Maharaja Education Trust ®
MIT First Grade College
Department of Computer Applications
Industrial Suburb, Manandavadi Road, Mysuru -570008, www.mitfgc.in
Affiliated to University of Mysore, Accredited by NAAC with "A" Grade, ISO 9001:2015 Certified Institution

VISION OF THE INSTITUTE

Empower the individuals and society at large through educational excellence; sensitize them for a life dedicated to the service of fellow human beings and mother land.

MISSION OF THE INSTITUTE

To impact holistic education that enables the students to become socially responsive and useful, with roots firm on traditional and cultural values; and to hone their skills to accept challenges and respond to opportunities in a global scenario.

Design and Analysis of algorithm

Course Code: DSC501 - Theory

Total Contact Hours: 4 Hours/wk

Formative Assessment Marks: 25

Exam Marks: 25

Exam Duration: 03 Hrs

Prepared by:

ARVIND G

HOD Department of Computer Application

Program Name	**BCA**	Semester	**V**
Course Title	**Design and Analysis of Algorithm (Theory)**		
Course Code:	**DSC13**	No. of Credits	**04**
Contact hours	**52 Hours**	Duration of SEA/Exam	**2 hours**
Formative Assessment Marks	**40**	Summative Assessment Marks	**60**

Course Outcomes (COs): After the successful completion of the course, the student will be able to: CO1. Understand the fundamental concepts of algorithms and their complexity, including time and space complexity, worst-case and average-case analysis, and Big-O notation. BL (L1, L2)

CO2. Design algorithms for solving various types of problems, such as Sorting, Searching, Graph traversal, Decrease-and-Conquer, Divide-and-Conquer and Greedy Techniques. BL (L1, L2, L3) CO3. Analyze and compare the time and space complexity of algorithms with other algorithmic techniques. BL (L1, L2,L3,L4)

CO4. Evaluate the performance of Sorting, Searching, Graph traversal, Decrease-and-Conquer, Divide-and-Conquer and Greedy Techniques using empirical testing and benchmarking, and identify their limitations and potential improvements. BL (L1, L2, L3, L4)

CO5. Apply various algorithm design to real-world problems and evaluate their effectiveness and efficiency in solving them. BL (L1, L2, L3)

Note: Blooms Level(BL): L1=Remember, L2=Understand, L3=Apply, L4=Analyze, L5= Evaluate, L6= Create

Contents	**52 Hrs**
Introduction: What is an Algorithm? Fundamentals of Algorithmic problem solving, Fundamentals of the Analysis of Algorithm Efficiency, Analysis Framework, Measuring the input size, Units for measuring Running time, Orders of Growth, Worst-case, Best-case and Average-case efficiencies.	10
Asymptotic Notations and Basic Efficiency classes, Informal Introduction, O-notation, Ω-notation, θ-notation, mathematical analysis of non-recursive algorithms, mathematical analysis of recursive algorithms.	10
Brute Force & Exhaustive Search: Introduction to Brute Force approach, Selection Sort and Bubble Sort, Sequential search, Exhaustive Search- Travelling Salesman Problem and Knapsack Problem, Depth First Search, Breadth First Search	11
Decrease-and-Conquer: Introduction, Insertion Sort, Topological Sorting **Divide-and-Conquer:** Introduction, Merge Sort, Quick Sort, Binary Search, Binary Tree traversals and related properties.	11
Greedy Technique: Introduction, Prim's Algorithm, Kruskal's Algorithm, Dijkstra's Algorithm, Lower-Bound Arguments, Decision Trees, P Problems, NP Problems, NP- Complete Problems, Challenges of Numerical Algorithms.	10

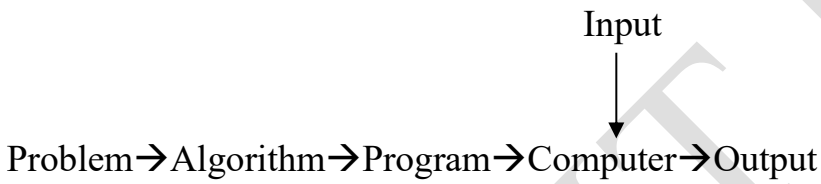
UNIT – 1

Introduction: What is an Algorithm? Fundamentals of Algorithmic problem solving, Fundamentals of the Analysis of Algorithm Efficiency, Analysis Framework, Measuring the input size, Units for measuring Running time, Orders of Growth, Worst-case, Best case and Average-case efficiencies.

INTRODUCTION

What is an algorithm?

Definition: An algorithm is defined as finite sequence of unambiguous instructions followed to accomplish a given task. It is also defined as unambiguous, step by step procedure (instructions) to solve a given problem in finite number of steps by accepting a set of inputs and producing the desired output. After producing the result, the algorithm should terminate. The notion of an algorithm is pictorially represented as shown below:



Observe the following activities to see how an algorithm is used to produce the desired result:

- The solution to a given problem is expressed in the form of an algorithm.
- The algorithm is converted into a program.
- The program when it is executed, accept the input and produces the desired output

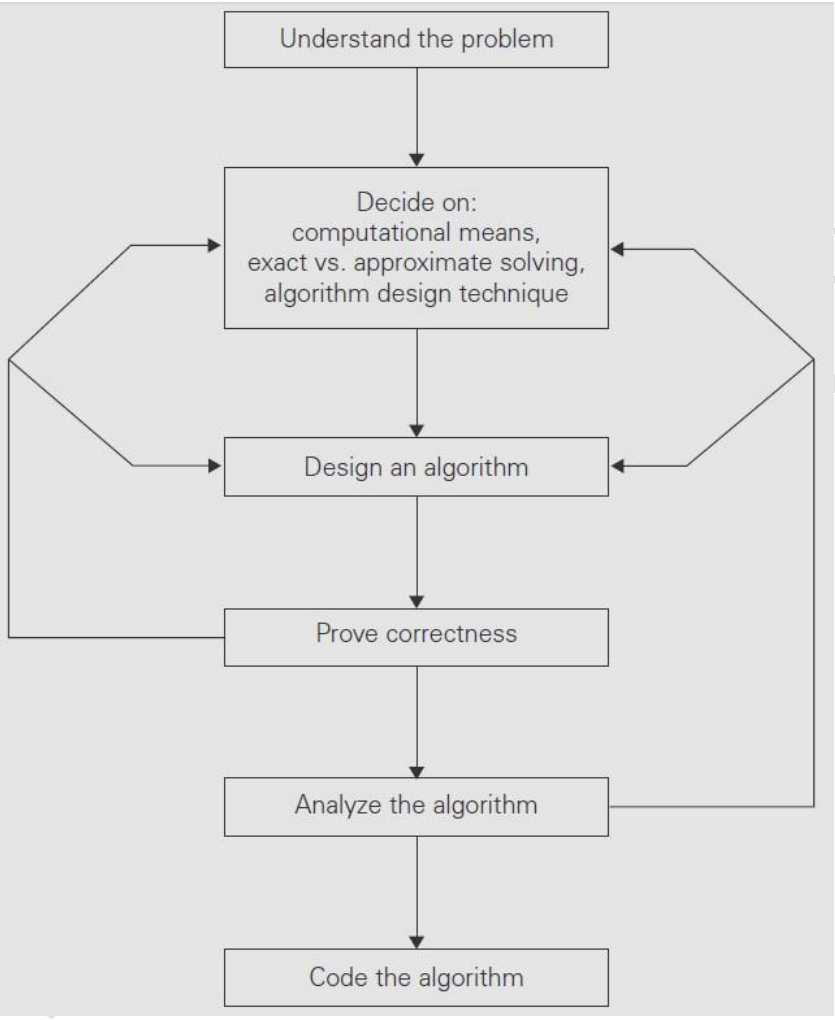
The properties of an algorithm?" An algorithm must satisfy the

- **Input:** Each algorithm should have zero or more inputs. The range of inputs for which algorithm works should be satisfied
- **Output:** The algorithm should produce correct results. At least one output has to be produced.
- **Definiteness:** Each instruction should be clear and unambiguous.
- **Effectiveness**: The instructions should be simple and should transform the given input to the desired output.
- **Finiteness:** The algorithm must terminate after a finite sequence of instructions.

Note: By looking at the algorithm, the programmer can write the program in C or C++ or any of the programming language. Before writing any program, the solution has to be expressed in the form of algorithms.

Algorithm design and analysis process

A sequence of steps involved in designing and analyzing an algorithm is shown inthe figure below.**:**



- Understand the problem
- Decide on Computational Device Exact Vs Approximate Algorithms
- Algorithm Design Techniques
- Design an algorithms
- Prove Correctness
- Analyze the Algorithm
- Code the Algorithm

Understanding the Problem:

Begin by clearly understanding the problem you are trying to solve. Identify the input, output, and constraints. This step is crucial for defining the scope of the algorithm.

Deciding on Computational Device:

Consider the computational device or model on which the algorithm will run. Different devices may have different resource constraints, and the algorithm should be designed to work efficiently within those constraints.

Exact Vs. Approximate Algorithms:

Decide whether an exact solution is necessary or if an approximate solution would suffice. Exact algorithms guarantee optimal solutions, while approximate algorithms provide solutions that are close to optimal but might be obtained faster.

Algorithm Design Techniques:

Choose appropriate algorithm design techniques based on the problem at hand. Common techniques include:

- **Divide and Conquer**: Break the problem into smaller subproblems and solve them independently.
- **Dynamic Programming:** Solve subproblems and store their solutions to avoid redundant computations.
- **Greedy Algorithms:** Make locally optimal choices at each step with the hope of finding a global optimum.

Design an Algorithm:

Develop the step-by-step procedure to solve the problem based on the chosen design technique. Clearly define the algorithm's input, output, and the sequence of operations to achieve the desired result.

Prove Correctness:

Provide a mathematical or logical proof that the algorithm produces the correct output for any valid input. This step ensures the algorithm reliably solves the problem it was designed for.

Analyze the Algorithm:

Assess the algorithm's efficiency, primarily in terms of time and space complexity. Perform a worst-case, best-case, and average-case analysis to understand how the algorithm performs under different scenarios.

Code the Algorithm:

Implement the algorithm in a programming language of choice. Pay attention to details, and ensure that the code accurately reflects the designed algorithm. Test the algorithm with various inputs to validate its correctness and efficiency.

Computing GCD

The notion of an algorithm can be explained by computing the GCD of two numbers. Now, let us see "What is GCD of two numbers?"

Definition: The GCD (short form for Greatest Common Divisor) of two numbers m and n denoted by GCD(m, n) is defined as the largest integer that divides both m and n such that the remainder is zero. GCD of two numbers is defined only for positive integers but, not defined for negative integers and floating point numbers. For example, GCD(10, 30) can be obtained as shown below:

Step 1: The numbers 1, 2, 5, 10 divide 10

Step 2: The numbers 1, 2, 5, 6, 10 and 30 divide 30

Step 3: Observe from step I and step 2 that the numbers 1, 2, 5 and 10 are common divisors of both 10 and 30 and 10 is the greatest number which is common and hence it is called Greatest Common Divisor

So, GCD(10,30) = 10.

Now, let us see "What are the different ways of computing GCD of two numbers?" The GCD of two numbers can be computed using various methods as shown below:

Different ways of computing GCD

- Euclid's algorithm (Using modulus)
- Repetitive subtraction (Euclid's algorithm)
- Consecutive inter checking algorithm
- Middle school procedure using prime factors

Step 2: [Eliminate the multiples of p between 2 to n]

```
for p = 21 sqrt(n)
    if(a * [p]! = 0)                    // Is a[p] is prime? If so proceed
        i← p* p                         // p is the next prime number obtained
        while (i<= n)                   // Obtain position of multiples of p
            a[i] ←0                     // Multiples of p may exist
            i←i + p                     // Eliminate a[i] which is multiple of p
        End while                       // Obtain the position of next multiple of p
    End if
End for
```

Step 3: [Obtain the prime numbers by copying the non-zero elements]

```
j←0
for i←2 to n do
        if(a * [i]! = 0)
                b[j] ← a[i] ;
                j← j + 1
        End if
End for
```

Step 4: [Output the prime numbers between 2 to n]

```
for i←0 to j– 1
        Write b[i]
End for
```

Step 5: [Finished]Exit

Units for measuring Running time

The running time of algorithms is often measured using different units depending on the context and the granularity of analysis. Here are common units for measuring running time:

Seconds (s):

- The most straightforward unit, representing the actual time in seconds that an algorithm takes to run on a specific machine.
- Suitable for measuring real-world performance but can be influenced by external factors like the machine's load.

Milliseconds (ms):

- A smaller unit than seconds, measuring time in thousandths of a second.
- Useful for more precise measurement when algorithms have relatively fast execution times.

Microseconds (µs):

- An even smaller unit than milliseconds, measuring time in millionths of a second.
- Appropriate for very fast algorithms or when extremely precise timing is required.

Nanoseconds (ns):

- A unit smaller than microseconds, measuring time in billionths of a second.

- Common in the context of measuring operations at the hardware level or very low-level algorithmic optimizations.

Operations or Basic Steps:

- Instead of measuring time, algorithms can be analyzed based on the number of basic steps or operations they perform.
- This unit provides an abstract measure of algorithmic complexity, independent of the specific hardware or software environment.

Big O Notation (O()):

- Represents the upper bound of an algorithm's time complexity in terms of a mathematical function, typically based on the input size.
- Provides a theoretical measure of efficiency, ignoring constant factors and lower-order terms.

Instruction Count:

- The number of machine instructions executed by an algorithm.
- Useful for low-level analysis and optimization, considering the specific machine architecture.

Comparisons or Swaps:

- For sorting algorithms, measuring the number of element comparisons or swaps provides insight into their efficiency.
- Relevant for algorithms where the primary operations involve comparisons or data rearrangement.

Analysis of algorithms

The main purpose of algorithm analysis is to design most efficient algorithms. Let use "On what factors efficiency of algorithm depends?" The efficiency of analgorithm depends on two factors:

1. Space efficiency
2. Time efficiency

"**What is space efficiency?"**

Definition: The space efficiency of an algorithm is the amount of memory required to run the program completely and efficiently. If the efficiency is measured with respect to the space (memory required), the word space complexity is often used. The space complexity of an algorithm depends on following factors:

Components that affect space

1. Program space
2. Data space

3. Stack space

- **Program space:** The space required for storing the machine program generated by the compiler or assembler is called program space.
- **Data space**: The space required to store the constants, variables etc., is called data
- **Stack space:** The space required to store the return address along with parameters that are passed to the function, local variables etc., is called stack space.

Note: The new technological innovations have improved the computer's speed and memory size by many orders of magnitude. Now a days, space requirement for an algorithm is not a concern and hence, we are not concentrating on space efficiency. Let us concentrate only on time efficiency..

"What is time efficiency?"

Definition: The time efficiency of an algorithm is measured purely on how fast a given algorithm is executed. Since the efficiency of an algorithm is measured using time, the word time complexity is often associated with an algorithm.

