest common subsequence (LCS) • 0/1 Knapsack Problem • Traveling salesmen problem (TSP)
Backtracking	• N-queen‟s problem • Sum-of subset
Branch & Bound	• Assignment problem • Traveling salesmen problem (TSP)

Table 1.1: Various Design Strategies

Classification of Algorithms

If ‘n’ is the number of data items to be processed or degree of polynomial or the size of the file to be sorted or searched or the number of nodes in a graph etc.

Next instructions of most programs are executed once or at most only a few times. If all the instructions of a program have this property, we say that its running time is a constant.

Log n When the running time of a program is logarithmic, the program gets slightly slower as n grows. This running time commonly occurs in programs that solve a big problem by transforming it into a smaller problem, cutting the size by some constant fraction. When n is a million, log n is a doubled. Whenever n doubles, log n increases by a constant, but log n does not double until n increases to n2.

n When the running time of a program is linear, it is generally the case that a small amount of processing is done on each input element. This is the optimal situation for an algorithm that must process n inputs.

nlog n This running time arises for algorithms that solve a problem by breaking it up into smaller sub-problems, solving then independently, and then combining the solutions. When n doubles, the running time more than doubles.

n^2 When the running time of an algorithm is quadratic, it is practical for use only on relatively small problems. Quadratic running times typically arise in algorithms that process all pairs of data items (perhaps in a double nested loop) whenever n doubles, the running time increases four-fold.

n3 Similarly, an algorithm that process triples of data items (perhaps in a triple–nested loop) has a cubic running time and is practical for use only on small problems. Whenever n doubles, the running time increases eight-fold.

2^n Few algorithms with exponential running time are likely to be appropriate for practical use, such algorithms arise naturally as “brute–force” solutions to problems. Whenever n doubles, the running time squares.

Complexity of Algorithms

The complexity of an algorithm M is the function f(n) which gives the running time and/or storage space requirement of the algorithm in terms of the size 'n' of the input data. Mostly, the storage space required by an algorithm is simply a multiple of the data size 'n'. Complexity shall refer to the running time of the algorithm.

The function f(n), gives the running time of an algorithm, depends not only on the size 'n' of the input data but also on the particular data. The complexity function f(n) for certain cases are:

1. Best Case : The minimum possible value of f(n) is called the best case.
2. Average Case : The expected value of f(n).
3. Worst Case : The maximum value of f(n) for any key possible input.

ASYMPTOTIC NOTATIONS (RATE OF GROWTH):

The following notations are commonly use notations in performance analysis and used to characterize the complexity of an algorithm:

1. Big–OH (O)
2. Big–OMEGA (Ω)
3. Big–THETA (Θ)

1. Big–OH O (Upper Bound)

f(n) = O(g(n)), (pronounced order of or big oh), says that the growth rate of f(n) is less than or equal (<) that of g(n) figure 1.1.

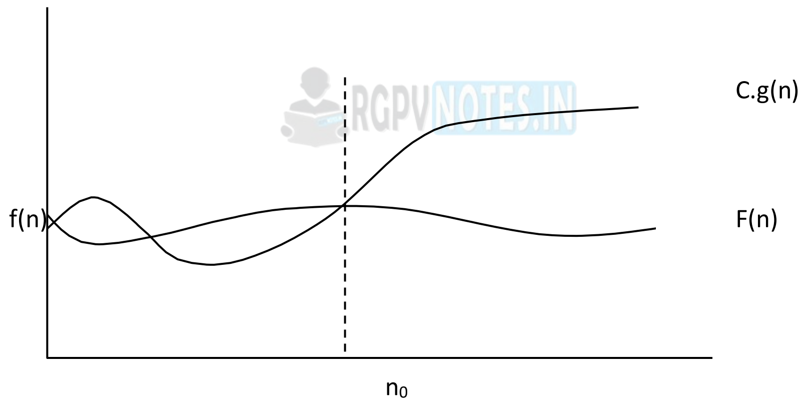

Figure 1.1: Big O Notation

2. Big–OMEGA Ω (Lower Bound)

f(n) = Ω (g(n)) (pronounced omega), says that the growth rate of f(n) is greater than or equal to (>) that of g(n) figure 1.2.

