1.7 ALGORITHM

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

From the data structure point of view, following are some important categories of algorithms −

- **Search** − Algorithm to search an item in a data structure.
- **Sort** − Algorithm to sort items in a certain order.
- **Insert** − Algorithm to insert item in a data structure.
- **Update** − Algorithm to update an existing item in a data structure.
- **Delete** − Algorithm to delete an existing item from a data structure.

1.7.1 Characteristics of an Algorithm

Not all procedures can be called an algorithm. An algorithm should have the following characteristics −

An algorithm should have the following characteristics −

- **Clear and Unambiguous**: Algorithm should be clear and unambiguous. Each of its steps should be clear in all aspects and must lead to only one meaning.
- **Well-Defined Inputs**: If an algorithm says to take inputs, it should be well-defined inputs.
- **Well-Defined Outputs:** The algorithm must clearly define what output will be yielded and it should be well-defined as well.
- **Finite-ness:** The algorithm must be finite, i.e. it should not end up in an infinite loops or similar.

- **Feasible:** The algorithm must be simple, generic and practical, such that it can be executed upon will the available resources. It must not contain some future technology, or anything.
- **Language Independent:** The Algorithm designed must be language-independent, i.e. it must be just plain instructions that can be implemented in any language, and yet the output will be same, as expected.

1.7.2 Advantages and Disadvantages of Algorithm

Advantages of Algorithms:

- It is easy to understand.
- Algorithm is a step-wise representation of a solution to a given problem.
- In Algorithm the problem is broken down into smaller pieces or steps hence, it is easier for the programmer to convert it into an actual program.

Disadvantages of Algorithms:

- Writing an algorithm takes a long time so it is time-consuming.
- Branching and Looping statements are difficult to show in Algorithms.

1.7.3 Different approach to design an algorithm

1. **Top-Down Approach:** A top-down approach starts with identifying major components of system or program decomposing them into their lower level components & iterating until desired level of module complexity is achieved . In this we start with topmost module & incrementally add modules that is calls.

2. **Bottom-Up Approach:** A bottom-up approach starts with designing most basic or primitive component & proceeds to higher level components. Starting from very bottom , operations that provide layer of abstraction are implemented

1.7.4 How to Write an Algorithm?

There are no well-defined standards for writing algorithms. Rather, it is problem and resource dependent. Algorithms are never written to support a particular programming code.

As we know that all programming languages share basic code constructs like loops (do, for, while), flow-control (if-else), etc. These common constructs can be used to write an algorithm.

1.7.4 How to Write an Algorithm?

There are no well-defined standards for writing algorithms. Rather, it is problem and resource dependent. Algorithms are never written to support a particular programming code.

As we know that all programming languages share basic code constructs like loops (do, for, while), flow-control (if-else), etc. These common constructs can be used to write an algorithm.

We write algorithms in a step-by-step manner, but it is not always the case. Algorithm writing is a process and is executed after the problem domain is well-defined. That is, we should know the problem domain, for which we are designing a solution.

Example

Let's try to learn algorithm-writing by using an example.

Problem − Design an algorithm to add two numbers and display the result.

Step 1 − START
Step 2 − declare three integers **a**, **b** & **c**
Step 3 − define values of **a** & **b**
Step 4 − add values of **a** & **b**
Step 5 − store output of step 4 to **c**
Step 6 − print **c**
Step 7 − STOP

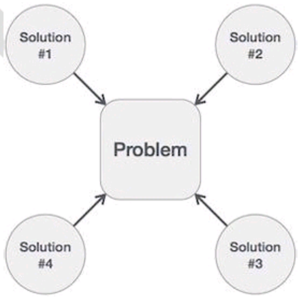
Algorithms tell the programmers how to code the program. Alternatively, the algorithm can be written as −

Step 1 − START ADD
Step 2 − get values of **a** & **b**
Step 3 − c ← a + b
Step 4 − display c
Step 5 − STOP

In design and analysis of algorithms, usually the second method is used to describe an algorithm. It makes it easy for the analyst to analyze the algorithm ignoring all unwanted definitions. He can observe what operations are being used and how the process is flowing.

Writing **step numbers**, is optional.

We design an algorithm to get a solution of a given problem. A problem can be solved in more than one ways.



Hence, many solution algorithms can be derived for a given problem. The next step is to analyze those proposed solution algorithms and implement the best suitable solution.

1.8 ALGORITHM COMPLEXITY

Suppose **X** is an algorithm and **n** is the size of input data, the time and space used by the algorithm X are the two main factors, which decide the efficiency of X.

- **Time Factor** − Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.
- **Space Factor** − Space is measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm **f(n)** gives the running time and/or the storage space required by the algorithm in terms of **n** as the size of input data.

The complexity of an algorithm **f(n)** gives the running time and/or the storage space required by the algorithm in terms of **n** as the size of input data.

1.8.1 Space Complexity

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. The space required by an algorithm is equal to the sum of the following two components −

- A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.
- A variable part is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stack space, etc.

Space complexity S(P) of any algorithm P is S(P) = C + SP(I), where C is the fixed part and S(I) is the variable part of the algorithm, which depends on instance characteristic I. Following is a simple example that tries to explain the concept −

Algorithm: SUM(A, B)
Step 1 - START
Step 2 - C ← A + B + 10
Step 3 - Stop

Here we have three variables A, B, and C and one constant. Hence S(P) = 1 + 3. Now, space depends on data types of given variables and constant types and it will be multiplied accordingly.

1.8.2 Time Complexity

Time complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function T(n), where T(n) can be measured as the number of steps, provided each step consumes constant time.

For example, addition of two n-bit integers takes **n** steps. Consequently, the total computational time is T(n) = c ∗ n, where c is the time taken for the addition of two bits. Here, we observe that T(n) grows linearly as the input size increases.

1.9 ALGORITHM ANALYSIS

Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation. They are the following –

- ***A Priori*** **Analysis** or Performance or Asymptotic Analysis − This is a theoretical analysis of an algorithm. Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.
- ***A Posterior*** **Analysis** or Performance Measurement − This is an empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.

We shall learn about *a priori* algorithm analysis. Algorithm analysis deals with the execution or running time of various operations involved. The running time of an operation can be defined as the number of computer instructions executed per operation.

Analysis of an algorithm is required to determine the amount of resources such as time and storage necessary to execute the algorithm. Usually, the efficiency or running time of an algorithm is stated as a function which relates the input length to the time complexity or space complexity.