The time efficiency of the algorithm depends on various factors that are shown below:

Components that affect time efficiency

- Speed of the computer
- Choice of the programming language
- Compiler used
- Choice of the algorithm
- Number (Size) of inputs/Outputs
- Since we do not have any control over speed of the computer, programming language and compiler, let us concentrate only on next two factors such as:n
    - Choice of an algorithm
    - Number (size) of inputs

"What is basic operation?"

Definition: The operation that contributes most towards the running time of the algorithm is called basic operation. A statement that executes maximum number of times in a function is also called basic operation. The number of times basic operation is executed depends on size of the input. The basic operation is the most time consuming operation in the algorithm. For example:

- a statement present in the innermost loop in the algorithm
- addition operation while adding two matrices, since it is present in innermost loop
- multiplication operation in matrix multiplication since it is present in innermost loop

Now, let us see "How to compute the running time of an algorithm using basic operation or How time efficiency is analyzed?" The time efficiency is analyzed by determining the number of times the basic operation is executed. The running time T(n) is given by:

$T(n) \sim b*C(n)$

- T is the running time of the algorithm
- n is the size of the input
- b execution time for basic operation.
- C represent number of times the basic operation is executed

Order of growth

"What is order of growth? For what values of n we find the order of growth?"

Definition: We expect the algorithms to work faster for all values of n. Some algorithms execute faster for smaller values of n. But, as the value of n increases, they tend to be very slow. So, the behaviour of some algorithm changes with increase in value of n. This change in behaviour of the algorithm and algorithm's efficiency can be analyzed byconsidering the highest order of n. The order of growth is normally determined for largervalues of n for the following reasons:

- The behavior of algorithm changes as the value of n increases
- In real time applications we normally encounter large values of n

For example, the order of growth with respect to two running times is shown below:

- Suppose T(n) approx c* C(n) . Observe that T(n) varies linearly with increase or decrease in the value of n. In this case, the order of growth is linear.
- Suppose T(n) approx c* C(n ^ 2) . In this context, the order of growth is quadratic.

Note: If the order of growth of one algorithm is linear and the order of growth of second algorithm to solve the same problem is quadratic, then it clearly indicates that running time of first algorithm is less and it is more efficient. So, while analyzing the time efficiency of an algorithm, the order of growth of n is important. Let us discuss orders of growth in the next section.

The concept of order of growth can be clearly understood by considering the common computing time functions shown in table 1.1.

N	log N	N	N log N	N^2	N^3	2^N	N!
1	0	1	0	1	1	2	1
2	1	2	2	4	8	4	2
4	2	4	8	16	64	16	24
8	3	8	24	64	512	256	40320
16	4	16	64	256	4096	65536	high
32	5	32	160	1024	32768	4294967296	very high

Fig 1.1 Values of some of the functions

Note: By comparing N (which is linear) and 2"(which is exponential) it is observed from the above table that exponential function grows very fast even for small variation of N. So, an algorithm with linear running time is preferred over an algorithm with exponential running time. The basic efficiency of asymptotic classes are shown below:

1 or any constant: Indicates that running time of a program is constant.

log N: Indicates that running time of a program is logarithmic. This running time occurs in programs that solve larger problems by reducing the problem size by a constant factor at each iteration of the loop (For example, binary search).

N: Indicates that running time of a program is linear. So, when N is 1000, the running time is 1000 units. When N is doubled, so does the running time. (For example linear search)

N log N: Indicates that running time of a program is N log N (For lack adjective, it is used as it is instead of linear, quadratic etc). The algorithm elements in ascending order such as quick sort, merge sort and heap sort wave this running time. (These sorting techniques are discussed in later chapters)

N^2 : Indicates that running time of a program is quadratic. The algorithms normallywill have two loops. For example, sorting algorithms such as bubble sort, selectionsort, addition and subtraction of two matrices have this running time.

N^3 : Indicates that running time of a program is cubic. The algorithms with runningtime will have three loops. For example, matrix multiplication, algorithm to solvesimultaneous equations using gauss-elimination method will have this running time.

2^N: Indicates that running time of an algorithm is exponential. The tower of Hanoiproblem and algorithms that generate subsets of a given set will have this runningtime

N! : Indicates that running time of an algorithm is factorial. The algongenerate all permutations of set will have this running time.

Note: All the above functions can be ordered according to their order of growth (fromlowest to highest) as shown below:

$$1 < \log(n) < n < n * \log(n) < n^2 < n^3 < 2^n < n!$$

Note: Even though running times of 2^N and N! are different, normally both classes are considered as exponential. The algorithms with this running time can be solved practically for smaller values of n.

Note: For a very large value of n, the exponential function 2" and factorial function n!generate a very high value such that even the fastest computer can take years to executean algorithm.

For example, a computer which executes 10^{12} instructions per second takes $4*10^{10}$ years to execute 2^{100} operations. The factorial function n! is much more than the value specified for 2^n.

Worst-case, Best-case and average case efficiencies

Now the question is "Will the algorithm efficiency depends on the size of algorithm's input alone?" For some of the problems, the time complexity will not depend on the number of inputs alone. For example, while searching for a specific item in an array of n elements using linear search, we have following three situations:

- An item we are searching for may be present in the very first location itself. In this case only one item is compared and this is the ***best case.***
- The item may be present somewhere in the middle which definitely takes some time. Running time is more when compared to the previous case for the same value of n. Since we do not know where the item is we have to consider the average number of cases and hence this situation is an ***average case.***
- The item we are searching for may not be present in the array requiring n number of comparisons and running time is more than the previous two cases. This may be considered as the ***worst case***.

So, knowing n alone itself is not enough to estimate the run time of an algorithm or a function. In such cases, we may have to find the worst-case efficiency, best case efficiency and average case efficiency. Now, let us see "What is worst case efficiency?"

"What is worst case efficiency?"

Definition: The efficiency of an algorithm for the input of size n for which the algorithm takes longest time to execute among all possible inputs is called worst case efficiency For example, if we use linear search and the item to be searched is no present in the array, then it is an example of ***worst case efficiency***.

In the worst case, the algorithm runs for the longest duration among all possible inputs for a given size. Here, maximum number of steps are executed for that input.

"What is best case efficiency?"

Definition: The efficiency of an algorithm for the input of size n for which the algorithm takes least time during execution among all possible inputs of that size is called best case efficiency.

In the best case, the algorithm runs fastest for the input specified. For example, in linear search, if the item to be searched is present in the beginning, then it is an example of the ***best case efficiency.***

Note: In the average case efficiency, the average number of basic operations executed will be considered. This is required only for the randomized input.

Example: Searching for an Element in an Array

Suppose you have an array of numbers, and you want to find a specific element in that array.

```
def search_element(arr, target):
    for num in arr:
        if num == target:
            return True
    return False
```

Now, let's analyze the efficiency of this algorithm in terms of worst-case, best-case, and average-case scenarios.

Worst-case efficiency:

- This is the scenario where the algorithm takes the maximum amount of time to complete.
- In our example, the worst-case scenario occurs when the target element is at the end of the array or is not present in the array.
- The worst-case time complexity is O(n), where n is the number of elements in the # Worst-case scenario
  arr = [1, 2, 3, 4, 5]
  target = 5
  result = search_element(arr, target)
  # In this case, the algorithm has to go through the entire array to find the target element.

Best-case efficiency:

- This is the scenario where the algorithm takes the minimum amount of time to complete.
- In our example, the best-case scenario occurs when the target element is at the beginning of the array.
- The best-case time complexity is O(1), as the algorithm may find the target element in the first iteration.
  # Best-case scenario
  arr = [1, 2, 3, 4, 5]
  target = 1
  result = search_element(arr, target)

In this case, the algorithm finds the target element in the first iteration.

Average-case efficiency:

- This is the scenario where the algorithm takes an average amount of time to complete, considering all possible inputs.
- The average-case time complexity is often more challenging to analyze and may involve statistical analysis.
- In our example, if the target element is equally likely to be anywhere in the array, the average-case time complexity is O(n/2), which simplifies to O(n).

```
# Average-case
 scenarioarr = [1, 2, 3, 4, 5]
target = 3
result = search_element(arr, target)
# On average, the algorithm might need to check half of the array element
```

UNIT 2

Asymptotic Notations and Basic Efficiency classes, Informal Introduction, O-notation, Ω-notation, θ-notation, mathematical analysis of non-recursive algorithms, mathematical analysis of recursive algorithms.

Asymptotic notations

The efficiency of the algorithm is normally expressed using asymptotic notations. The order of growth can be expressed using two methods:

- Order of growth using asymptotic notations
- Order of growth using limits.

Before proceeding further, let us see "What do you mean by asymptotic behavior of a function?"

Definition: The value of the function may increase or decrease as the value of n increases. Based on the order of growth of n, the behavior of the function varies. Asymptotic notations are the notations using which two algorithms can be compared with respect to efficiency based on the order of growth of an algorithm's basic operation.

Now, let us see "What are the different types of asymptotic notations?" The different types of asymptotic notations are shown below:

Asymptotic notations

- O(Big Oh)
- Ω(Big Omega)
- ΘBigTheta

Now, let us see the informal definition and formal definition of above asymptotic notations

Informal Definitions of Asymptotic notations

In this section, let us "Give informal definitions of asymptotic notations"

Informal Definitions of Big-Oh(0)

Definition: Assuming n indicates the size of input and g(n) is a function, informally O(g(n)) is defined as set of functions with a small or same order of growth as g(n)) as n goes to infinity.

Ex 1: Let g(n) =n. Since n and I have smaller order of growth and n' has same order of growth when compared to n^2, we say.

Formal Definitions of Asymptotic notations

O (Big-Oh)

Now, let us see "What is Big Oh (0) notation?"

Definition: Let f(n) be the time efficiency of an algorithm. The function f(n) is said to be O(g(n)) [read as big-oh of g(n)), denoted by

$$f(n) = O(g(n))$$

or

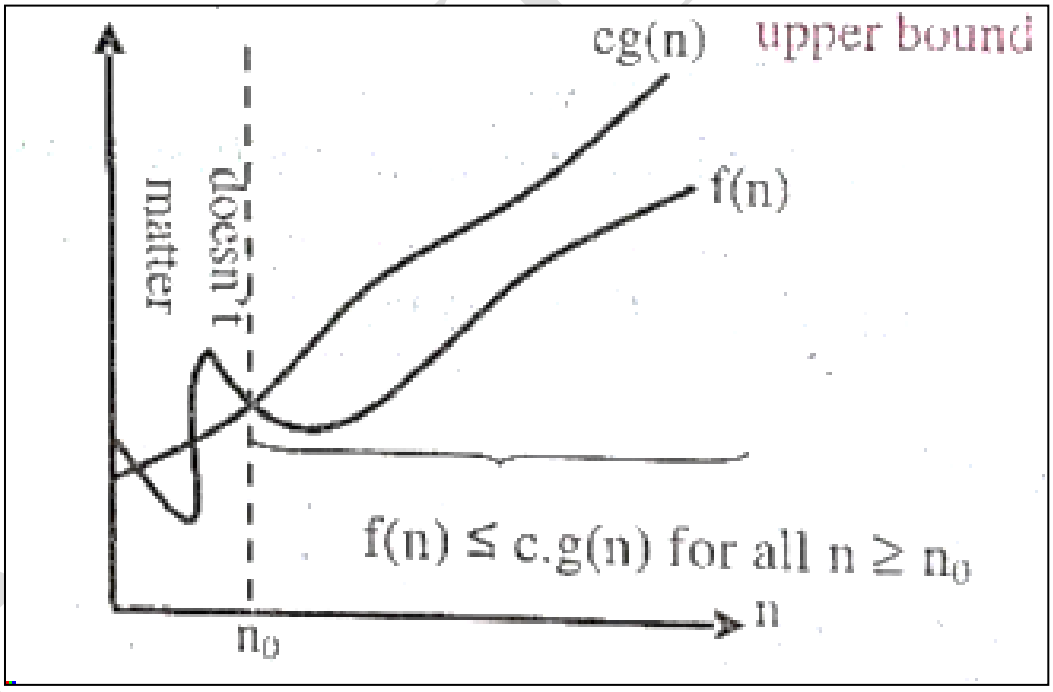
$$f(n) = O(g(n))$$

If and only if there exists a positive constant c and positive integer $n_0$satisfying the constraint

$f(n) <= c*g(n)$ for all $n \geq n_0$

So, if we draw the graph f(n) and c*g(n) verses n, the graph of the function f(n) lies

Below the graph of c g(n) for sufficiently large value of n as shown below:



Here, c .g(n) is the upper bound. The upper bound on f(n) indicates that function f(n) will not consume more than the specified time c* g(n) i.e., running time of function f(n) maybe equal to c .g(n), but it will never be worse than the upper bound. So, we can say that f(n) is generally faster than g(n)

Note: Big-O is the formal method of expressing the upper bound of an algorithm's running time. It is a measure of the longest amount of time it could possibly take for theAlgorithm to complete.