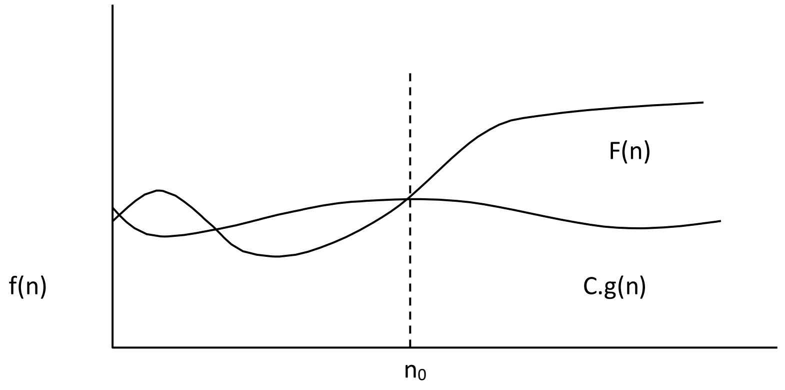

Figure 1.2: Big Ω Notation

3. Big–THETA Ɵ (Same order)

f(n) = ⍰(g(n)) (pronounced theta), says that the growth rate of f(n) equals (=) the growth rate of g(n) [if f(n) = O(g(n)) and T(n) = Ɵ (g(n)] figure 1.3.

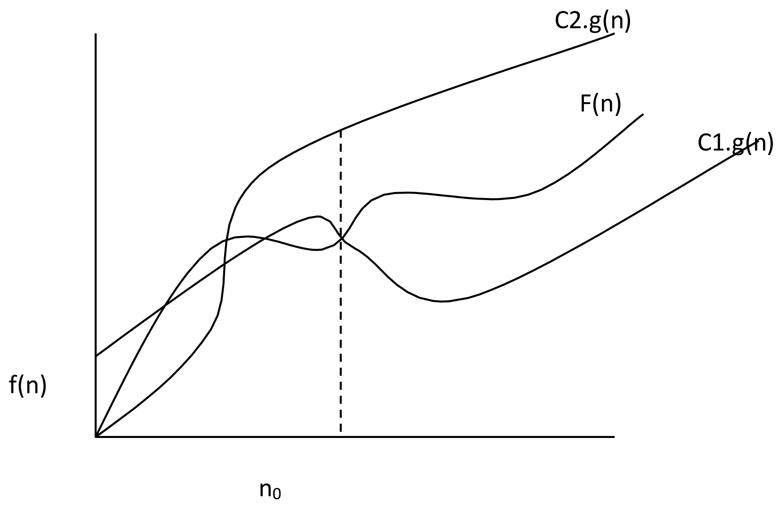

Figure 1.3: Big Ω Notation

Analyzing Algorithms

Suppose 'M' is an algorithm, and suppose 'n' is the size of the input data. Clearly the complexity f(n) of M increases as n increases. It is usually the rate of increase of f(n) we want to examine. This is usually done by comparing f(n) with some standard functions. The most common computing times are:

O(1), O(log n), O(n), O(nlogn), O(n^2), O(n^3), O(2^n), n! and n^n

Numerical Comparison of Different Algorithms

The execution time for six of the typical functions is given below:

n	**logn**	**n*logn**	**n^2**	**n^3**	**2^n**
1	0	0	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4096	65,536
32	5	160	1024	32,768	4,294,967,296
64	6	384	4096	2,62,144	Note 1
128	7	896	16,384	2,097,152	Note 2
256	8	2048	65,536	1,677,216	????????

Table 1.2: Comparison among various complexities

HEAP AND HEAP SORT:

A heap is a data structure that stores a collection of objects (with keys), and has the following properties:

- Complete Binary tree
- Heap Order

It is implemented as an array where each node in the tree corresponds to an element of the array.

Binary Heap:

- A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.
- A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater (or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min heap. The heap can be represented by binary tree or array.