Algorithm analysis framework involves finding out the time taken and the memory space required by a program to execute the program. It also determines how the input size of a program influences the running time of the program.

In theoretical analysis of algorithms, it is common to estimate their complexity in the asymptotic sense, i.e., to estimate the complexity function for arbitrarily large input. Big-O notation, Omega notation, and Theta notation are used to estimate the complexity function for large arbitrary input.

1.9.1 Types of Analysis

The efficiency of some algorithms may vary for inputs of the same size. For such algorithms, we need to differentiate between the worst case, average case and best case efficiencies.

1.9.1.1 Best Case Analysis

If an algorithm takes the least amount of time to execute a specific set of input, then it is called the best case time complexity. The best case efficiency of an algorithm is the efficiency for the best case input of size **n**. Because of this input, the algorithm runs the fastest among all the possible inputs of the same size.

1.9.1.2 Average Case Analysis

If the time complexity of an algorithm for certain sets of inputs are on an average, then such a time complexity is called average case time complexity.

Average case analysis provides necessary information about an algorithm's behavior on a typical or random input. You must make some assumption about the possible inputs of size **n** to analyze the average case efficiency of algorithm.

1.9.1.3 Worst Case Analysis

If an algorithm takes maximum amount of time to execute for a specific set of input, then it is called the worst case time complexity. The worst case efficiency of an algorithm is the efficiency for the worst case input of size **n**. The algorithm runs the longest among all the possible inputs of the similar size because of this input of size **n**.

1.10 MATHEMATICAL NOTATION

Algorithms are widely used in various areas of study. We can solve different problems using the same algorithm. Therefore, all algorithms must follow a standard. The mathematical notations use symbols or symbolic expressions, which have a precise semantic meaning.

1.10.1 Asymptotic Notations

A problem may have various algorithmic solutions. In order to choose the best algorithm for a particular process, you must be able to judge the time taken to run a particular solution. More accurately, you must be able to judge the time taken to run two solutions, and choose the better among the two.
To select the best algorithm, it is necessary to check the efficiency of each algorithm. The efficiency of each algorithm can be checked by computing its time complexity. The asymptotic notations help to represent the time complexity in a shorthand way. It can generally be represented as the fastest possible, slowest possible or average possible.

The notations such as O (Big-O), Ω (Omega), and θ (Theta) are called as asymptotic notations. These are the mathematical notations that are used in three different cases of time complexity.

1.10.1.1 Big-O Notation

'O' is the representation for Big-O notation. Big -O is the method used to express the upper bound of the running time of an algorithm. It is used to describe the performance or time complexity of the algorithm. Big-O specifically describes the worst-case scenario and can be used to describe the execution time required or the space used by the algorithm.

Table 2.1 gives some names and examples of the common orders used to describe functions. These orders are ranked from top to bottom.

Table 2.1: Common Orders

Time complexity			Examples
1	O(1)	Constant	Adding an element to the front of a linked list

Table 2.1: Common Orders			
Time complexity			**Examples**
1	O(1)	Constant	Adding to the front of a linked list
2	O(log n)	Logarithmic	Finding an entry in a sorted array
3	O(n)	Linear	Finding an entry in an unsorted array
4	O(n log n)	Linearithmic	Sorting 'n' items by 'divide-and-conquer'
5	O(n2)	Quadratic	Shortest path between two nodes in a graph
6	O(n3)	Cubic	Simultaneous linear equations
7	O(2n)	Exponential	The Towers of Hanoi problem

Big-O notation is generally used to express an ordering property among the functions. This notation helps in calculating the maximum amount of time taken by an algorithm to compute a problem. Big-O is defined as:

$$f(n) \leq c * g(n)$$

where, **n** can be any number of inputs or outputs and **f(n)** as well as **g(n)** are two non-negative functions. These functions are true only if there is a constant **c** and a non-negative integer **n0** such that,
$n \geq n0$.

The Big-O can also be denoted as f(n) = O(g(n)), where **f(n)** and **g(n)** are two non -negative functions and f(n) < g(n) if **g(n)** is multiple of some constant **c**. The graphical representation of f(n) = O(g(n)) is shown in figure 2.1, where the running time increases considerably when **n** increases**.**

Example: Consider f(n)=15n3+40n2+2nlog n+2n. As the value of **n** increases, n3 becomes much larger than n2, nlog n, and n. Hence, it dominates the function f(n) and we can consider the running time to grow by the order of n3. Therefore, it can be written as f(n)=O(n3).

The values of n for f(n) and C* g(n) will not be less than n0. Therefore, the values less than n0 are not considered relevant.

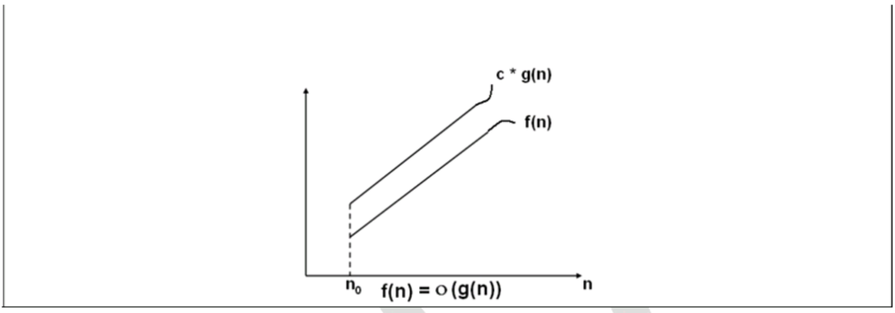


Figure 1.8: Big-O Notation f(n) = O(g(n))

Let us take an example to understand the Big-O notation more clearly.

Example:
Consider function f(n) = 2(n)+2 and g(n) = n2.

We need to find the constant c such that $f(n) \leq c * g(n)$.
Let n = 1, then
f(n) = 2(n)+2 = 2(1)+2 = 4

g(n) = n2 = 12 = 1
Here, f(n)>g(n)
Let n = 2, then
f(n) = 2(n)+2 = 2(2)+2 = 6
g(n) = n2 = 22 = 4

Here, f(n)>g(n)
Let n = 3, then
f(n) = 2(n)+2 = 2(3)+2 = 8
g(n) = n2 = 32 = 9
Here, f(n)<g(n)

Thus, when n is greater than 2, we get f(n)<g(n). In other words, as **n** becomes larger, the running time increases considerably. This concludes that the Big-O helps to determine the 'upper bound' of the algorithm's run-time.