Note: The function g(n) is normally expressed using higher order terms of f(n) This is achieved using the following steps:

- Take the lower order term of f(n) replace the constant with next higher order variable. Repeat this step, till we get the higher order term and call it as c .g(n)
- Once the constraint " f(n) <= c.g(n) for n >= $n_0$is obtained we say f(n) ∈ O(g(n))

Example: Let f(n) = 100n + 5 Express f(n) using big-oh

Solution: It is given that f(n) = 100n + 5 Replacing 5 with n (so that next higher order term is obtained), we get 100n + n and call it c.g(n)

i.e..c. g(n) = 100n + n for n = 5

= 101nforn =5;

Now, the following constrain is satisfied:

f(n) <=c * g(n) for n ≥ n_0

I .e.,100n+5<=101 nfor n >= 5

It is clear from the above relations that c = 101 g(n) = n and n_0=5. So , by definition

f(n) ∈ O(g(n)) * I .e.., f(n) ∈ O(n)

Ω(Big-Omega)

Definition: Let f(n) be the time complexity of an algorithm. The function f(n) is said tobe Ω(g(n)) [read as big-omega of g(n)] which is denoted by

$$f(n) = \Omega g((n))$$

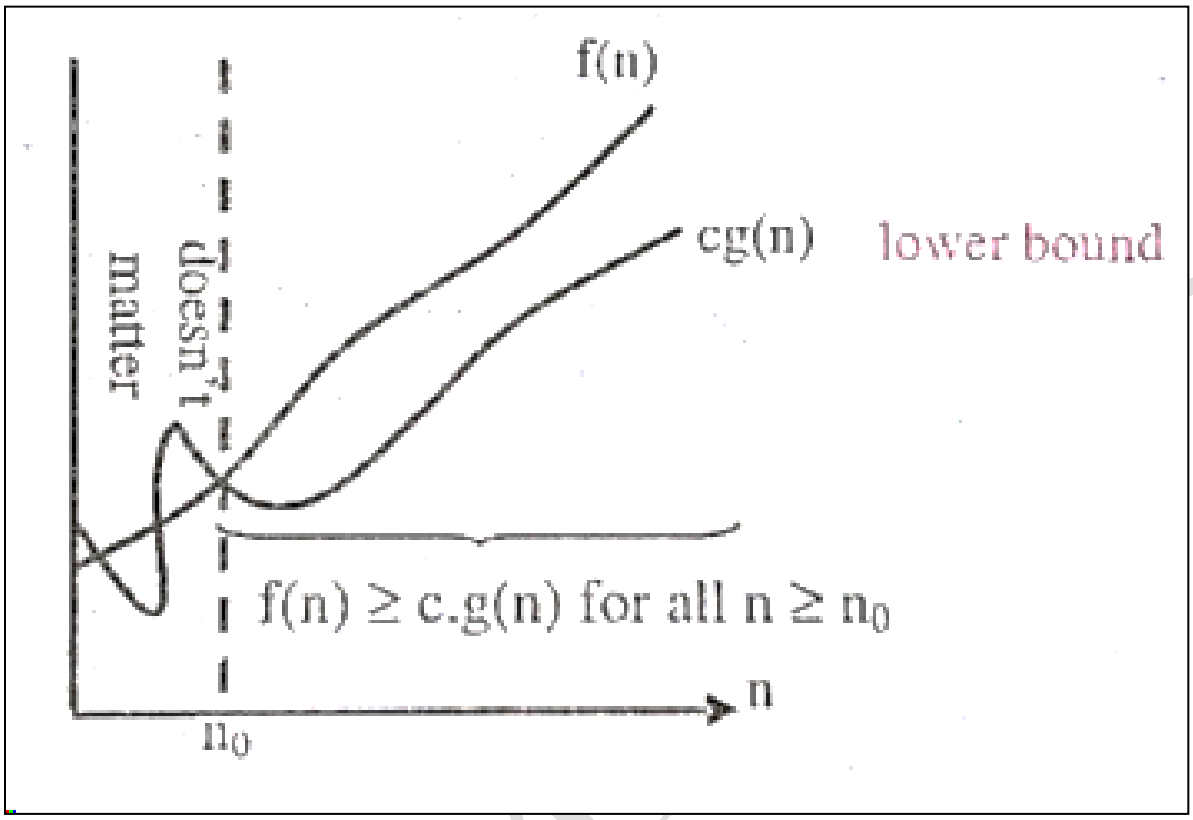
or

$$f(n) \text{ or } = \Omega(a(n))$$

If and only if there exists a positive constant c and non-negative integer no satisfying the constraint

f(n)>=c*g(n) for all n≥ n_0.

So, if we draw the graph f(n) and c*g(n) verses n, the graph of f(n) lies above the graph of g(n) for sufficiently large value of n as shown below:



This notation gives the lower bound on a function f(n) within a constant factor. The lower bound on f(n) indicates that function f(n) will consume at least the specified time c*g(n)i.e., the algorithm has a running time that is always greater than c*g(n). In general, the lower bound implies that below this time the algorithm cannot perform better.

Note: f(n) ≥ c*g(n) indicates that g(n) is a lower bound and the running time of an algorithm is always greater than g(n). So, big-omega notation is used for finding best case time efficiency.

Example: Let f(n) = 100n+ 5. Express f(n) using big-omega

Solution: The constrain to be satisfied is

f(n) >=c* g(n) for n >= n_0

i.e., 100n+5 ≥ 100 n for n>=0

It is clear from the above relations that c = 100. g(n) = n and n_0 = 0. So, by definition

f(n) = Ω(g(n))i.e.,f(n) ∈ Ω(n)

Example 1.21: Let $f(n) = 10n^3+5$. Express f(n) using Big-omega.

Solution: The constrain to be satisfied is

$$f(n) \geq c * g(n) \quad \text{for } n \geq n_0$$

$$\text{i.e, } 10n^3 + 5 \geq 10 * n^3 \quad \text{for } n \geq 0$$

It is clear from the above relations that $c = 10$, $g(n) = n^3$ and $n_0 = 0$. By definition,

$f(n) \in \Omega(g(n))$ i.e., $\boxed{f(n) \in \Omega(n^3)}$

Example 1.22: Let $f(n) = 6*2^n + n^2$. Express f(n) using Big-Omega.

Solution: The constrain to be satisfied is

$$f(n) \geq c * g(n) \quad \text{for } n \geq n_0$$

$$\text{i.e, } 6*2^n + n^2 \geq 6 * 2^n \quad \text{for } n \geq 0$$

It is clear from the above relations that $c = 6$, $g(n) = 2^n$ and $n_0 = 0$. By definition,

$f(n) \in \Omega(g(n))$ i.e., $\boxed{f(n) \in \Omega(2^n)}$

Big-Theta

Definition: Let f(n) be the time complexity of an algorithm. The function f(n) is said to be big-theta of g(n), denoted

$$f(n) = \Theta(g(n))$$

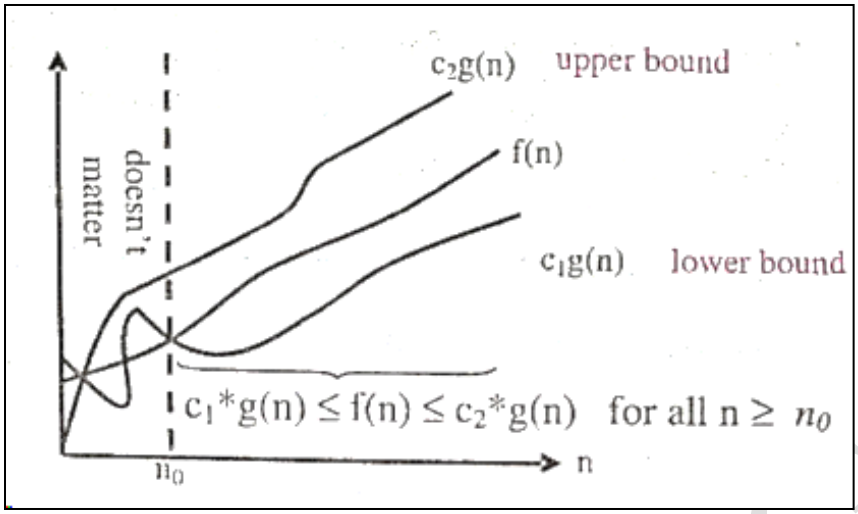
or

$$f(n) = \Theta(g(n))$$

if and only if there exists some positive constants c_1, c_2 and non-negative integer n_o satisfying the constraint

$c_1 * g(n) <= f(n) <= c_2 * g(n)$ for all $n \geq n_0$.

So, if we draw the graph f(n), c_1,*g(n) and c_2 g*(n) verses n, the graph of function f(n) lies above the graph of c_1*g(n) and lies below the graph of c_2* g(n) for sufficiently large value of n as shown below:



This notation is used to denote both lower bound and upper bound on a function f(n) within a constant factor. The upper bound on f(n) indicates that function f(n) will not consume more than the specified time cog(n).The lower bound on f(n) indicates that function f(n) in the best case will consume at least the specified time $c_1*g(n)$.

Example 1.23: Let f(n) = 100n + 5. Express f(n) using big-theta

Solution: The constraint to be satisfied is

$$c_1*g(n) \le f(n) \le c_2 * g(n) \quad \text{for } n \ge n_0$$

$$100*n \le 100n + 5 \le 105 * n \quad \text{for } n \ge 1$$

It is clear from the above relations that $c_1 = 100$, $c_2 = 105$, $n_0 = 1$, $g(n) = n$.
So, by definition

$$f(n) \in \theta(g(n)) \text{ i.e., } \boxed{f(n) \in \theta(n)}$$

Example 1.44: Simplify: $\sum_{i=0}^{n-1} i(i+1)$

Given: $\sum_{i=0}^{n-1} i(i+1) = \sum_{i=0}^{n-1} (i^2 + i)$

$= \sum_{i=0}^{n-1} i^2 + \sum_{i=0}^{n-1} i = \sum_{i=1}^{n-1} i^2 + \sum_{i=1}^{n-1} i$

of the form $\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \ldots\ldots n^2 = \frac{n(n+1)(2n+1)}{6}$

of the form $\sum_{i=1}^{n} i = 1 + 2 + \ldots\ldots n = \frac{n(n+1)}{2}$

$= \frac{(n-1)n(2n-1)}{6} + \frac{(n-1)n}{2}$

Taking $\frac{(n-1)n}{2}$ outside we get

$$\sum_{i=0}^{n-1} i(i+1) = \frac{(n-1)n}{2}\left[\frac{2n-1}{3} + 1\right] = \frac{(n-1)n}{2}\left[\frac{2n-1+3}{3}\right]$$

$$= \frac{(n-1)n}{2}\left[\frac{2n+2}{3}\right] = \frac{(n-1)n(n+1)}{3}$$

So, $\boxed{\sum_{i=0}^{n-1} i(i+1) = \frac{(n-1)n(n+1)}{3}}$

Mathematical analysis of non - recursive algorithms

The general plan of energy non recursive algorithm is shown below:

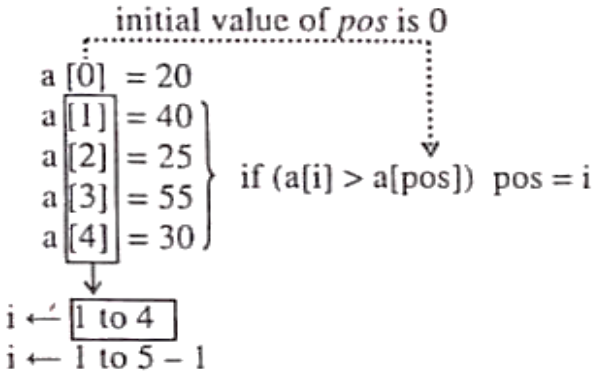
- Based on the size of input data in the number of parameters to be considered.
- Identify the basic operation in the algorithm
- Check whether the number of times the basic operation is executed depends only on the size of the input, if the basic operation to be executed depends on the some other conditions, then it is necessary to obtain the worst case, best case and average case separately.
- Open the total number of times a basic operation is executed
- Simplify using standard formula and obtain the order of growth.

Now let us consider some of the algorithms and see how to analyze these algorithms.

1.5.1 Maximum of n elements

Let us "Design the algorithm to find largest of *n* numbers and obtain the time efficiency"

Design: Consider the array *a* consisting of the 5 elements 20, 40, 25, 55 and 30. Here *n* = 5 represent the number of elements in the array. So, the parameters are *a* and *n*. The pictorial representation is shown below:



In general,

$i \leftarrow 1$ to $n - 1$ where *n* is the number of elements in the array

Now, the complete code can be written as shown below:

```
pos ← 0
for i ←1 to n – 1
        if (a[i] > a[pos])  pos ← i
end for
```

Now, the complete algorithm to find the largest of N elements can be written as shown below:

Example 1.45: Algorithm to find maximum of *n* elements

ALGORITHM Maximum(a[], n)
//Purpose : Find the largest of *n* numbers

```
//Inputs    : n – the number of items present in the table
              a – the table consisting of n elements
//Output    : pos – contains the position of largest element

pos ← 0                                  // 0 is assumed to be the position of largest
                                         // element

for i ← 1 to n-1 do                      // Find the position of largest element in the
      if ( a[i] > a[pos]) ) pos ← i      // remaining elements of array from 1 to n-1
end for

return pos                               // return the position of largest element
```

End of algorithm Largest

Analysis: The time efficiency can be calculated as shown below:

Step 1: The parameter to be considered is *n* which represent the size of the input

Step 2: The element comparison i.e., **"if (a[i] > a[pos])"** is the basic operation

Step 3: The total number of times the basic operation is executed can be calculated as shown below:

```
for i ← 1 to n-1 do
    if ( a[i] > a[pos] ) pos ← i
        ......
```

$$f(n) = \sum_{i=1}^{n-1} 1$$ **Note:** Upper bound = n-1, Lower bound = 1

$= (n-1)-1+1$ // Result = [Upper bound – lower bound +1]

$= n-1$

i.e., **$f(n) = n-1 \approx n$** **// By neglecting lower order terms and constants**

Step 4: Express f(n) using asymptotic notation. So, time complexity is given by:

$f(n) \in O(n)$

Mathematical analysis of recursive algorithms

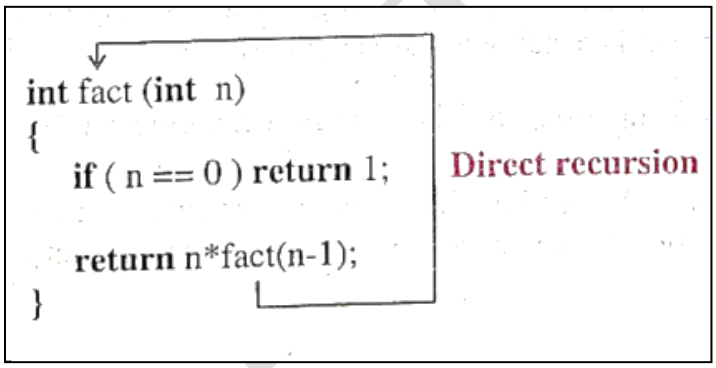
Recursion is a powerful tool but least understood by most novice students. Programming languages such as Pascal, C, C++ etc support recursion. Now, let us see "What is recursion? What are the various types of recursion?"

Definition: A recursion is a method of solving the problem where the solution to a problem depends on solutions to smaller instances of the same problem. Thus, a recursive function is a function that calls itself during execution. This enables the function to repeat itself several times to solve a given problem.

The various types of recursion are shown below:

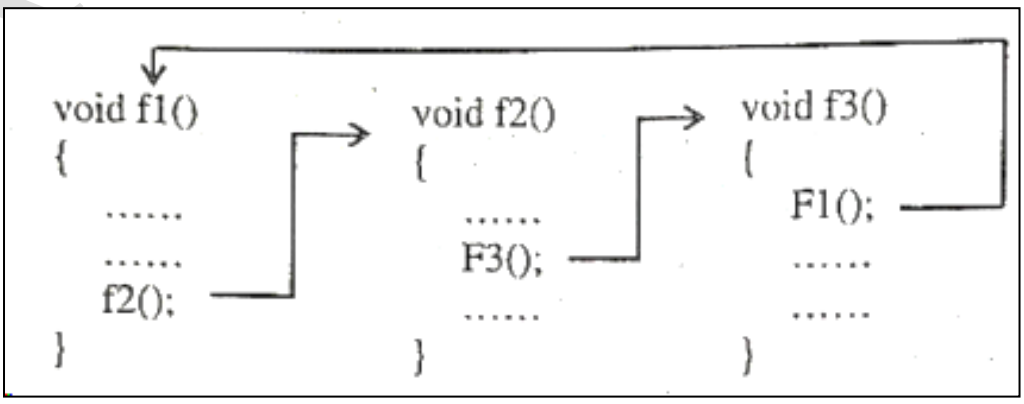
1. Direct recursion
2. Indirect recursion

Direct recursion: A recursive function that invokes itself is said to have direct recursion. For example, the factorial function calls itself (detailed explanation is given later) and hence the function is said to have direct recursion.



Indirect recursion: A function which contains a call to another function which in turn calls another function which in turn calls another function and so on and eventually calls the first function is called indirect recursion.