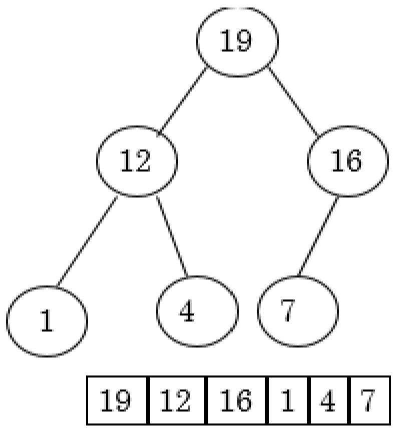

Figure 1.4: Heap

- The root of the tree A[1] and given index *i* of a node, the indices of its parent, left child and right child can be computed

```
PARENT (i)
    return floor(i/2)
LEFT (i)
    return 2i
RIGHT (i)
    return 2i + 1
```

Types of Heaps

Heap can be of 2 types:

Max Heap

- Store data in ascending order
- Has property of

A[Parent(i)] ≥ A[i]

Min Heap

- Store data in descending order
- Has property of

A[Parent(i)] ≤ A[i]

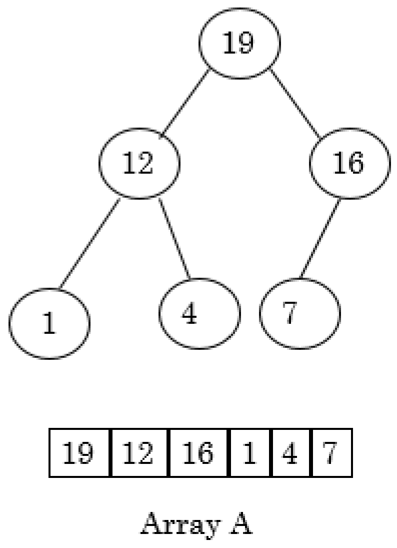

Figure 1.5: Max Heap example

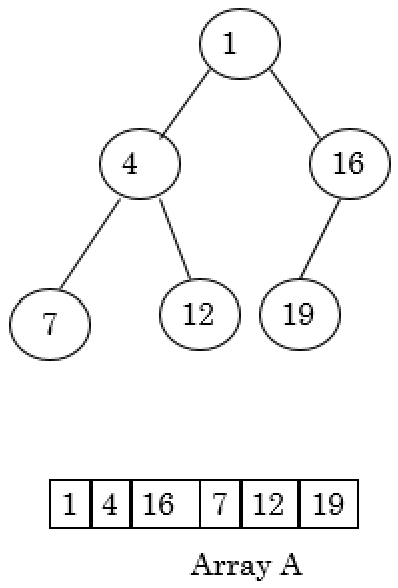

Figure 1.6: Min Heap example

Heap sort is a comparison-based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.

Heap Sort Algorithm for sorting in increasing order:

1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
3. Repeat above steps while size of heap is greater than 1.

Procedures on Heap

- Heapify
- Build Heap
- Heap Sort

1. **Heapify**

   Heapify picks the largest child key and compare it to the parent key. If parent key is larger than heapify quits, otherwise it swaps the parent key with the largest child key. So that the parent is now becomes larger than its children.

```
Heapify(A, i)
{
    l ← left(i)
    r ← right(i)
    if l <= heapsize[A] and A[l] > A[i]
      then largest ←l
      else largest ← i
    if r <= heapsize[A] and A[r] > A[largest]
      then largest ← r
    if largest != i
      then swap A[i] ←→ A[largest]
        Heapify(A, largest)
}
```

2. **Build Heap**

   We can use the procedure 'Heapify' in a bottom-up fashion to convert an array A[1 . . *n*] into a heap. Since the elements in the subarray A[*n*/2 +1 . . *n*] are all leaves, the procedure BUILD_HEAP goes through the remaining nodes of the tree and runs 'Heapify' on each one. The bottom-up order of processing node guarantees that the subtree rooted at children are heap before 'Heapify' is run at their parent.

```
Buildheap(A)
  {
      heapsize[A] ←length[A]
      for i ←|length[A]/2  //down to 1
        do Heapify(A, i)
  }
```

3. **Heap Sort**

   The heap sort algorithm starts by using procedure BUILD-HEAP to build a heap on the input array A[1 . . *n*]. Since the maximum element of the array stored at the root *A*[1], it can be put into its correct final position by exchanging it with *A*[*n*] (the last element in *A*). If we now discard node n from the heap than the remaining elements can be made into heap. Note that the new element at the root may violate the heap property. All that is needed to restore the heap property.

```
Heapsort(A)
{
    Buildheap(A)
    for i ← length[A] //down to 2
      do swap A[1] ←→ A[i]
      heapsize[A] ← heapsize[A] - 1
      Heapify(A, 1)
}
```

Complexity

Time complexity of heapify is O(Logn). Time complexity of create and BuildHeap() is O(n) and overall time complexity of Heap Sort is O(n Logn).