Limitations of Big O Notation

There are certain limitations with the Big O notation of expressing the complexity of algorithms. These limitations are as follows:

- Many algorithms are simply too hard to analyse mathematically.
- There may not be sufficient information to calculate the behaviour of the algorithm in the average case.
- Big O analysis only tells us how the algorithm grows with the size of the problem, not how efficient it is, as it does not consider the programming effort.
- It ignores important constants. For example, if one algorithm takes O(n2) time to execute and the other takes O(100000n2) time to execute, then as per Big O, both algorithm have equal time complexity. In real-time systems, this may be a serious consideration.

1.10.1.2 Omega Notation

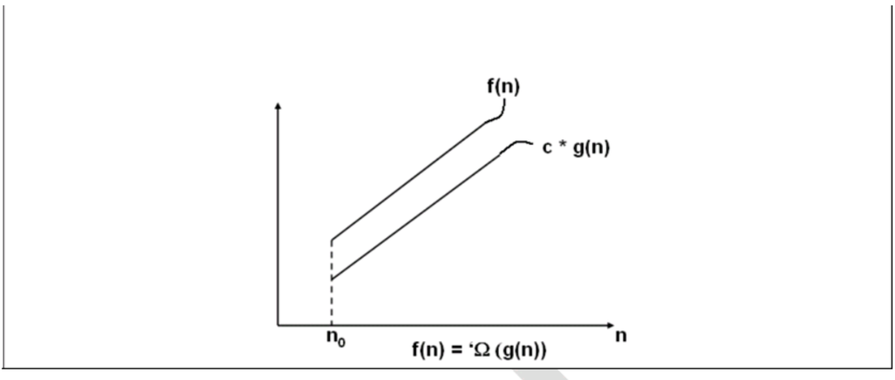
'Ω' is the representation for Omega notation. Omega describes the manner in which an algorithm performs in the best case time complexity. This notation provides the minimum amount of time taken by an algorithm to compute a problem. Thus, it is considered that omega gives the "lower bound" of the algorithm's run-time. Omega is defined as:

$$f(n) \geq c * g(n)$$

Where, **n** is any number of inputs or outputs and **f(n)** and **g(n)** are two non-negative functions. These functions are true only if there is a constant **c** and a non-negative integer **n0** such that n>n0.

Omega can also be denoted as f(n) = Ω (g(n)) where, **f** of **n** is equal to Omega of **g** of **n** . The graphical representation of f(n) = Ω (g(n)) is shown in figure 2.2. The function **f(n)** is said to be in Ω (g(n)), if **f(n)** is bounded below by some constant multiple of **g(n)** for all large values of **n**, i.e., if there exists some positive constant **c** and some non-negative integer **n0,** such that f(n) ≥ c * g(n) for all n ≥n0.
Figure 2.2 shows Omega notation.

Figure 1.9 Omega Notation f(n) = Ω (g(n))



Let us take an example to understand the Omega notation more clearly.

Example:
Consider function f(n) = 2n2+5 and g(n) = 7n.
We need to find the constant c such that f(n) ≥ c * g(n).

Let n = 0, then
f(n) = 2n2+5 = 2(0)2+5 = 5
g(n) = 7(n) = 7(0) = 0
Here, f(n)>g(n)
Let n = 1, then

f(n) = 2n2+5 = 2(1)2+5 = 7
g(n) = 7(n) = 7(1) = 7
Here, f(n)=g(n)
Let n = 2, then
f(n) = 2n2+5 = 2(2)2+5 = 13

1.10.1.3 Theta Notation

'θ' is the representation for Theta notation. Theta notation is used when the upper bound and lower bound of an algorithm are in the same order of magnitude. Theta can be defined as:

$$c1 * g(n) \leq f(n) \leq c2 * g(n) \quad \text{for all } n > n0$$

Where, **n** is any number of inputs or outputs and **f(n)** and **g(n)** are two non-negative functions. These functions are true only if there are two constants namely, **c1**, **c2,** and a non-negative integer **n0**.

Theta can also be denoted as f(n) = θ(g(n)) where, **f** of **n** is equal to Theta of **g** of **n**. The graphical representation of f(n) = θ(g(n)) is shown in figure 2.3. The function **f(n)** is said to be in θ (g(n)) if **f(n)** is bounded both above and below by some positive constant multiples of **g(n)** for all large values of **n**, i.e., if there exists some positive constant **c1** and **c2** and some non-negative integer **n0,** such that C2g(n)≤f(n)≤ C1g(n) for all n≥n0.

Figure shows Theta notation.

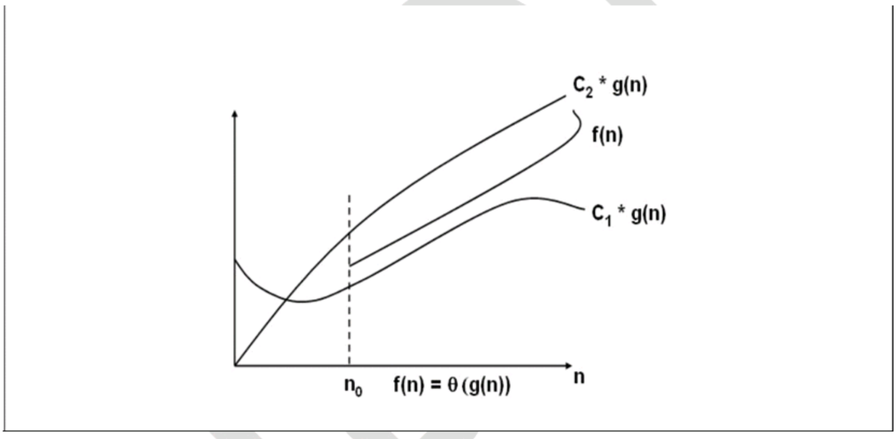


Figure 1.10: Theta Notation f(n) = θ(g(n))

Let us take an example to understand the Theta notation more clearly.

Example: Consider function f(n) = 4n + 3 and g(n) = 4n for all n ≥ 3; and f(n) = 4n + 3 and g(n) = 5n for all n ≥ 3.