It is very difficult to read, understand and find any logical errors in a function that has indirect recursion. For example, a function fl invokes f2 which in turn invokes f3 which in turn invokes fl is said to have indirect recursion. This is pictorially represented as shown below:



Now, the question is ***"How to design recursive functions?"*** Every recursive call must solve one part of the problem using base case or reduce the size (or instance) of the problem using general case. Now, let us see ***"What is base case? What is a general case?"***

Definition: A ***base case*** is a special case where solution can be obtained without using recursion. This is also called base/terminal condition. Each recursive function must have a base case. A base case serves two purposes:

- It acts as terminating condition.
- The recursive function obtains the solution from the base case it reaches.

For example, in the function factorial O! is I is the base case or terminal condition.

Definition: In any recursive function, the part of the function except base case is called general case. This portion of the code contains the logic required to reduce the size (or instance) of the problem so as to move towards the base case or terminal condition. Here, each time the function is called, the size (or instance) of the problem is reduced.

For example, in the function fact, n*fact(n-1) is general case. By decreasing the value of n by 1, the function fact is heading towards the base case.

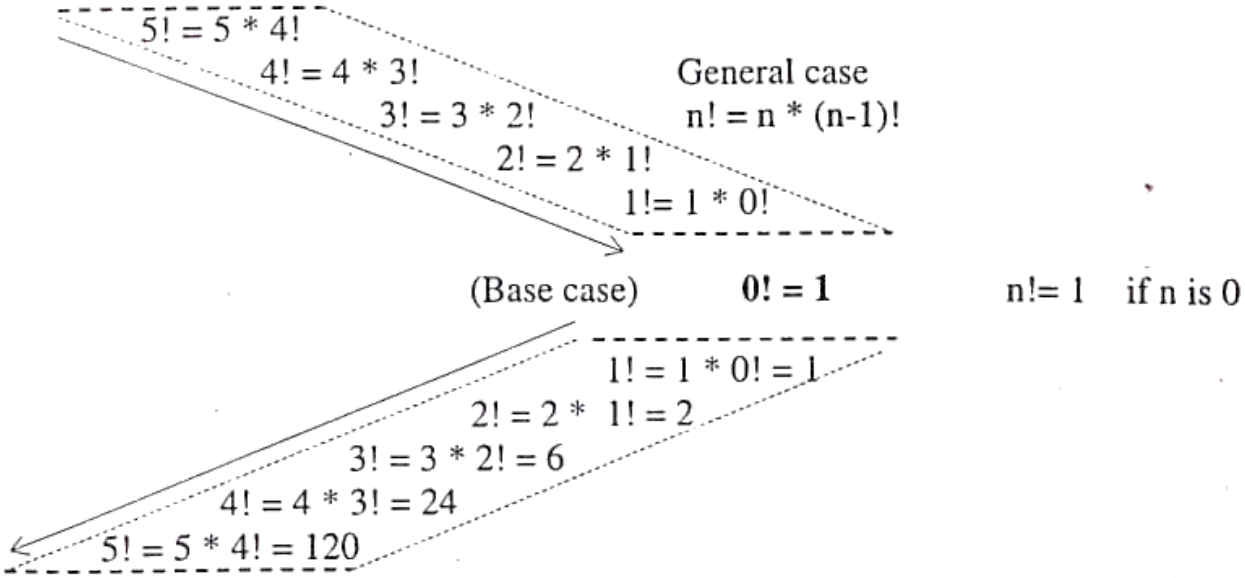
So, the general rules that we are supposed to follow while designing any recursive algorithm are:

- **Determine the base case**. Careful attention should be given here, because: when base case is reached, the function must execute a return statement without a call to recursive function.
- **Determine the general case.** Here also careful attention should be given and see that each call must reduce the size of the problem and moves towards base case.
- **Combine the base case and general case** into a function.

Note: A recursive function should never generate infinite sequence of calls on itself. An algorithm exhibiting this sequence of calls will never terminate and hence it is called infinite recursion. If a base case does not exist, no recursive function can ever be computed.

1.6.1 Factorial of a number

Now, let us see "How to compute factorial of 5 using recursion?" We can compute 5! as shown below:



Thus the recursive definition can be written as shown below:

n!= 1	if n == 0
n! = n * (n-1)!	otherwise

or

$$n! = \begin{cases} 1 & \text{if } n == 0 \\ n * (n-1)! & \text{otherwise} \end{cases}$$

The above definition can also be written as shown below:

$$F(n) = \begin{cases} 1 & \text{if } n == 0 \\ n * F(n-1) & \text{otherwise} \end{cases}$$

Using the above recursive definition, the recursive algorithm can be written as shown below:

Example 1.53: Recursive algorithm to find the factorial of N

```
Algorithm fact(n)
// Purpose     : This function computes factorial of n
// Input       : n: represent a positive integer
//Output       : factorial of n

        if ( n == 0 )    return 1          // n! = 1 if n is 0
        return           n*fact(n-1)       // n! = n*(n-1)! otherwise
```

End of algorithm factorial

The C function for the above algorithm can be written as shown below:

Example 1.54: C function to find the factorial of N

```
int fact(int n)
{
        if ( n == 0 )    return 1;         /* factorial of n when n = 0 */
        return           n*fact(n-1);      /* factorial of n when n > 0 */
}
```

Now, let us see "What is the general plan to analyze the efficiency of recursive algorithms?" The general plan to analyze the recursive algorithms is shown below:

- Identify the parameters based on the size of the input
- Identify the basic operation in the algorithm
- Obtain the number of times the basic operation is executed on different inputs of the same size. If it varies, then it is necessary to obtain the worst case, best case and average case separately.
- Obtain a recurrence relation with an appropriate initial condition
- Solve the recurrence relation and obtain the order of growth and express using asymptotic notations.

Analysis The time efficiency of the algorithm to find the factorial of a number can be obtained as shown below:

Step 1: The parameter to be considered is n, which is a measure of input's size

Step 2: The basic operation is the multiplication statement.

Step 3: The total number of multiplications can be obtained using the recurrence relation as shown below:

$$t(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 + t(n-1) & \text{otherwise} \end{cases}$$

OR

$$t(0) = 0$$
$$t(n) = 1 + t(n-1) \quad \text{----------(1)}$$

The above recurrence relation can be solved using repeated substitution as shown below:

$t(n) = 1 + t(n-1)$

$= 1 + 1 + t(n-2)$

$= 2 + t(n-2)$

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

$= 3 + t(n-3)$

$= 4 + t(n-4)$

.....

$t(n) = 1 + t(n-1)$ ----------(2)

$t(n-1) = 1 + t(n-2)$ replacing n by n-1 in equation (2)

$t(n-2) = 1 + t(n-3)$ replacing n by n-2 in equation (2)

Note: Looking at the previous expression it is written

.....

.....

$= i + t(n-i)$

⋮ ⟶ Finally, to get initial condition t(0), let i = n

$= n + t(n-n)$

$= n + t(0)$

$= n + 0 = n$

Note: t(0) = 0 from equation (1)

So, the time complexity of factorial of N is given by $t(n) \in \theta(n)$

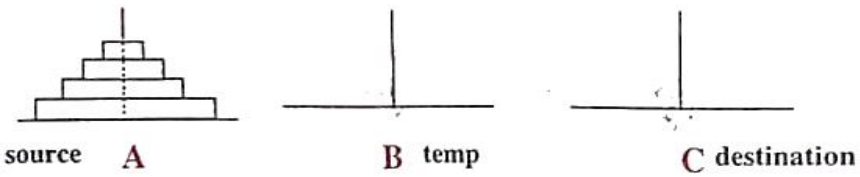
1.6.2 Tower of Hanoi

Let us see, "What is tower of Hanoi problem?" In this problem, there are **three** needles say **A**, **B** and **C**. The different diameters of **n** discs are placed one above the other through the needle **A** and the discs are placed such that always a smaller **disc is** placed above the larger disc. The two needles B and C are empty. All the discs **from** needle A are to be transferred to needle C using needle B as temporary storage.

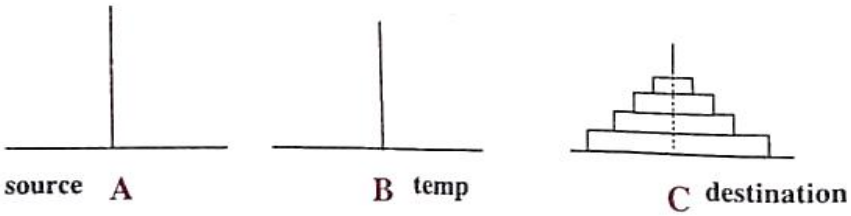
The rules to be followed while transferring the discs are

- Only one disc is moved at a time from one needle to another needle
- Smaller disc is on top of the larger disc at any time.
- Only one needle can be used to for storing intermediate discs

The initial set up of the problem is shown below.



After transferring all discs from **A** to **C** we get the following setup.



Base Case: Now, let us see "What is the base case?" This case occurs when there are no discs. In such situation we simply return the control to the calling function using the statement **return**. The base case can be written as:

Action	Condition
return	**if n = 0**

General case: Now, let us see "What is the general case?" This case occurs if one or more discs have to be transferred from source to destination. If there are *n* discs, then all *n* discs can be transferred recursively using following three steps:

- Move **n-1** discs recursively from **source** to **temp**.
- Move **n**th disc from **source** to **destination**.
- Move **n-1** discs recursively from **temp** to **destination.**

The algorithm to implement tower of Hanoi problem is shown below:

Example 1.56: Algorithm for tower of Hanoi problem

```
Algorithm TowerOfHanoi(n, source, temp, destination)

//Purpose: To move n discs from source to destination and see that only one disc is
           moved and always the smaller disc should be placed above the larger disc
           using one of the needle as temporary holder for the disc being moved
// Inputs :
//        n: total number of disks to be moved

//Output:
//        all n disks should be available on the destination needle.

Step 1: [ Check for base case ]
        if ( n = 0 ) return

Step 2: [Recursively move n-1 disks from source to temp]
        TowerOfHanoi(n-1, source, destination, temp)

Step 3: [Move nth disk from source to destination]
        write("Move disk", n, "from", source, "to", destination)

Step 4: [Recursively move n-1 disks from temp to destination]
        TowerOfHanoi(n-1, temp, source, destination)
```

End of tower of Hanoi algorithm

Analysis The time efficiency can be calculated as shown below:

Step 1: The parameter to be considered is *n*, which represent number of disks

Step 2: The basic operation is the movement of disk

Step 3: The total number of disk movements can be obtained using the recurrence relation as shown below:

$$t(n) = \begin{cases} 1 & \text{if } n = 1 \\ t(n-1) + 1 + t(n-1) & \text{otherwise} \end{cases}$$

- $t(n-1)$: Number of disk movements from source to temp
- 1: Number of disk movements from source to destination
- $t(n-1)$: Number of disk movements from temp to destination

The above recurrence relation can also be written as shown below:

$$f(1) = 1 \quad \text{----------(1)}$$
$$f(n) = 2f(n-1) + 1$$

The above recurrence relation can be solved using repeated substitution as shown below:

$f(n) = f(n-1) + 1 + f(n-1)$

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

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

$f(n-1) = 2f(n-2) + 1$ by replacing n by n-1 in eq. (2)

$= 2\,[2f(n-2) + 1\,] + 1$

$f(n-2) = 2f(n-3) + 1$ by replacing n by n-2 in eq. (2)

$= 2^2 f(n-2) + 2 + 1$

$= 2^2\,[2f(n-3) + 1] + 2 + 1$

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

$= 2^4 f(n-4) + 2^3 + 2^2 + 2 + 1$ **Note:** Written by looking at the previous expression

.....

.....

In general,

$= 2^i f(n-i) + 2^{i-1} + 2^{i-2} \ldots\ldots 2^3 + 2^2 + 2 + 1$

$= 2^i f(n-i) + 2^{i-1} + 2^{i-2} \ldots\ldots 2^3 + 2^2 + 2^1 + 2^0$ ---------- (3)

This geometric series can be solved using $S = \frac{a(r^n - 1)}{r - 1}$

where a = 1, r = 2, n = i **(since number of terms from 0 to i-1 = i)**

So, $S = \frac{1(2^i - 1)}{2 - 1} = 2^i - 1$. Substituting this value in equation (3) we have,

$= 2^i f(n-i) + 2^i - 1$

⋮ ……> Finally, to get initial condition f(1), let i = n-1

$= 2^{n-1} f(n-[n-1]) + 2^{n-1} - 1$

$= 2^{n-1} f(n - n + 1) + 2^{n-1} - 1$

$= 2^{n-1} f(1) + 2^{n-1} - 1$

$= 2^{n-1} * 1 + 2^{n-1} - 1$ **Note:** f(1) = 1 from equation (1)

$= 2^{n-1} + 2^{n-1} - 1$

$= 2*2^{n-1} - 1 = 2* \frac{2^n}{2} - 1 = 2^n - 1$

So, $f(n) = 2^n - 1$

Step 4: Since number of disk movements in best case is same as worst case, we express f(n) using θ-notation as shown below:

So, the time complexity for Tower of Hanoi problem is given by $f(n) \in \theta(2^n - 1) \in \theta(2^n)$

UNIT-3

Brute Force & Exhaustive Search: Introduction to Brute Force approach, Selection Sort and Bubble Sort, Sequential search, Exhaustive Search- Travelling Salesman Problem and Knapsack Problem, Depth First Search, Breadth First Search

Introduction to Brute Force approach

There are various approaches to solve a given problem. In a straight forward approach and simple technique such as brute force, there is no emphasis on the efficiency of the algorithm. The concept used in this technique is "Just do it".

Definition:

The straight forward method of solving a given problem based on the problem's statement and definitions is called Brute Force technique. This method is often easier to implement than a more sophisticated one and, because of this simplicity. sometimes it can be more efficient.

For example, the algorithms to find GCD of 2 numbers (Consecutive integer checking method, Matrix multiplication, addition etc, Selection sort, Bubble sort, Linear search (sequential search), Brute-force string matching etc. can be solved using brute force technique.

The various advantages and disadvantages of this approach are shown below:

Advantages

- This method is applicable for wide variety of problems such as finding the sum of n numbers, Computing power, Computing GCD and so on
- Simple and easy algorithms can be written. For example, bubble sort, selection sort, matrix multiplication etc
- Can be used to judge more efficient alternative approaches to solve a problem

Disadvantages:

- Rarely yields efficient algorithms
- Some root force algorithms are unacceptably slow, for example bubble sort.
- Not as a constructive or creative as other design techniques such as divide and conquer.

Selection Sort

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

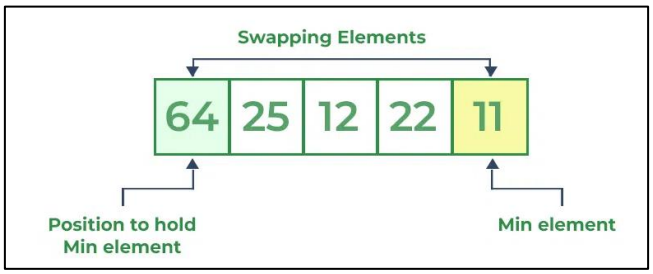
Procedure:

As the name indicates, we first find the smallest item in the list and we exchange it with the first item, obtain the second smallest in the list and exchange it with the second element, and so on. Finally, all the elements will be arranged in ascending order, since the next last item is selected and exchanged appropriately so that the elements are finally sorted. This technique is called ***selection sort***.