INTRODUCTION TO DIVIDE AND CONQUER TECHNIQUE:

Divide & conquer technique is a top-down approach to solve a problem.

The algorithm which follows divide and conquer technique involves 3 steps:

- Divide the original problem into a set of sub problems.
- Conquer (or Solve) every sub-problem individually, recursive.
- Combine the solutions of these sub problems to get the solution of original problem.

BINARY SEARCH:

The Binary search technique is a search technique which is based on Divide & Conquer strategy. The entered array must be sorted for the searching, then we calculate the location of mid element by using formula mid= (Beg + End)/2, here Beg and End represent the initial and last position of array. In this technique we compare the Key element to mid element. So there May be three cases:-

1. If array[mid] = = Key (Element found and Location is Mid)
2. If array[mid] > Key, then set End = mid-1. (continue the process)
3. If array [mid] < Key, then set Beg=Mid+1. (Continue the process)

Binary Search Algorithm

```
1. [Initialize segment variable] set beg=LB,End=UB and Mid=int(beg+end)/2.
2. Repeat step 3 and 4 while beg<=end and Data[mid] != item.
3. If item< data[mid] then set end=mid-1
      Else if Item>data[mid] then set beg=mid+1[end of if structure]
4. Set mid= int(beg+end)/2.[End of step 2 loop]
5. If data[mid]=item then set Loc=
     Mid. Else set loc=null[end of
     if structure]
6. Exit.
```

Time complexity

As we dispose off one part of the search case during every step of binary search, and perform the search operation on the other half, this results in a worst case time complexity of O($\log_2 n$).

MERGE SORT:

Merge sort is a divide-and-conquer algorithm based on the idea of breaking down a list into several sub-lists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list.

- Divide the unsorted list into NN sublists, each containing 11 element.
- Take adjacent pairs of two singleton lists and merge them to form a list of 2 elements. NN will now convert into N/2N/2 lists of size 2.
- Repeat the process till a single sorted list of obtained.

While comparing two sublists for merging, the first element of both lists is taken into consideration. While sorting in ascending order, the element that is of a lesser value becomes a new element of the sorted list. This procedure is repeated until both the smaller sublists are empty and the new combined sublist comprises all the elements of both the sublists.

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

How Merge Sort Works?

To understand merge sort, we take an unsorted array as the following –

We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.