Then the result of the function will be:
Let n = 3
f(n) = 4n + 3 = 4(3)+3 = 15
g(n) = 4n =4(3) = 12 and

f(n) = 4n + 3 = 4(3)+3 = 15
g(n) = 5n =5(3) = 15 and

here, c1 is 4, c2 is 5 and n0 is 3

Thus, from the above equation we get c1 *g*(n) *f*(n) c2 *g*(n). This concludes that Theta notation depicts the running time between the upper bound and lower bound.

1.11 ALGORITHM DESIGN TECHNIQUE

1.11.1 Divide and Conquer
1.11.2 Back Tracking Method
1.11.3 Dynamic programming

1.11.1 Divide and Conquer

Introduction

Divide and Conquer approach basically works on breaking the problem into sub problems that are similar to the original problem but smaller in size & simpler to solve. once divided sub problems are solved recursively and then combine solutions of sub problems

1.11 ALGORITHM DESIGN TECHNIQUE

1.11.1 Divide and Conquer
1.11.2 Back Tracking Method
1.11.3 Dynamic programming

1.11.1 Divide and Conquer

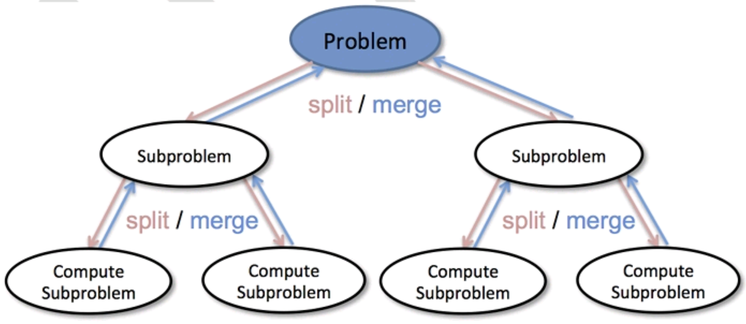
Introduction

Divide and Conquer approach basically works on breaking the problem into sub problems that are similar to the original problem but smaller in size & simpler to solve. once divided sub problems are solved recursively and then combine solutions of sub problems to create a solution to original problem.

At each level of the recursion the divide and conquer approach follows three steps:
Divide: In this step whole problem is divided into several sub problems.

Conquer: The sub problems are conquered by solving them recursively, only if they are small enough to be solved, otherwise step1 is executed.

Combine: In this final step, the solution obtained by the sub problems are combined to create solution to the original problem.



Generally, we can follow the **divide-and-conquer** approach in a three-step process.

Examples: The specific computer algorithms are based on the Divide & Conquer approach:

1. Maximum and Minimum Problem
2. Binary Search

3. Sorting (merge sort, quick sort)
4. Tower of Hanoi.

Fundamental of Divide & Conquer Strategy:

There are two fundamentals of Divide & Conquer Strategy:

1. Relational Formula
2. Stopping Condition

1. Relational Formula: It is the formula that we generate from the given technique. After generation of Formula, we apply D&C Strategy, i.e., we break the problem recursively & solve the broken subproblems.

2. Stopping Condition: When we break the problem using Divide & Conquer Strategy, then we need to know that for how much time, we need to apply divide & Conquer. So, the condition where the need to stop our recursion steps of D&C is called as Stopping Condition.

Applications of Divide and Conquer Approach:

Following algorithms are based on the concept of the Divide and Conquer Technique:

1. **Binary Search:** The binary search algorithm is a searching algorithm, which is also called a half-interval search or logarithmic search. It works by comparing the target value with the middle element existing in a sorted array. After making the comparison, if the value differs, then the half that cannot contain the target will eventually eliminate, followed by continuing the search on the other half. We will again consider the middle element and compare it with the target value. The process keeps on repeating until the target value is met. If we found the other half to be empty after ending the search, then it can be concluded that the target is not present in the array.
2. **Quicksort:** It is the most efficient sorting algorithm, which is also known as partition-exchange sort. It starts by selecting a pivot value from an array followed

Applications of Divide and Conquer Approach:

Following algorithms are based on the concept of the Divide and Conquer Technique:

1. **Binary Search:** The binary search algorithm is a searching algorithm, which is also called a half-interval search or logarithmic search. It works by comparing the target value with the middle element existing in a sorted array. After making the comparison, if the value differs, then the half that cannot contain the target will eventually eliminate, followed by continuing the search on the other half. We will again consider the middle element and compare it with the target value. The process keeps on repeating until the target value is met. If we found the other half to be empty after ending the search, then it can be concluded that the target is not present in the array.
2. **Quicksort:** It is the most efficient sorting algorithm, which is also known as partition-exchange sort. It starts by selecting a pivot value from an array followed by dividing the rest of the array elements into two sub-arrays. The partition is made by comparing each of the elements with the pivot value. It compares whether the element holds a greater value or lesser value than the pivot and then sort the arrays recursively.
3. **Merge Sort:** It is a sorting algorithm that sorts an array by making comparisons. It starts by dividing an array into sub-array and then recursively sorts each of them. After the sorting is done, it merges them back.

Advantages of Divide and Conquer

- Divide and Conquer tend to successfully solve one of the biggest problems, such as the Tower of Hanoi, a mathematical puzzle. It is challenging to solve complicated problems for which you have no basic idea, but with the help of the divide and conquer approach, it has lessened the effort as it works on dividing the main problem into two halves and then solve them recursively. This algorithm is much faster than other algorithms.
- It efficiently uses cache memory without occupying much space because it solves simple subproblems within the cache memory instead of accessing the slower main memory.

Disadvantages of Divide and Conquer

- Since most of its algorithms are designed by incorporating recursion, so it necessitates high memory management.
- An explicit stack may overuse the space.
- It may even crash the system if the recursion is performed rigorously greater than the stack present in the CPU.

1.11.2 Backtracking

Introduction

The Backtracking is an algorithmic-method to solve a problem with an additional way. It uses a recursive approach to explain the problems. We can say that the backtracking is needed to find all possible combination to solve an optimization problem.

Backtracking is a systematic way of trying out different sequences of decisions until we find one that "works."