Let us see, How the elements {***64, 25, 12, 22,11***} can be sorted using Selection Sort.

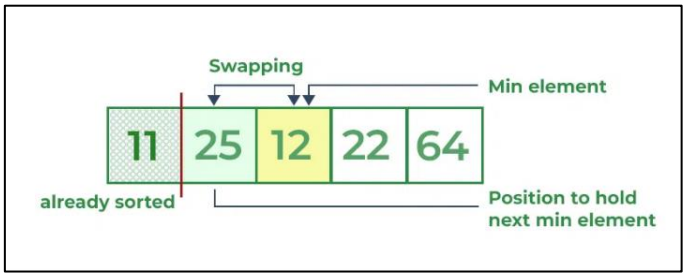
First pass:

- For the first position in the sorted array, the whole array is traversed from index 0 to 4 sequentially. The first position where 64 is stored presently, after traversing whole array it is clear that 11 is the lowest value.
- Thus, replace 64 with 11. After one iteration 11, which happens to be the least value in the array, tends to appear in the first position of the sorted list.



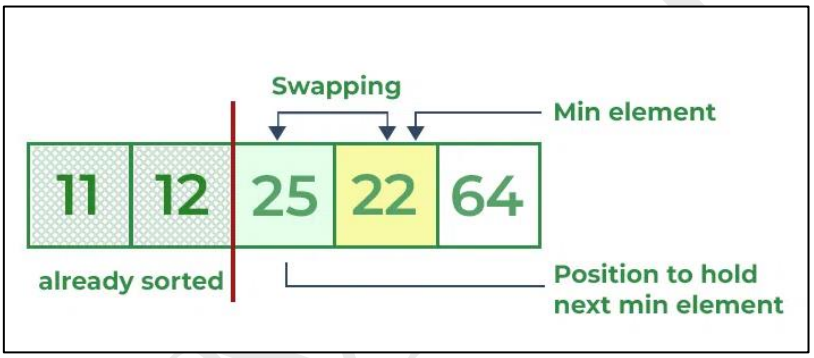
Second Pass:

- For the second position, where 25 is present, again traverse the rest of the array in a sequential manner.
- After traversing, we found that 12 is the second lowest value in the array and it should appear at the second place in the array, thus swap these values.



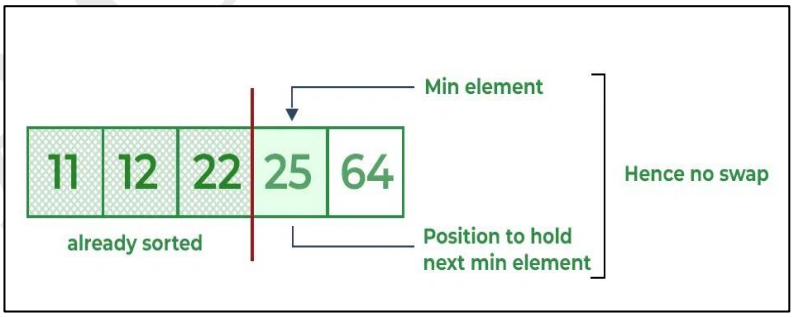
Third Pass:

- Now, for third place, where 25 is present again traverse the rest of the array and find the third least value present in the array.
- While traversing, 22 came out to be the third least value and it should appear at the third place in the array, thus swap 22 with element present at third position.



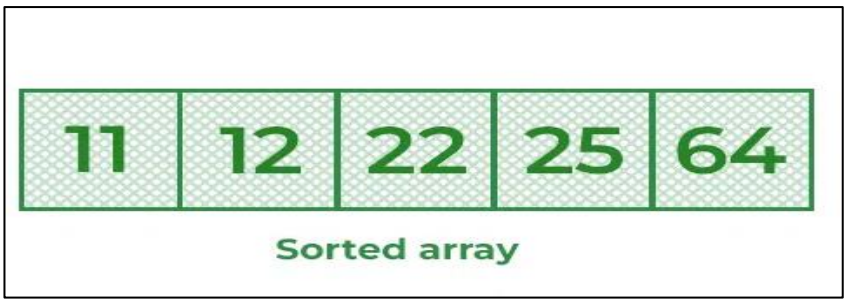
Fourth pass:

- Similarly, for fourth position traverse the rest of the array and find the fourth least element in the array
- As 25 is the 4th lowest value hence, it will place at the fourth position.



Fifth Pass:

- At last the largest value present in the array automatically get placed at the last position in the array
- The resulted array is the sorted array.



Algorithm for Selection Sort:

```
Algorithm SelectionSort(arr, n)
//Purpose: Sort the given elements using selection sort
//Inputs:
//  - n: the number of items present in the array
//  - arr: the items to be sorted are present in the array
//Output:
//  - arr: contains the sorted list

for i ← 0 to n-2 do
    pos = i                    //Assumes ith element as smallest
    for j ← i+1 to n-1 do          // Find the position of the
smallest item
        if (arr[j] < arr[pos])
        pos ← j
    end for
    temp = arr[pos]
    arr[pos] = arr[i]
    arr[i] = temp
end for
```

Complexity Analysis of Selection Sort

Input: Given n input elements.

Output: Number of steps incurred to sort a list.

Logic:

- If we are given n elements, then in the first pass, it will do n-1 comparisons;
- in the second pass, it will do n-2; in the third pass,
- it will do n-3 and so on. Thus, the total number of comparisons can be found by

Output;

(n-1) + (n-2) + (n-3) + (n-4) + ... + 1

$$Sum = \frac{n(n-1)}{2}$$

i.e., $O(n^2)$

Therefore, the selection sort algorithm encompasses a time complexity of ***O(n2)*** and a space complexity of ***O(1)*** because it necessitates some extra memory space for temp variable for swapping.

Time Complexities:

- **Best Case Complexity:** The selection sort algorithm has a best-case time complexity of ***O(n2)*** for the already sorted array.
- **Average Case Complexity:** The average-case time complexity for the selection sort algorithm is ***O(n2)***, in which the existing elements are in jumbled ordered, i.e., neither in the ascending order nor in the descending order.
- **Worst Case Complexity:** The worst-case time complexity is also ***O(n2),*** which occurs when we sort the descending order of an array into the ascending order.

In the selection sort algorithm, the time complexity is ***O(n2)*** in all three cases. This is because, in each step, we are required to find m

Bubble Sort

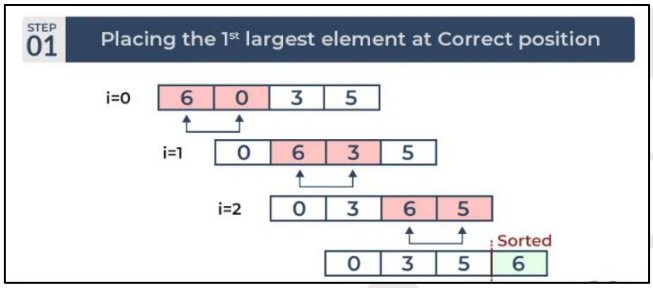
Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets.

Let us understand the working of bubble sort with the help of the following illustration:

Input: arr[] = {6, 3, 0, 5}

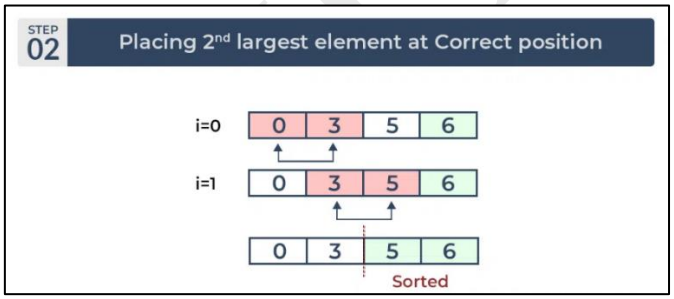
First Pass:

The largest element is placed in its correct position, i.e., the end of the array.



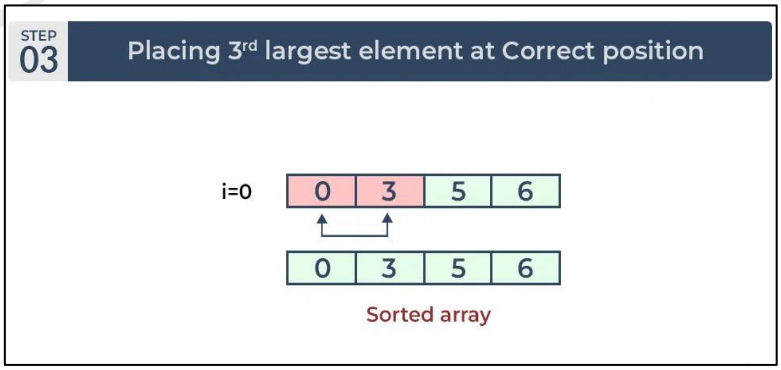
Second Pass:

Place the second largest element at correct position



Third Pass:

Place the remaining two elements at their correct positions.



Algorithm for Bubble Sort:

```
Algorithm BubbleSort(a[], n)
// Purpose: Arrange the numbers in ascending order
// Inputs:
//   n:the items to be sorted are present in the array
//   a: the number of items present in the array
// Output:
//   a: contains the sorted list

    for j ← 1 to n-1 do              // Perform n-1 passes
         for i ← 0 to n-j-1 do       // To compare items in each
pass
           if (a[i] > a[i+1]) then   // If out of order, exchange
adjacent
                temp = a[i]                elements
                a[i] = a[i+1]
                a[i+1] = temp
           end if
        end for
end for
```

Complexity Analysis of Bubble Sort

Input: Given a list of n elements.

Output: The number of steps required to sort the list.

Logic:

- During the first pass, Bubble Sort performs n-1 comparisons.
- In the second pass, it does n-2 comparisons,
- and in the third pass, it does n-3, continuing in this manner. Consequently, the total number of comparisons can be determined by:

Output;

(n-1) + (n-2) + (n-3) + (n-4) + ... + 1

$$Sum = \frac{n(n-1)}{2}$$

i.e., $O(n^2)$

Time Complexity:

- **Best Case:** Bubble sort has a best-case time complexity of O(n) when the array is already sorted.
- **Average Case:** The average-case time complexity is O(n^2), occurring when two or more elements are in a jumbled order (neither ascending nor descending).
- **Worst Case:** The worst-case time complexity is O(n^2), happening when sorting a descending order array into ascending order.

Space Complexity: The space complexity is O(1) as it requires some extra memory space for a temporary variable used in swapping.

Sequential search:

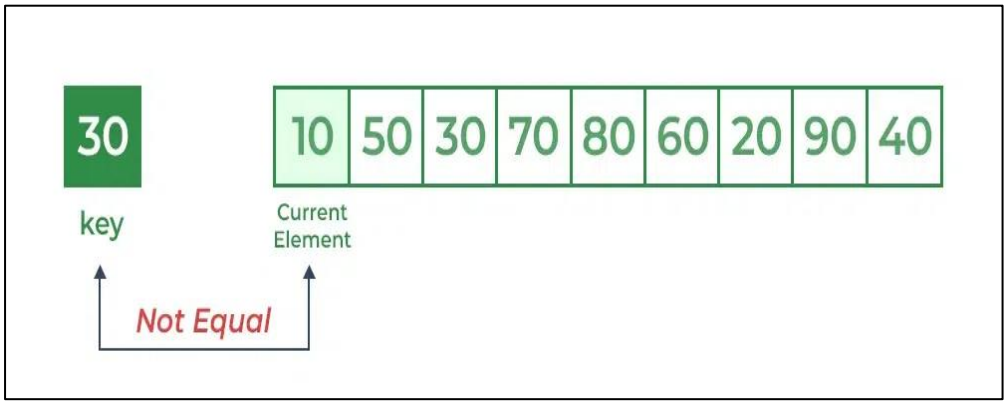
A linear search is a simple searching technique. In this technique we search for a given key item in the list of linear order, that is one after another. The item to be searched is often called key item. The *linear search* is also called as sequential search.

For example: Consider the list{10, 50, 30, 70, 80, 20, 90, 40} and key = 30 is present in the list,we say search is successful. If key is 100 and after searching we say that key is not present and hence search is unsuccessful.

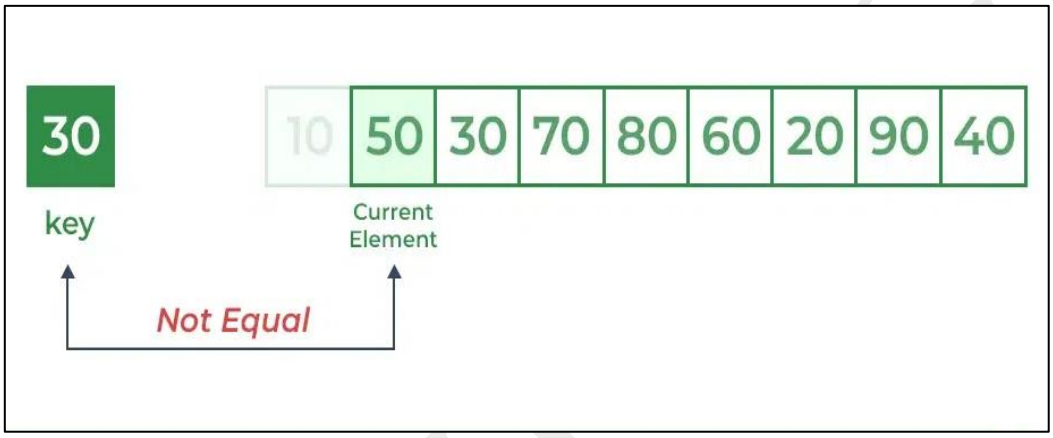
Working of Sequential Search:

Step 1: Start from the first element (index 0) and compare key with each element (arr[i]).

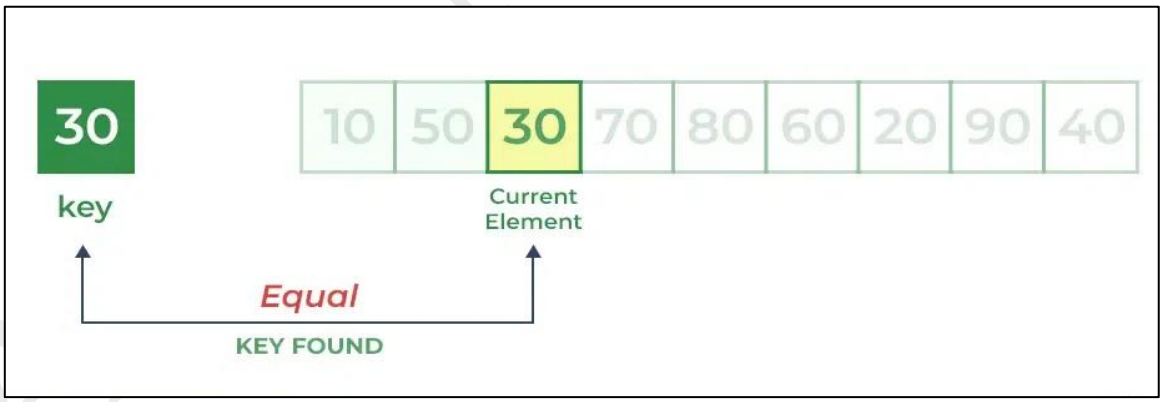
- Comparing key with first element arr[0]. SInce not equal, the iterator moves to the next element as a potential match.