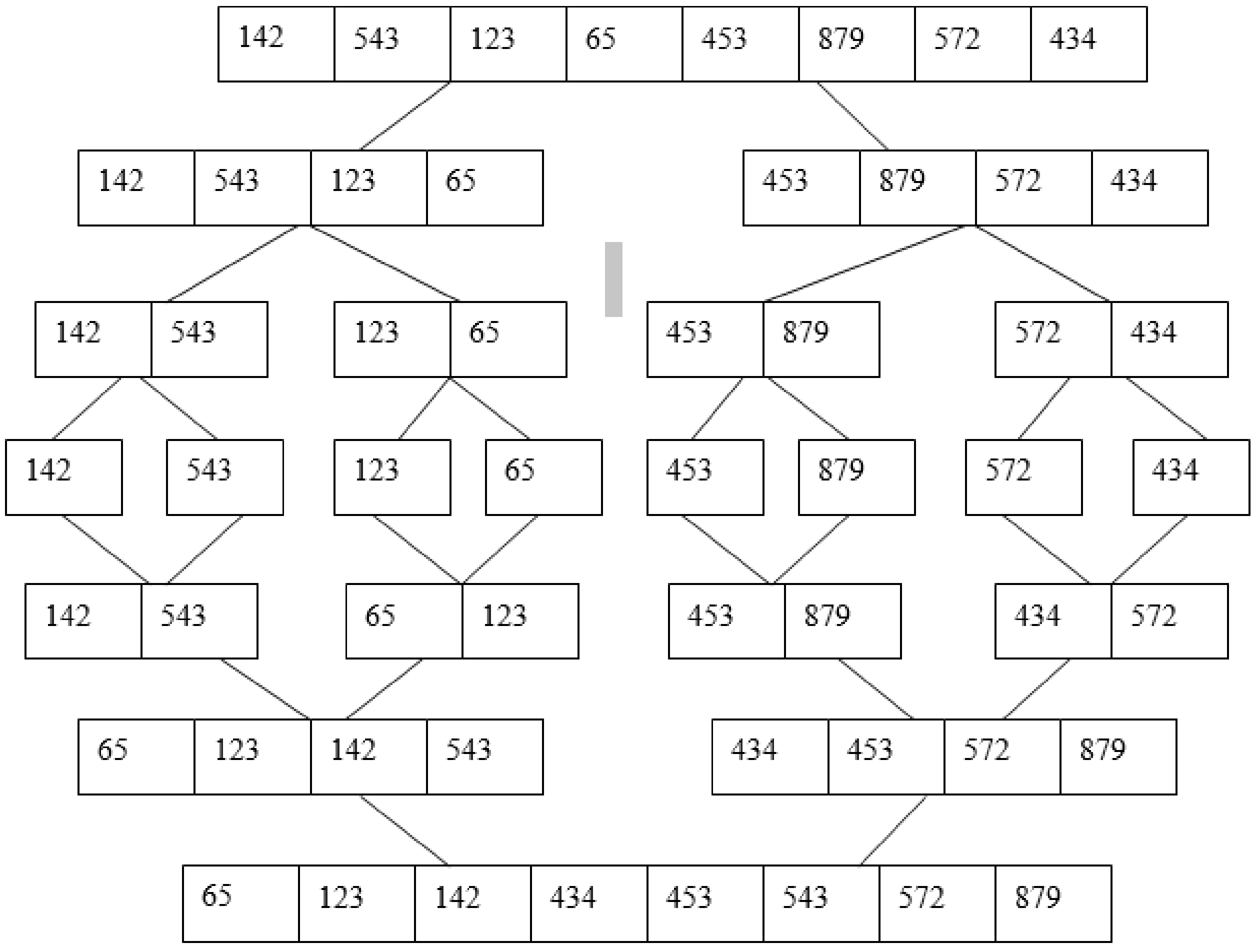

Figure 1.7: Merge Sort

Algorithm:

```
MergeSort(A, p, r)
{
    if( p < r )
    {
        q = (p+r)/2;
        mergeSort(A, p, q);
        mergeSort(A, q+1, r);
        merge(A, p, q, r);
    }
}
```

```
Merge (A, p, q, r )
{
    $n_1 = q - p + 1$
```

```
        $n_2 = r – q$
        declare L[1…$n_1$ + 1] and [R1…$n_2$ + 1] temporary arrays
        for i = 1 to $n_1$
           L[i] = A[p + i - 1]
        for j = 1 to $n_2$
       R[j] = numbers[q+ j]
       L[$n_1$ + 1] = ∞
       R[$n_2$ + 1] = ∞
       i = 1
       j = 1
      for k = p to r
           If (L[i] ≤ R[j])

               A[k] = L[i]

               i = i + 1
          else
              A[k] = R[j]
              j = j + 1
}
```

Time Complexity:
Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.
$T(n) = 2T(n/2) + \Theta(n)$
The above recurrence can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is $\Theta(n\text{Log}n)$.
Time complexity of Merge Sort is $\Theta(n\text{Log}n)$ in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.
Auxiliary Space: O(n)
Algorithmic Paradigm: Divide and Conquer
Sorting In Place: No in a typical implementation
Stable: Yes

QUICK SORT:
Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

- Always pick first element as pivot.
- Always pick last element as pivot (implemented below)
- Pick a random element as pivot.
- Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.
/* low --> Starting index, high --> Ending index */