In the following Figure:

- Each non-leaf node in a tree is a parent of one or more other nodes (its children)
- Each node in the tree, other than the root, has exactly one parent

1.11.2 Backtracking

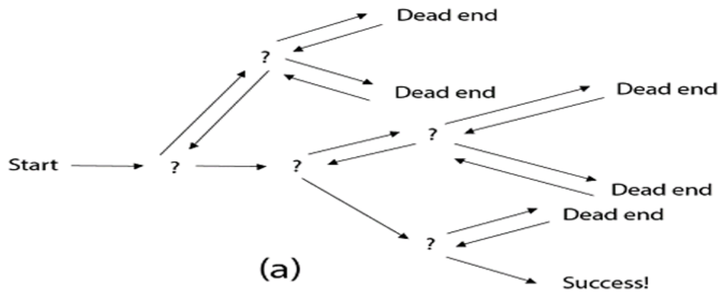
Introduction

The Backtracking is an algorithmic-method to solve a problem with an additional way. It uses a recursive approach to explain the problems. We can say that the backtracking is needed to find all possible combination to solve an optimization problem.

Backtracking is a systematic way of trying out different sequences of decisions until we find one that "works."

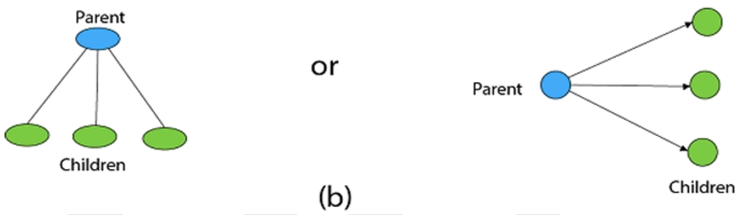
In the following Figure:

- Each non-leaf node in a tree is a parent of one or more other nodes (its children)
- Each node in the tree, other than the root, has exactly one parent



(a)

Generally, however, we draw our trees downward, with the root at the top.



(b)

A tree is composed of nodes.

1.11.3 Dynamic programming

Dynamic Programming Technique is similar to divide-and-conquer technique. Both techniques solve a problem by breaking it down into several sub-problems that can be solved recursively. The main difference between is that, Divide & Conquer approach partitions the problems into independent sub-problems, solve the sub-problems recursively, and then combine their solutions to solve the original problems. Whereas dynamic programming is applicable when the sub-problems are not independent, that is, when sub-problems share sub subproblems. Also, A dynamic programming algorithms solves every sub problem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the sub subproblems is encountered.

Therefore **"Dynamic programming is a applicable when sub problem are not independent, that is when sub problem share sub problems."**

As Greedy approach, Dynamic programming is typically applied to optimization problems and for them there can be many possible solutions and the requirement is to find the optimal solution among those. But Dynamic programming approach is little different greedy approach. In greedy solutions are computed by making choices in serial forward way and in this no backtracking & revision of choices is done where as Dynamic programming computes its solution bottom up by producing them from smaller sub problems, and by trying many possibilities and choices before it arrives at the optimal set of choices.

The Development of a dynamic-programming algorithm can be broken into a sequence of four steps:
Divide, Sub problems: The main problems are divided into several smaller sub problems. In this the solution of the main problem is expressed in terms of the solution for the smaller sub problems. Basically, it is all about characterizing the structure of an optimal solution and recursively define the value of an optimal solution.

Table, Storage: The solution for each sub problem is stored in a table, so that it can be used many times whenever required.

Combine, bottom-up Computation: The solution to main problem is obtained by combining the solutions of smaller sub problems. i.e., compute the value of an optimal solution in a bottom-up fashion.

Construct an optimal solution from computed information. **(This step is optional and is required in case if some additional information is required after finding out optimal solution.)**

Now for any problem to be solved through dynamic programming approach it must follow the following conditions:

Principle of Optimality: It states that for solving the master problem optimally, its sub problems should be solved optimally. It should be noted that not all the times each sub problem(s) is solved optimally, so in that case we should go for optimal majority.

Polynomial Breakup: For solving the main problem, the problem is divided into several sub problems and for efficient performance of dynamic programming the total number of sub problems to be solved should be at-most a polynomial number.

Various algorithms which make use of Dynamic programming technique are as follows:

1. Knapsack problem.
2. Chain matrix multiplication.
3. All pair shortest path.
4. Travelling sales man problem.
5. Tower of hanoi.
6. Checker Board.
7. Fibonacci Sequence.
8. Assembly line scheduling.
9. Optimal binary search trees.

1.12 SUMMARY

A data structure is a particular way of storing and organizing data either in computer's memory or on the disk storage so that it can be used efficiently.
There are two types of data structures: primitive and non-primitive data structures. Primitive data structures are the fundamental data types which are supported by a programming language. Nonprimitive data structures are those data structures which are created using primitive data structures.
Non-primitive data structures can further be classified into two categories: linear and non-linear data structures.
If the elements of a data structure are stored in a linear or sequential order, then it is a

Unit 2: **CHAPTER-2**

Sorting Techniques

2.0 OBJECTIVE :

After studying this unit, you will be able to:

- To study basic concepts of sorting.
- To study different sorting methods and their algorithms.
- Study different sorting methods and compare with their time complexity

2.1 INTRODUCTION TO SORTING

Arranging the data in ascending or descending order is known as sorting.
Sorting is very important from the point of view of our practical life.
The best example of sorting can be phone numbers in our phones. If, they are not maintained in an alphabetical order we would not be able to search any number effectively.
There are two types of sorting:
Internal Sorting:
If all the data that is to be sorted can be adjusted at a time in the main memory, the internal sorting method is being performed.
External Sorting:

When the data that is to be sorted cannot be accommodated in the memory at the same time and some has to be kept in auxiliary memory such as hard disk, floppy disk, magnetic tapes etc, then external sorting methods are performed.

Application of sorting
1. The sorting is useful in database applications for arranging the data in desired ORDER.
2. In the dictionary like applications the data is arranged in sorted order.
3. For searching the element from the list of elements the sorting is required
4. For checking the uniqueness of the element the sorting is required.
5. For finding the closest pair from the list of elements the sorting is required.

2.2 SORTING TECHNIQUES

1) Bubble sort
2) Insertion sort
3) Radix Sort
4) Quick sort
5) Merge sort
6) Heap sort
7) Selection sort
8) shell Sort

2.2.1 BUBBLE SORT

In *bubble sorting*, consecutive adjacent pairs of elements in the array are compared with each other. If the element at the lower index is greater than the element at the higher index, the two elements are interchanged so that the element is placed before the bigger one. This process will continue till the list of unsorted elements exhausts.