- Comparing key with next element arr[1]. SInce not equal, the iterator moves to the next element as a potential match.



Step 2: Now when comparing arr[2] with key, the value matches. So the Linear Search Algorithm will yield a successful message and return the index of the element when key is found (here 2).



Algorithm:Sequential Search

```
Algorithm LinearSearch(key, a[], n):
// Purpose: This algorithm searches for the key in the array 'a'
which has 'n' elements.

// Inputs:
// - n: the number of items present in the array
// - a: the items in the array where searching takes place
// - key: the item to be searched

// Output:
//  The function returns the position if the key is found;
Otherwise, the function returns -1 indicating the search is
unsuccessful.

    a[n] = key             // Insert the search key at the end of
the list
    i = 0                  // Start searching from the beginning of
the array

    while (a[i] != key) do
        i = i + 1
    end while

    if (i < n) then
        return i           // Key found at position i
    else
        return -1          // Key not found
    end if
```

Sequential Search Complexity Analysis

Input: A list of n elements and a target element to search.

Output: The number of steps required to find the target element in the list.

Logic: Sequential Search checks each element in the list one by one until the target element is found or the entire list is traversed.

Total comparisons = 1 + 2 + 3 + ... + (n-1) + n

Time Complexity:

- **Best Case:** O(1) when the target is at the beginning of the list.
- **Average Case:** O(n), occurs when the target is randomly located in the list.
- **Worst Case**: O(n), when the target is at the end of the list or not present.
- **Space Complexity:** O(1), as it requires only a constant amount of extra memory for control variables.

Exhaustive Search

Traveling Sales Man Problem:

Definition:

Given n cities, a salesperson starts at a specified city (often called source), visit all n-1 cities only once and return to the city from where he has started.

The ***objective*** of this problem is to find a route through the cities that minimizes the cost there by maximizing the profit.

This problem can be modelled by a directed weighted graph as shown below:

- The vertices of the graphs represent the various cities
- The weights associated with edges represent the distances between two cities or the cost involved while traveling from one city to other city.
- The cost involved from city i to city j is represented using a 2-dimensional array and C[i, j] gives the cost from city i to city j.

TSP Using Brute force:

Using brute force approach the TSP can be solved as shown below:

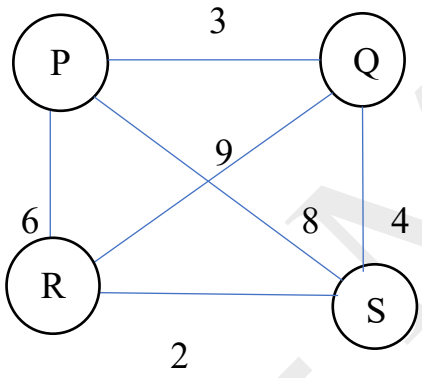
- Get all the routes from one city to another city by taking various permutations
- Compute the route length for each permutation and select the shortest among them.

But, in this technique, the execution time increases very rapidly as the size of the TSP problem increases. **For example,**

- If the number of cities = 16, the number of possible routes are 1,307,674,368,000
- If number of cities =20, the number of possible routes are 12,164,510,040,883,200.

Note: Thus, the easiest way to find an exact solution to the traveling salesman problem is to compute the cost of all tours and determine the tour with the minimum cost. The above method is called exhaustive sequential search algorithm and uses brute force approach.

Example:



- Where the cities are represented as P ,Q ,R & S and the distance between various cities are represented as number in each of the edge.
- The graph represented using cost adjacency matrix is also shown below

	P	Q	R	S
P	0	3	6	8
Q	3	0	9	4
R	6	9	0	2
S	8	4	2	0

Working of TSP Brute Force

1. Consider city P is the Starting and ending vertex

2. Generate all (n-1)! Permutations of cities
3. Calculate the cost of the every permutation and keep track on the minimum cost permutation.
4. Return the Permutation of the Minimum cost

Various Possible routs are:

p→q→r→s→p (3 + 9+ 2 + 8 = 22)

p→q→s→r→p (3 + 4 + 2 + 6 = 15)

p→r→q→s→p (6 + 9 + 4 + 8 = 27)

p→r→s→q→p (6 + 2 + 4 + 3 = 15)

p→s→q→r→p (8 + 4 + 9 + 6= 27)

p→s→r→q→p (8 + 2 + 9 + 3= 22)

- we have variety of routes with varying costs, we have to consider the route with minimum cost to get the maximum profit. So, the following routes can be selected by the salesperson:
  p→q→s→r→p (Cost = 15)
  p→r→s→q→p (Cost = 15)

Knapsack Problem Greedy Method

Definition:

Given a knapsack, bag or container of capacity m&n objectives of weights w_1,w_2,w_3…w_n with profit $p_1p_2p_32…p_n$, let $x_1x_2x_3$ …$x_n$be the fraction of the objects that are supposed to be added to the knapsack.

Objective:

The main objective is to place the objects into the knapsack so that the maximum profit is obtained and the weight of the object chosen should not exceed the capacity of knapsack.

Example:

Greedy Method

Fractional Knapsack problem

W = 15 n = 7

Objects	-	1	2	3	4	5	6	7
Profit (P)	-	5	10	15	7	8	9	4
Weight (w)	-	1	3	5	4	1	3	2
P/w	-	5	3.3	3	1.75	8	3	2

I Max Profit

Objects	Profit	Weight	Remaining Weight
3	15	5	15 - 5 = 10
2	10	3	10 - 3 = 7
6	9	3	7 - 3 = 4
5	8	1	4 - 1 = 3
4	7 x 3/4	3	3 - 3 = 0
	= 5.25		

= 47.25

Min weight

Objects	Profit	Weight	Remaining Weight
1	5	1	15 - 1 = 14
5	8	1	14 - 1 = 13
7	4	2	13 - 2 = 11
2	10	3	11 - 3 = 8
6	9	3	8 - 3 = 5
4	7	4	5 - 4 = 1
3	3 15x1/5	1	1 - 1 = 0
	46		

III Max P/W Ratio

Objects	Profit	Weight	Remaining Weight
5	8	1	15 - 1 = 14
1	5	1	14 - 1 = 13
2	10	3	13 - 3 = 10
3	15	5	10 - 5 = 5
6	9	3	5 - 3 = 2
7	4	2	2 - 2 = 0
	51		

Algorithm for Knapsack Greedy method:

```
Algorithm GreedyKnapsack(m, n, w, x)
// Purpose: To find the solution vector that shows the fraction of
the object selected in the knapsack problem
// Input: m - the capacity of the knapsack, n - the number of
objects, w - an array of weights of all n objects
// Output: x - solution vector containing the fraction of each
object selected

// Initialize the solution vector x
for i from 1 to n do
    x[i] = 0
end for

// Initialize the remaining capacity of the knapsack
rc = m

// Greedy selection of objects
for i from 1 to n do
    // If the weight of the current object is greater than the
remaining capacity, break
    if w[i] > rc then
        break
    end if

    // Select the entire object
    x[i] = 1

    // Update the remaining capacity of the knapsack
    rc = rc - w[i]

    // If there is more space, select a fraction of the object
    if i <= n then
        x[i] = rc / w[i]
    end if
end for
```

Knapsack 0/1 Problem:

What is 0/1 knapsack problem

The 0/1 knapsack problem means that the items are either completely or no items are filled in a knapsack. For example, we have two items having weights 2kg and 3kg, respectively. If we pick the 2kg item then we cannot pick 1kg item from the 2kg item (item is not divisible); we have to pick the 2kg item completely. This is a 0/1 knapsack problem in which either we pick the item completely or we will pick that item. The 0/1 knapsack problem is solved by the dynamic programming.

Objective:

The main objective is to place the objects into the knapsack so that the maximum profit is obtained and the weight of the object chosen should not exceed the capacity of knapsack.

Definition : Given an knapsack. with following.

M - capacity of the knapsack.

n - number of objects.

W - an array consisting of weights $w_1,w_2,w_3 \ldots w_n$

P - Consisting of Profit $p_1p_2p_32 \ldots p_n$.

Example problem:

$$V[i, j] = \begin{cases} 0 & \text{if } i = j = 0 \\ V[i-1, j] & \text{if } w_i > j \\ \max(V[i-1, j], V[i-1, j-w_i] + P_i) & \text{if } w_i \le j \end{cases}$$

Note: **V[n, M]** gives the optimal profit obtained. Now, let us solve one problem and see how to get the optimal solution using knapsack problem.

Example 5.8: Apply bottom-up dynamic programming technique to the following instance of the knapsack problem with capacity M = 5

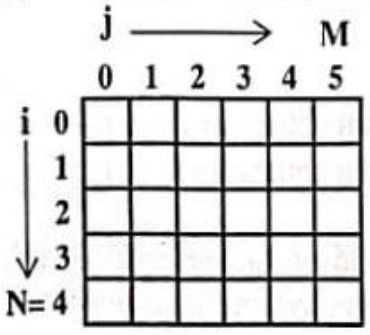
:

Item	Weight	Value
1	2	$12
2	1	$10
3	3	$20
4	2	$15

Solution: The following data is provided:

- Number of objects N = 4
- Capacity of the knapsack M = 5
- $w_1 = 2, w_2 = 1, w_3 = 3, w_4 = 2$
- $p_1 = 12, p_2 = 10, p_3 = 20, p_4 = 15$

Step 1: Since N = 4, M = 5 we have a solution matrix consisting of N+1 rows and M+1 columns as shown below:



Step 2: In this bottom up approach, we start from the initial solution with i = 0 or j = 0. Using equation (1) we have

$$V[i, j] = 0 \qquad \text{if } (i = 0 \text{ or } j = 0)$$

So, when i = 0 (i.e., when no objects are there), irrespective of the remaining capacity j = 0, 1, 2, 3, 4, 5 the profit V[i, j] = 0. Similarly, when j = 0 (i.e., when there is no knapsack) irrespective of number of objects i = 0, 1, 2, 3, 4 the profit V[i, j] = 0. These entries are made in above table. The resulting table is shown below:

j ⟶ M

i	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
N=4	0					

Profit table

Step 3: When i = 1, W_i = 2 and P_i = 12, we have to compute the results for various values of j = 1, 2, 3, 4, 5 (See row 1 of above table. Those values should be computed). The results of row 1 can be computed as shown below:

$V[i, j] = V[i-1, j]$ if $W_i > j$ .....(1)

$V[i,j] = Max(V[i-1, j], V[i-1, j-W_i]+P_i)$ if $W_i \leq j$.....(2)

j=1

W_i	j	
2	1	V[1, 1] = V[0, 1] = 0
2	2	V[1, 2] = Max(V[0, 2], V[0, 0] + 12) = Max (0 , 0 + 12) = 12
2	3	V[1, 3] = Max(V[0, 3], V[0, 1] + 12) = Max(0, 0 + 12) = 12
2	4	V[1, 4] = Max(V[0, 4], V[0, 2] + 12) = Max(0, 0 + 12) = 12
2	5	V[1, 5] = Max(V[0, 5], V[0, 3] + 12) = Max)9, 0 + 12) = 12

Note: Only when Wi = 2, = 1, relation (1) is used. For the rest we use the relation (2)

Substituting these computed values in previous profit table, following table is obtained:

j ⟶ M

i	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0					
3	0					
N=4	0					

Profit table

Step 4: When i = 2, W_i = 1 and P_i = 10, compute the results for various values of j as shown below:

$V[i, j] = V[i-1, j]$ if $W_i > j$ .....(1)

W_i j $V[i,j] = \text{Max}(V[i-1, j], V[i-1, j-W_i]+P_i)$ if $W_i \le j$.....(2)

1	1	V[2, 1] = Max(V[1, 1], V[1, 0] + 10) = Max (0, 0 + 10) = 10
1	2	V[2, 2] = Max(V[1, 2], V[1, 1] + 10) = Max(12, 0 + 10) = 12
1	3	V[2, 3] = Max(V[1, 3], V[1, 2] + 10) = Max(12, 12+10) = 22
1	4	V[2, 4] = Max(V[1, 4], V[1, 3] + 10) = Max(12, 12 + 10) = 22
1	5	V[2, 5] = Max(V[1, 5], V[1, 4] + 10) = Max(12, 12 + 10) = 22

By storing all these values in the previous profit table, the following table is obtained.

i \ j	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0					
N= 4	0					

(j ⟶ M; i ↓)

Profit table

Step 5: When i = 3, $W_i = 3$ and $P_i = 20$, the values of row 3 (of above table) are computed as shown below:

$V[i, j] = V[i-1, j]$ if $W_i > j$ .....(1)

W_i j $V[i,j] = \text{Max}(V[i-1, j], V[i-1, j-W_i]+P_i)$ if $W_i \le j$.....(2)

3	1	V[3, 1] = V[2, 1] = 10
3	2	V[3, 2] = V[2, 2] = 12
3	3	V[3, 3] = Max(V[2, 3], V[2, 0] + 20) = Max(22, 0 + 20) = 22
3	4	V[3, 4] = Max(V[2, 4], V[2, 1] + 20) = Max(22, 10+20) = 30
3	5	V[3, 5] = Max(V[2, 5], V[2, 2] + 20) = Max(22, 12 + 20) = 32

By substituting these values in the previous profit table, we have the following table:

i \ j	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
N= 4	0					

(j ⟶ M; i ↓)

Profit table

Step 6: When i = 4, $W_i = 2$ and $P_i = 15$, the values of row 4 (of above table) are computed as shown below:

$$V[i, j] = V[i-1, j] \quad \text{if } W_i > j \quad \text{.....(1)}$$

W_i j $\quad V[i,j] = \text{Max}(V[i-1, j], V[i-1, j-W_i]+P_i) \quad \text{if } W_i \le j \text{.....(2)}$

2	1	V[4, 1] = V[3, 1] = 10
2	2	V[4, 2] = Max(V[3, 2], V[3, 0] + 15) = Max(12, 15) = 15
2	3	V[4, 3] = Max(V[3, 3], V[3, 1] + 15) = Max(22, 12 + 15) = 27
2	4	V[4, 4] = Max(V[3, 4], V[3, 2] + 15) = Max(30, 12 + 15) = 27
2	5	V[4, 5] = Max(V[3, 5], V[3, 3] + 15) = Max(32, 22+15) = 37