Quick Sort Algorithm

```
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
/* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
    array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low - 1)  // Index of smaller element

    for (j = low; j <= high- 1; j++)
    {
        // If current element is smaller than or
        // equal to pivot
        if (arr[j] <= pivot)
        {
            i++;    // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

Analysis of QuickSort

Time taken by QuickSort in general can be written as following.

$$T(n) = T(k) + T(n-k-1) + \Theta(n)$$

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements which are smaller than pivot.
The time taken by QuickSort depends upon the input array and partition strategy. Following are three cases.

Worst Case: The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for worst case.

$T(n) = T(0) + T(n-1) + \Theta(n)$

which is equivalent to

$T(n) = T(n-1) + \Theta(n)$

The solution of above recurrence is $\Theta(n^2)$.

Best Case: The best case occurs when the partition process always picks the middle element as pivot. Following is recurrence for best case.

$T(n) = 2T(n/2) + \Theta(n)$

The solution of above recurrence is Θ (nLogn). It can be solved using case 2 of Master Theorem.

Average Case:

To do average case analysis, we need to consider all possible permutation of array and calculate time taken by every permutation which doesn't look easy.

We can get an idea of average case by considering the case when partition puts O(n/9) elements in one set and O(9n/10) elements in other set. Following is recurrence for this case.

$T(n) = T(n/9) + T(9n/10) + \Theta(n)$

Solution of above recurrence is also O(nlogn)

Although the worst-case time complexity of QuickSort is $O(n^2)$ which is more than many other sorting algorithms like Merge Sort and Heap Sort, QuickSort is faster in practice, because its inner loop can be efficiently implemented on most architectures, and in most real-world data. QuickSort can be implemented in different ways by changing the choice of pivot, so that the worst case rarely occurs for a given type of data. However, merge sort is generally considered better when data is huge and stored in external storage.

STRASSEN'S MATRIX MULTIPLICATION:

The Strassen's method of matrix multiplication is a typical divide and conquer algorithm. We've seen so far, some divide and conquer algorithms like merge sort and the Karatsuba's fast multiplication of large numbers. However, let's get again on what's behind the divide and conquer approach.

Unlike the dynamic programming where we "expand" the solutions of sub-problems in order to get the final solution, here we are talking more on joining sub-solutions together. These solutions of some sub-problems of the general problem are equal and their merge is somehow well defined.

General Algorithm without Strassen's

```
MMult(A,B, n)
1. If n = 1 Output A × B
2. Else
3. Compute A11,B11, . . . ,A22,B22 % by computing m = n/2
4. X1   MMult(A11,B11, n/2)
5. X2   MMult(A12,B21, n/2)
6. X3   MMult(A11,B12, n/2)
7. X4   MMult(A12,B22, n/2)
8. X5   MMult(A21,B11, n/2)
```

```
9. X6   MMult(A22,B21, n/2)
10. X7   MMult(A21,B12, n/2)
11. X8   MMult(A22,B22, n/2)
12. C11   X1 + X2
13. C12   X3 + X4
14. C21   X5 + X6
15. C22   X7 + X8
16. Output C
17. End If
```

Complexity of above algorithm is: $T(n) = \Theta (n^{\log_2 (8)}) = \Theta (n^3)$

Strassen Multiplication Algorithm

```
Strassen(A,B)
1. If n = 1 Output A × B
2. Else
3. Compute A11,B11, . . . ,A22,B22 % by computing m = n/2
4. P1   Strassen(A11,B12 − B22)
5. P2   Strassen(A11 + A12,B22)
6. P3   Strassen(A21 + A22,B11)
7. P4   Strassen(A22,B21 − B11)
8. P5   Strassen(A11 + A22,B11 + B22)
9. P6   Strassen(A12 − A22,B21 + B22)
10. P7   Strassen(A11 − A21,B11 + B12)
11. C11   P5 + P4 − P2 + P6
12. C12   P1 + P2
13. C21   P3 + P4
14. C22   P1 + P5 − P3 − P7
15. Output C
16. End If
```

Complexity = $T(n) = \Theta (n^{\log_2 (7)}) = \Theta (n^{2.8})$