This procedure of sorting is called bubble sorting because elements ‘bubble’ to the

2.2.1 BUBBLE SORT

In *bubble sorting*, consecutive adjacent pairs of elements in the array are compared with each other. If the element at the lower index is greater than the element at the higher index, the two elements are interchanged so that the element is placed before the bigger one. This process will continue till the list of unsorted elements exhausts.

This procedure of sorting is called bubble sorting because elements 'bubble' to the top of the list. Note that at the end of the first pass, the largest element in the list will be placed at its proper position (i.e., at the end of the list).

Note : If the elements are to be sorted in descending order, then in first pass the smallest element is movedto the highest index of the array.

Example To discuss bubble sort in detail, let us consider an arrayA[]that has the followingelements:

A[] = {30, 52, 29, 87, 63, 27, 19, 54}

Pass 1:

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

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

Compare 87 and 27. Since 87 > 27, swapping is
done. 30, 29, 52, 63, **27, 87**

2.2.1 BUBBLE SORT

In *bubble sorting*, consecutive adjacent pairs of elements in the array are compared with each other. If the element at the lower index is greater than the element at the higher index, the two elements are interchanged so that the element is placed before the bigger one. This process will continue till the list of unsorted elements exhausts.

This procedure of sorting is called bubble sorting because elements 'bubble' to the top of the list. Note that at the end of the first pass, the largest element in the list will be placed at its proper position (i.e., at the end of the list).

Note : If the elements are to be sorted in descending order, then in first pass the smallest element is movedto the highest index of the array.

Example To discuss bubble sort in detail, let us consider an arrayA[]that has the followingelements:

A[] = {30, 52, 29, 87, 63, 27, 19, 54}

Pass 1:

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

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

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

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

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

Observe that after the end of the first pass, the largest element is placed at the highest index of the array. All the other elements are still unsorted.

Pass 2:

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

Compare 30 and 52. Since 30 < 52, no swapping is done.

Compare 52 and 63. Since 52 < 63, no swapping is done.

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

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

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

Observe that after the end of the second pass, the second largest element is placed at the second highest index of the array. All the other elements are still unsorted.

Pass 3:

Pass 3:

Compare 29 and 30. Since 29 < 30, no swapping is done.

Compare 30 and 52. Since 30 < 52, no swapping is done.

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

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

Compare 52 and 54. Since 52 < 54, no swapping is done.

Observe that after the end of the third pass, the third largest element is placed at the third highest index of the array. All the other elements are still unsorted.

Pass 4:

Compare 29 and 30. Since 29 < 30, no swapping is done.

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

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

Compare 30 and 52. Since 30 < 52, no swapping is done.

Observe that after the end of the fourth pass, the fourth largest element is placed at the fourth highest index of the array. All the other elements are still unsorted.

Pass 5:

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

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

Compare 29 and 30. Since 29 < 30, no swapping is done.

Observe that after the end of the fifth pass, the fifth largest element is placed at the fifth highest index of the array. All the other elements are still unsorted.

Pass 6:

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

Compare 27 and 29. Since 27 < 29, no swapping is done.

Observe that after the end of the sixth pass, the sixth largest element is placed at the sixth largest index of the array. All the other elements are still unsorted.

Pass 7:

(a) Compare 19 and 27. Since 19 < 27, no swapping is done.

Observe that the entire list is sorted now.

Algorithm for bubble sort
BUBBLE_SORT(A, N) Step 1: Repeat Step 2 For 1 = 0 to N-1

Pass 7:

(a) Compare 19 and 27. Since 19 < 27, no swapping is done.

Observe that the entire list is sorted now.

Algorithm for bubble sort
BUBBLE_SORT(A, N) Step 1: Repeat Step 2 For 1 = 0 to N-1 Step 2: Repeat For J = to N – I Step 3: IF A[J] >A[J+ 1] SWAP A[J] and A[J+1] [END OF INNER LOOP] [END OF OUTER LOOP] Step 4: EXIT

Advantages :

- Simple and easy to implement
- In this sort, elements are swapped in place without using additional temporary storage, so the space requirement is at a minimum.

Disadvantages :

- It is slowest method . $O(n^2)$
- Inefficient for large sorting lists.

Program

```
#include<stdio.h>
void main ()
{
   int i, j,temp;
   int a[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
   for(i = 0; i<10; i++)
   {
      for(j = i+1; j<10; j++)
      {
         if(a[j] > a[i])
         {
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
         }
      }
   }
   printf("Printing Sorted Element List ...\n");
   for(i = 0; i<10; i++)
   {
      printf("%d\n",a[i]);
   }
}
```

Output:

```
Printing Sorted Element List . . .
7
9
10
12
23
34
34
```

Complexity of Bubble Sort

The complexity of any sorting algorithm depends upon the number of comparisons. In bubble sort, we have seen that there are N–1 passes in total. In the first pass, N–1comparisons are made to place the highest element in its correct position. Then, in Pass 2, there are N–2 comparisons and the second highest element is placed in its position. Therefore, to compute the complexity of bubble sort, we need to calculate the total number of comparisons. It can be given as:

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

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

$f(n) = n2/2 + O(n) = O(n2)$

Therefore, the complexity of bubble sort algorithm is O(n2). It means the time required to execute bubble sort is proportional to n2, where n is the total number of elements in the array.

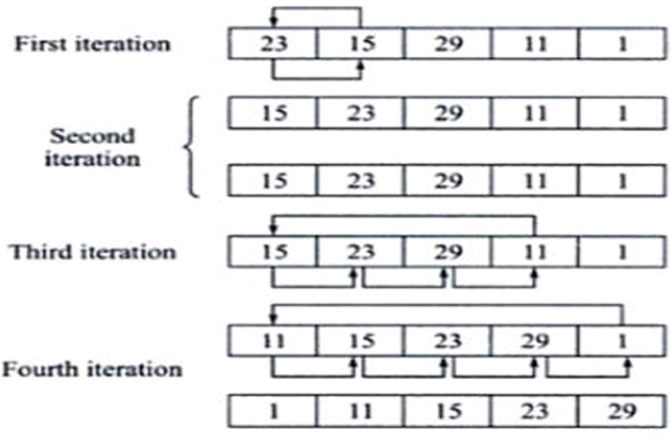
2.2.2 INSERTION SORT

Insertion sort is a very simple sorting algorithm in which the sorted array (or list) is built one element at a time. We all are familiar with this technique of sorting, as we usually use it for ordering a deck of cards while playing bridge.