By substituting these values in the previous profit table, we have the table shown below:

j ⟶ M

i	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
N= 4	0	10	15	27	27	37

Profit table

Optimal solution

It is given that n = 4 and M = 5. Therefore, optimal solution **V[n, M] = V[4, 5] = 37**

Now, the complete algorithm to find the optimal solution for the knapsack can be obtained using the recurrence relation

$$V[i, j] = \begin{cases} 0 & \text{if } i = j = 0 \\ V[i-1, j] & \text{if } w_i > j \\ \max(V[i-1, j], V[i-1, j-w_i] + P_i) & \text{if } w_i \le j \end{cases}$$

```
Algorithm KNAPSACK(n,m,w,p,v)
// Purpose: To find the optimal solution for the knapsack problem
using dynamic // programming
// Input:
// - n: Number of objects to be selected
// - m: Capacity of the knapsack
// - w: Weights of all the objects
// - p: Profits of all the objects

// Output:
// - v: The optimal solution for the number of objects selected
with specified remaining capacity
for i ← 1 to n do
    for j ← 0 to m do
         if(i = 0 or j = 0)
              V[i,j] ← 0
         else if(w[i] > j) then
              V[i, j] ← V[i-1, j]
         else
              V[i, j] ← max(V[i-1, j], V[i-1, j - w[i]] + p[i])
         end if
    end for
end for
```

Depth First Search

The depth first search is a method of traversing the graph by visiting each node of the graph in a systematic order. As this name implies, ***depth- first- search*** means to search the deeper in the graph

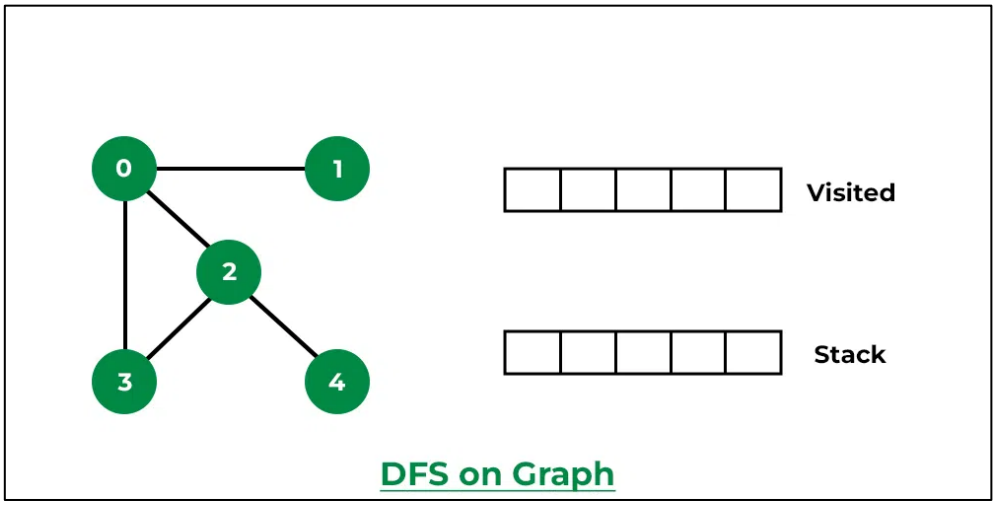
- In depth first search ***Stack*** is used.
- When a vertex is reached for the first time it pushed onto the stack and each vertex is numbered in the order in which it pushed on to the stack
- The order in which the vertices become dead ends when a vertices is dead end (i.e.. All adjacent vertices are explored). It is removed from the stack. Each node is numbered in order in which it is deleted from the stack.

The step by step process to implement the DFS traversal is given as follows -

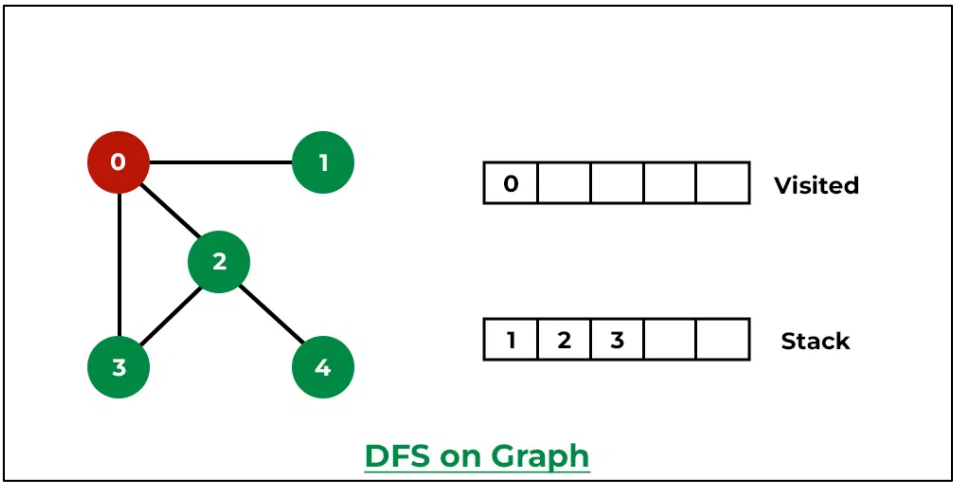
1. First, create a stack with the total number of vertices in the graph.
2. Now, choose any vertex as the starting point of traversal, and push that vertex into the stack.
3. After that, push a non-visited vertex (adjacent to the vertex on the top of the stack) to the top of the stack.
4. Now, repeat steps 3 and 4 until no vertices are left to visit from the vertex on the stack's top.
5. If no vertex is left, go back and pop a vertex from the stack.
6. Repeat steps 2, 3, and 4 until the stack is empty.

Example:

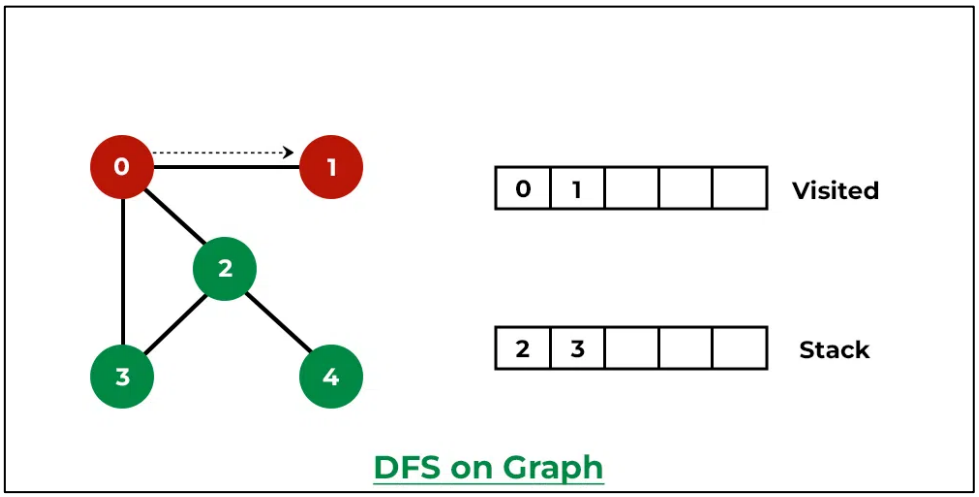
Step1: Initially stack and visited arrays are empty.



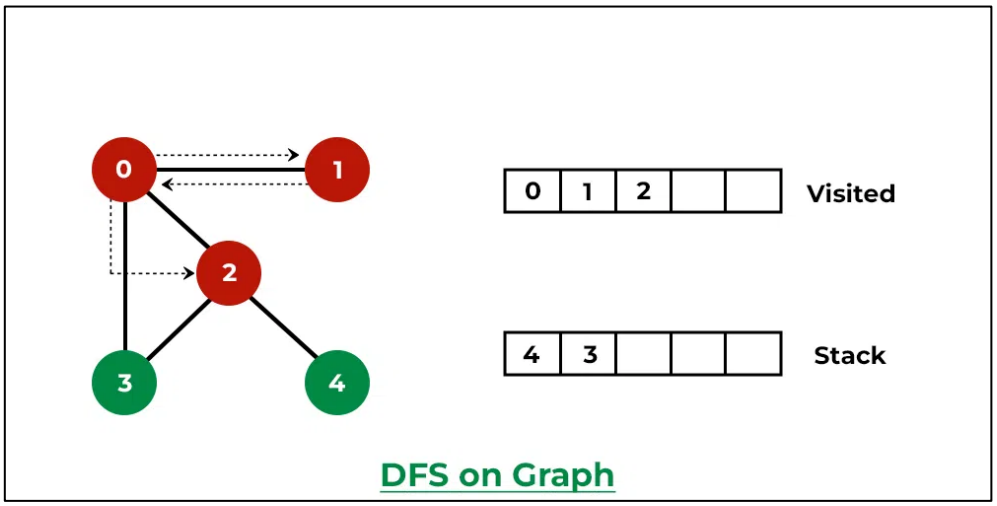
Step 2: Visit 0 and put its adjacent nodes which are not visited yet into the stack.



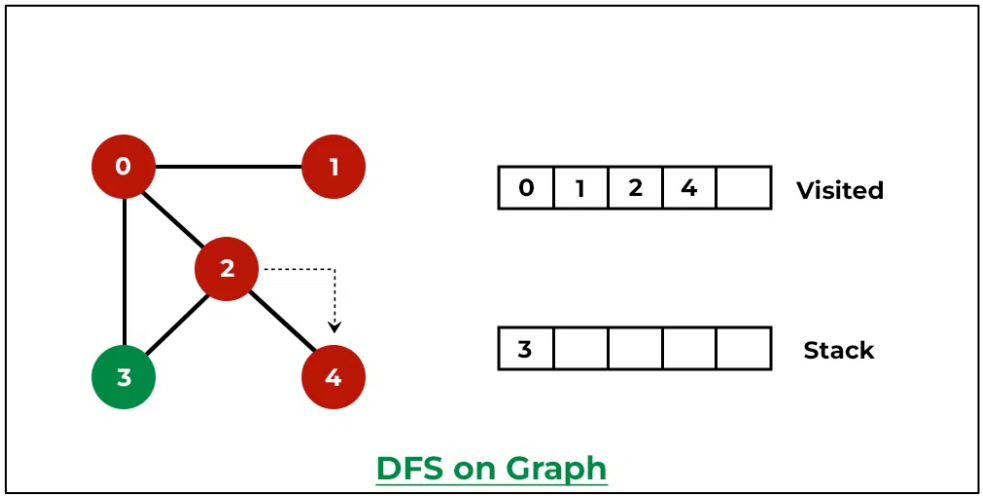
Step 3: Now, Node 1 at the top of the stack, so visit node 1 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.



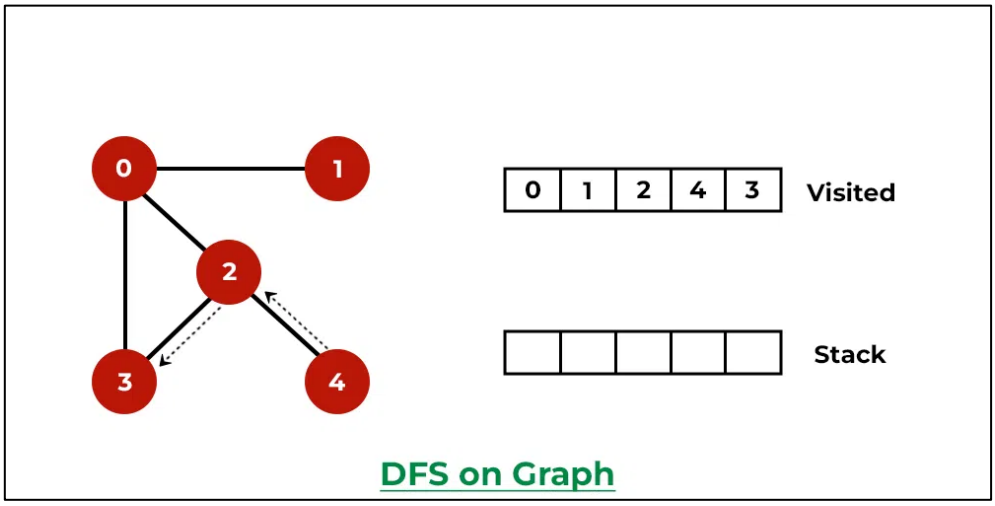
Step 4: Now, Node 2 at the top of the stack, so visit node 2 and pop it from the stack and put all of its adjacent nodes which are not visited (i.e, 3, 4) in the stack.



Step 5: Now, Node 4 at the top of the stack, so visit node 4 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.



Step 6: Now, Node 3 at the top of the stack, so visit node 3 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.



Now, Stack becomes empty, which means we have visited all the nodes and our DFS traversal ends.

Algorithm to traverse the graph using DFS

```
Algorithm: DFS (a,n,u,s,t)

// Purpose: Traverse the graph from the given node source in DFS

// Input: a - adjacency matrix of the given graph

//        n - the number of nodes in the graph

//        u - the node from where the traversal is initiated

//        s - indicates the vertices that are visited and that are
not visited
```

```
// Output: (u, v) - the nodes 'v' reachable from 'u' are stored in
a vector 't'


s[u] = 1        //visit the node
for every v adjacent to vertex u do
    if v is not visited then
        t[k][0] ← u         //v is the node visited
        t[k][1] ← v         //Store the edge u-v
        k ← k + 1
        dfs(a, n, v, s, t)
    end if
end for
```

Breadth First Search

Breadth-first search is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighboring nodes. Then, it selects the nearest node and explores all the unexplored nodes. While using BFS for traversal, any node in the graph can be considered as the root node

The step by step process to implement the BFS traversal is given as follows –

1. Create a queue and mark all vertices as not visited.

2. Choose a starting vertex and enqueue it into the queue.

3. Mark the chosen vertex as visited.

4. Dequeue a vertex from the queue and visit it.

5. Enqueue all non-visited neighbors (adjacent vertices) of the dequeued vertex.

6. Repeat steps 4 and 5 until the queue is empty.

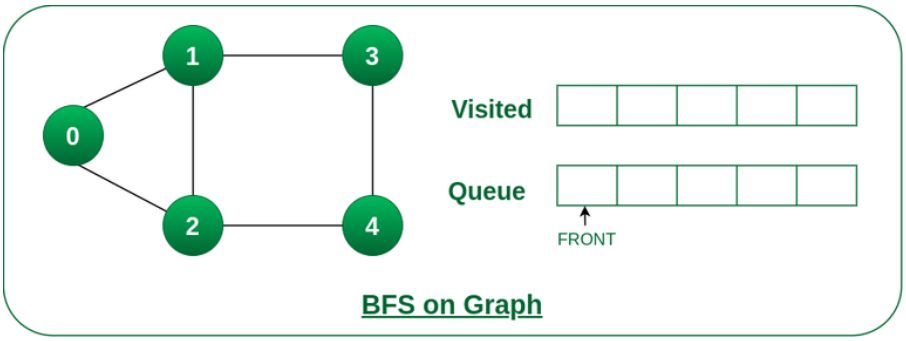
7. If there are still unvisited vertices, go back to step 2 and choose another starting vertex.