Insertion sort inserts each item into its proper place in the final list. In insertion sort , the first iteration starts with comparison of 1^{st} element with 0^{th} element. In the second iteration 2^{nd} element is compared with the 0^{th} and 1^{st} element and so on. In every iteration an element is compared with all elements. The main idea is to insert the i^{th} pass the i^{th} element in A[1], A[2]…A[i] in its proper place.

Example Consider an array of integers given below. We will sort the values in the arrayusing insertion sort

23 15 29 11 1



Algorithm for insertion sort

INSERTION-SORT (ARR, N)

Step 1: Repeat Steps 2 to 5 for K = 1 to N – 1
Step 2: SET TEMP = ARR[K]
Step 3: SET J = K - 1
Step 4: Repeat while TEMP <= ARR[J]
SET ARR[J + 1] = ARR[J]
SET J = J - 1
[END OF INNER LOOP]
Step 5: SET ARR[J + 1] = TEMP
[END OF LOOP]
Step 6: EXIT

Advantages:

Program

```
#include<stdio.h>
void main ()
{
   int i,j, k,temp;
   int a[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
   printf("\nprinting sorted elements...\n");
   for(k=1; k<10; k++)
   {
      temp = a[k];
      j= k-1;
      while(j>=0 && temp <= a[j])
      {
         a[j+1] = a[j];
         j = j-1;
      }
      a[j+1] = temp;
   }
   for(i=0;i<10;i++)
   {
      printf("\n%d\n",a[i]);
   }
}
```

Output:

```
Printing Sorted Elements . . .
7
9
10
12
23
23
34
44
78
101
```

Complexity of Insertion Sort

If the initial tile is sorted, only one comparison is made on each iteration, so that the sort is O(n). If the file is initially sorted in the reverse order the worst case complexity is $O(n^2)$. Since the total number of comparisons is:

$(n-1)+ (n-2) +...+3+2+1= (n-1) * n/2$, which is $O (n^2)$

The average case or the average number of comparisons in the simple insertion sort is $O (n^2)$.

2.2.3 SELECTION SORT

Selection sorting is conceptually the simplest sorting algorithm. This algorithm first finds the smallest element in the array and exchanges it with the element in the first position, then finds the second smallest element and exchange it with the element in the second position, and continues in this way until the entire array is sorted.

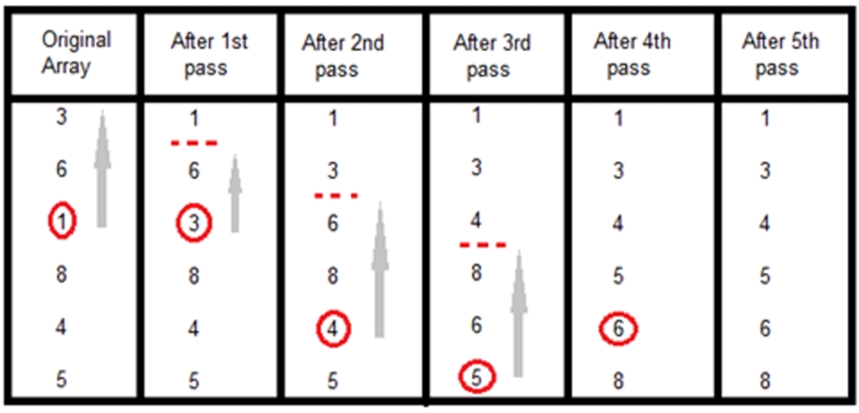
Example 1 : 3, 6, 1, 8, 4, 5

Original Array	After 1st pass	After 2nd pass	After 3rd pass	After 4th pass	After 5th pass
3	1	1	1	1	1
6	6	3	3	3	3
1	3	6	4	4	4

2.2.3 SELECTION SORT

Selection sorting is conceptually the simplest sorting algorithm. This algorithm first finds the smallest element in the array and exchanges it with the element in the first position, then finds the second smallest element and exchange it with the element in the second position, and continues in this way until the entire array is sorted.

Example 1 : 3, 6, 1, 8, 4, 5



Algorithm for selection sort

SELECTION SORT(ARR, N)

Step 1: Repeat Steps 2 and 3 for K = 1 to N-1
Step 2: CALL SMALLEST(ARR, K, N, POS)
Step 3: SWAP A[K] with ARR[POS]

[END OF LOOP]
Step 4: EXIT

SMALLEST (ARR, K, N, POS)

Step 1: [INITIALIZE] SET SMALL = ARR[K]
Step 2: [INITIALIZE] SET POS = K
Step 3: Repeat for J = K+1 to N
IF SMALL > ARR[J]
SET SMALL = ARR[J]
SET POS = J
[END OF IF]
[END OF LOOP]
Step 4: RETURN POS

Advantages:

- It is simple and easy to implement.
- It can be used for small data sets.
- It is 60 per cent more efficient than bubble sort.

Disadvantages:

- Running time of Selection sort algorithm is very poor of 0 (n2).
- However, in case of large data sets, the efficiency of selection sort drops as compared to insertion sort.

Program

```
#include<stdio.h>
int smallest(int[],int,int);
void main ()
{
  int a[10] = {10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
  int i,j,k,pos,temp;
  for(i=0;i<10;i++)
  {
    pos = smallest(a,10,i);
    temp = a[i];
    a[i]=a[pos];
```

```
    }
    printf("\nprinting sorted elements...\n");
    for(i=0;i<10;i++)
    {
        printf("%d\n",a[i]);
    }
}
int smallest(int a[], int n, int i)
{
    int small,pos,j;
    small = a[i];
    pos = i;
    for(j=i+1;j<10;j++)
    {
        if(a[j]<small)
        {
            small = a[j];
            pos=j;
        }
    }
    return pos;
}
```

Output:

```
printing sorted elements...
7
9
10
12
23
23
34
44
78
101
```

Complexity of Selection Sort

The first element is compared with the remaining n-1 elements in pass 1. Then n-2 elements are taken in pass 2, this process is repeated until the last element is encountered. The mathematical expression for these iterations will be equal to: (n-1)+(n-2)+….,+(n-(n-1)).Thus the expression become n*(n-1)/2. Thus, the number of comparisons is proportional to (n^2). The time complexity of selection sort is O(n^2).

2.2.4 MERGE SORT

Merging means combining two sorted lists into one-sorted list. The merge sort splits the array to be sorted into two equal halves and each array is recursively sorted, then merged back together to form the final sorted array. The logic is to split the array into two sub arrays each sub array is individually sorted and the resulting sequence is then combined to produce a single sorted sequence of n elements.

The merge sort recursively follows the steps:
1) Divide the given array into equal parts.
2) Recursively sorts the elements on the left side of the partitions.
3) Recursively sorts the elements on the right side of the partitions.
4) Combine the sorted left and right partitions into a single sorted array.

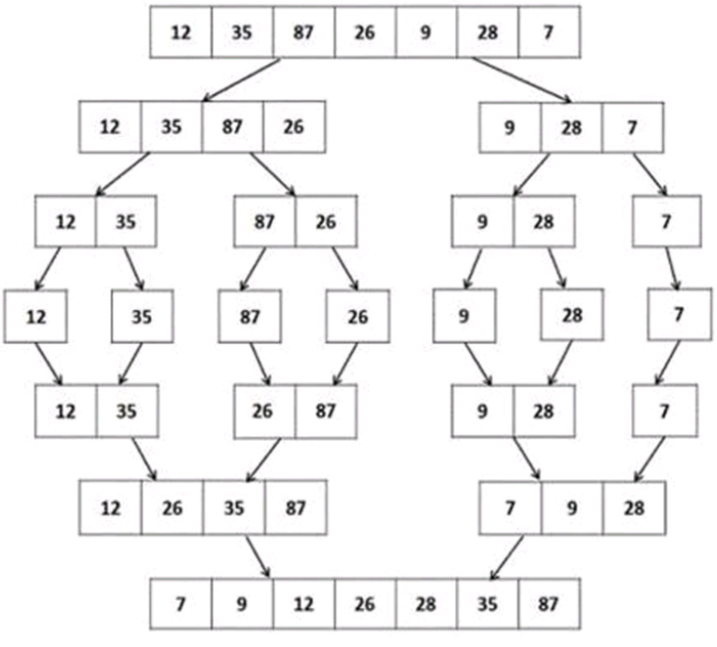
Example Sort the array given below using merge sort.
12 , 35 ,87, 26, 9, 28,7

2.2.4 MERGE SORT

Merging means combining two sorted lists into one-sorted list. The merge sort splits the array to be sorted into two equal halves and each array is recursively sorted, then merged back together to form the final sorted array. The logic is to split the array into two sub arrays each sub array is individually sorted and the resulting sequence is then combined to produce a single sorted sequence of n elements.

The merge sort recursively follows the steps:
1) Divide the given array into equal parts.
2) Recursively sorts the elements on the left side of the partitions.
3) Recursively sorts the elements on the right side of the partitions.
4) Combine the sorted left and right partitions into a single sorted array.

Example Sort the array given below using merge sort.
12 , 35 ,87, 26, 9, 28,7



Merge Sort

Algorithm for merge sort

MERGE_SORT(ARR, BEG, END)
Step 1: IF BEG < END
SET MID = (BEG + END)/2
CALL MERGE_SORT (ARR, BEG, MID)

CALL MERGE_SORT (ARR, MID + 1, END)
MERGE (ARR, BEG, MID, END)
[END OF IF]
Step 2: END

```
MERGE (ARR, BEG, MID, END)

Step 1: [INITIALIZE] SET I = BEG, J = MID + 1, INDEX = 0
Step 2: Repeat while (I <= MID) AND (J<=END)
            IF ARR[I] < ARR[J]
                SET TEMP[INDEX] = ARR[I]
                SETI=I+1
            ELSE
                SET TEMP[INDEX] = ARR[J]
                SETJ=J+1
            [END OF IF]
            SET INDEX = INDEX + 1
        [END OF LOOP]
Step 3: [Copy the remaining elements of right sub-array, if any]
            IFI>MID
              Repeat while J <= END
                SET TEMP[INDEX] = ARR[J]
                SET INDEX = INDEX + 1, SET J = J + 1
              [END OF LOOP]
        [Copy the remaining elements of left sub-array, if any]
            ELSE
              Repeat while I <= MID
                SET TEMP[INDEX] = ARR[I]
                SET INDEX = INDEX + 1, SET I = I + 1
              [END OF LOOP]
            [END OF IF]
Step 4: [Copy the contents of TEMP back to ARR] SET K=0
Step 5: Repeat while K < INDEX
            SET ARR[K] = TEMP[K]
            SETK=K+1
        [END OF LOOP]
Step 6: END
```

Advantages :

- Merge sort algorithm is best case for sorting slow-access data e.g) tape drive.
- Merge sort algorithm is better at handling sequential - accessed lists.

Disadvantages:

- Slower comparative to the other sort algorithms for smaller tasks.
- Merge sort algorithm requires additional memory space of 0(n) for the temporary array .
- It goes through the whole process even if the array is sorted.

Program

```
#include<stdio.h>
void mergeSort(int[],int,int);
void merge(int[],int,int,int);
void main ()
{
   int a[10]= {10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
   int i;
   mergeSort(a,0,9);
   printf("printing the sorted elements");
   for(i=0;i<10;i++)
   {
      printf("\n%d\n",a[i]);
   }

}
void mergeSort(int a[], int beg, int end)
{
   int mid;
   if(beg<end)
   {
      mid = (beg+end)/2;
      mergeSort(a,beg,mid);
      mergeSort(a,mid+1,end);
      merge(a,beg,mid,end);
   }
}
void merge(int a[], int beg, int mid, int end)
{
   int i=beg,j=mid+1,k,index = beg;
   int temp[10];
   while(i<=mid && j<=end)
   {
      if(a[i]<a[j])
      {
         temp[index] = a[i];
         i = i+1;
      }
      else
      {
         temp[index] = a[j];
         j = j+1;
      }
      index++;
   }
   if(i>mid)
   {
      while(j<=end)
      {
         temp[index] = a[j];
         index++;
         j++;
      }
   }
```

```
        }
    }
    else
    {
        while(i<=mid)
        {
            temp[index] = a[i];
            index++;
            i++;
        }
    }
    k = beg;
    while(k<index)
    {
        a[k]=temp[k];
        k++;
    }
}
```

Output:

```
printing the sorted elements

7
9
10
12
23
23
34
44
78
101
```

Complexity of Merge Sort

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