8. Repeat steps 2-7 until all vertices are visited.

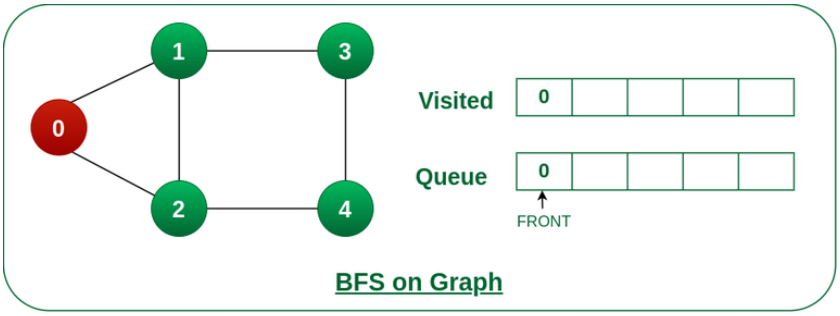
Illustration:

Let us understand the working of the algorithm with the help of the following example.

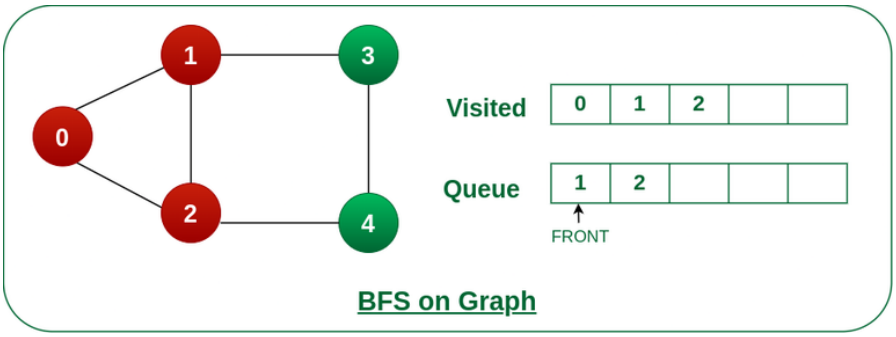
Step1: Initially queue and visited arrays are empty.



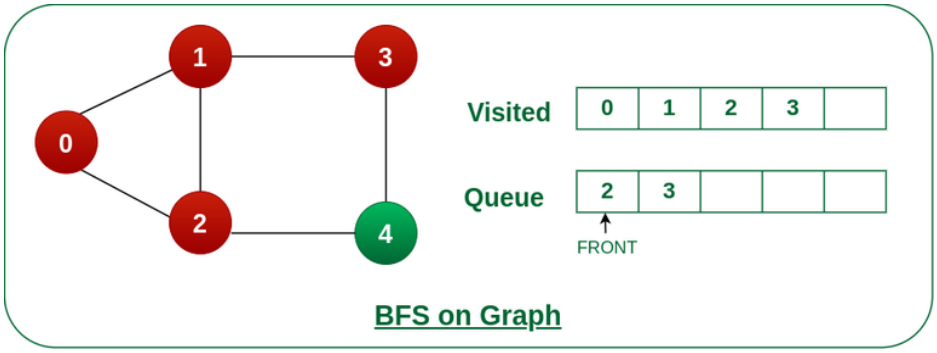
Step2: Push node 0 into queue and mark it visited.



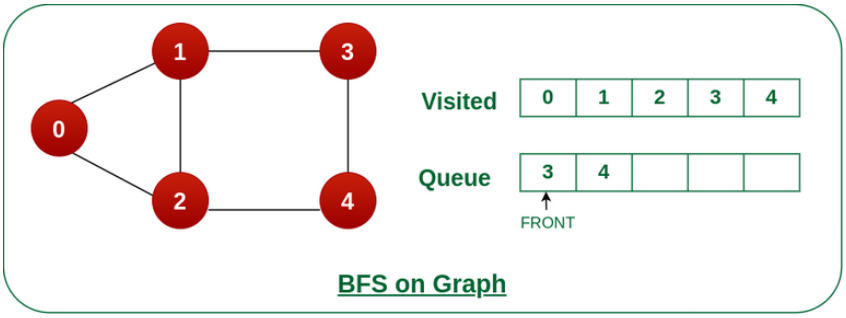
Step 3: Remove node 0 from the front of queue and visit the unvisited neighbours and push them into queue.



Step 4: Remove node 1 from the front of queue and visit the unvisited neighbours and push them into queue.

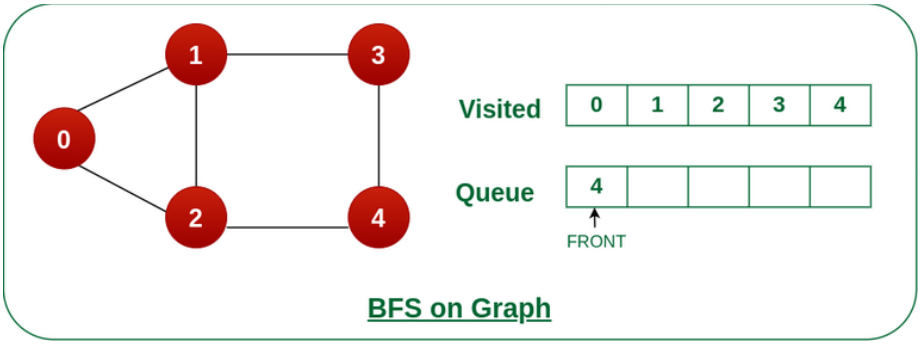


Step 5: Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.



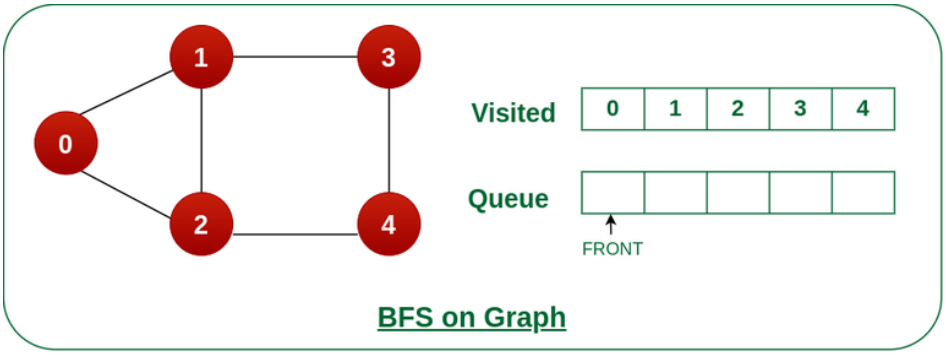
Step 6: Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.

As we can see that every neighbours of node 3 is visited, so move to the next node that are in the front of the queue.



Steps 7: Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.

As we can see that every neighbours of node 4 are visited, so move to the next node that is in the front of the queue.



Now, Queue becomes empty, So, terminate these process of iteration.

Algorithm to traverse the graph using BFS

Algorithm BFS (a,n,source,T)

```
// Purpose: Traverse the graph from the given source node in BFS
// Input: a - adjacency matrix of the given graph
//        n - the number of nodes in the graph
//        source - the node from where the traversal is initiated
// Output:
//       (u, v) - the nodes v reachable from u are stored in a vector T
for i from 0 to n-1 do
    s[i] = 0
end for
// No node is visited
f ← 0
r ← 0
q[r] ← source
s[source] ← 1
k ← 0
while (f <= r) do
    u ← q[f]
    f ← f + 1

    for every v adjacent to u do
       if s[v] is not visited
            s[v] ← 1
            r ← r + 1
            q[r] ← v

            T[k, 1] ← u
            T[k, 2] ← v,  k ← k + 1
```

```
        end if
    end for
end while
```

UNIT-4 Design and Analysis and Algorithm

Decrease-and-Conquer: Introduction, Insertion Sort, Topological Sorting
Divide-and-Conquer: Introduction, Merge Sort, Quick Sort, Binary Search, Binary Tree traversals and related properties.

Decrease-and-Conquer:

Introduction:

Definition: The decrease in conquer is a method of solving a problem by:

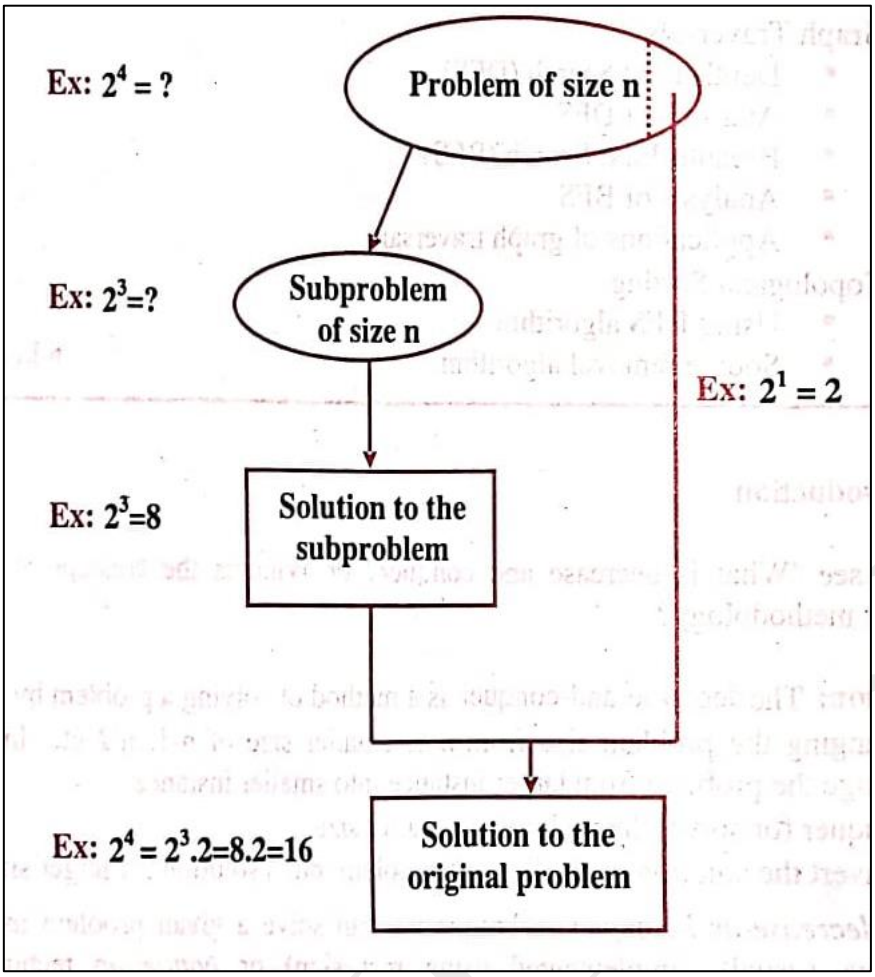
- Changing the problem size from n to smaller size n-1, n/2 etc. In other words, changing the problem from larger instances, smaller instance.
- Conquer (or solve) the problem of smaller size.
- Convert the solution of smaller size problem into a solution for larger size problem.

Using decrease and conquer technique, we can solve a given problem using top -down technique (usually implemented using recursion) or bottom up technique (usually iterative procedure) The decrease in conquer technique is slight variant of divide and conquer method.

Three Major variations of decrease and conquer method:

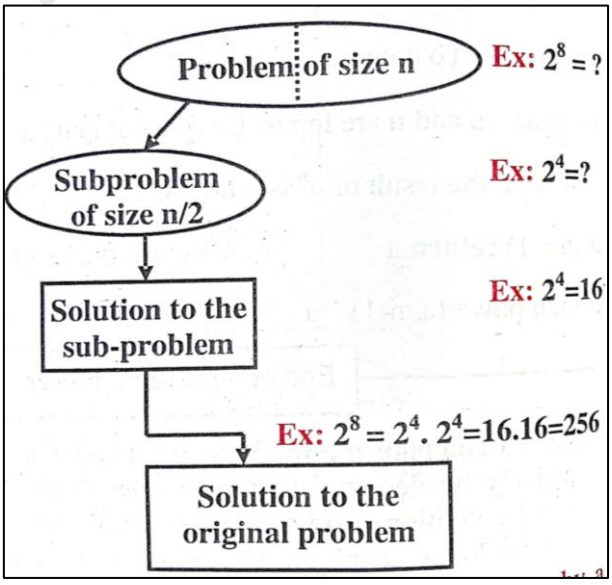
- Decrease by constant
- Decrease by constant factor
- Variable size decrease

1. **Decrease by constant**
   The decrease by constant is one of the variations of decrease and conquer technique. Here the problem size is usually decremented by one in each iteration of the loop. The various problems that can be solved using this method are computing a^n, insertion sort algorithm, traversing the graph using DFS and BFS method, topological sorting, etc
   The decrease by a constant is illustrated in the following figure.



2. Decrease by a constant factor

The decrease by constant factor is one of the variations of decrease and conquer technique here. The problem size is reduced by same constant factor (usually by 2) on each iteration of the algorithm, the idea of decrease by constant factor is illustrated in the following figure



3. Variable size decrease

The variable size decrease is one of the variation of decrease and Conquer technique. In *variable-size-decrease* Each iteration of the loop, the size reduction pattern varies from one iteration of the algorithm to another iteration.

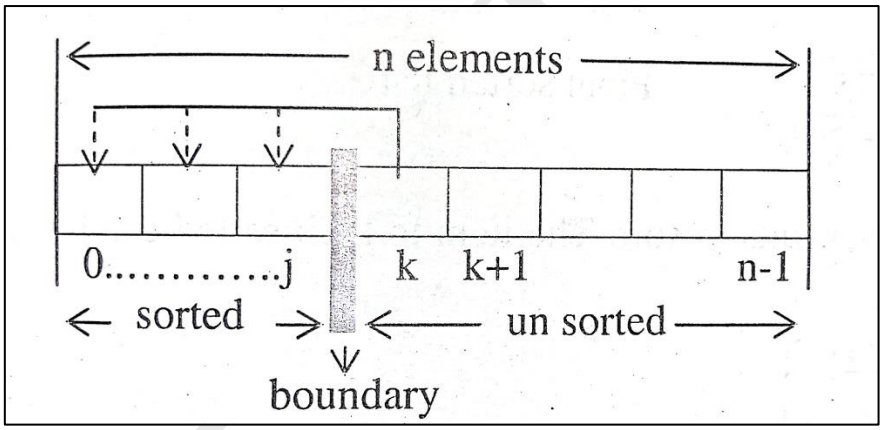
Ex: Using this idea of variable size decrease gcd of two numbers, m and n can be obtained using equilibrium algorithm

Insertion Sort

As we can arrange a number in ascending order using bubble sort, using insertion sorts. Also, we can arrange number in ascending order. The insertion sort works using principle of decrease and conquer

Procedure: The sorting procedure is similar to the way we play cards. After shuffling the cards, we pick the each card and insert into the proper place so that the card and are arranged in ascending order. The same technique is being followed while arranging the elements in ascending order

The given list is dived in to two parts: Sorted part and Unsorted part as shown below:

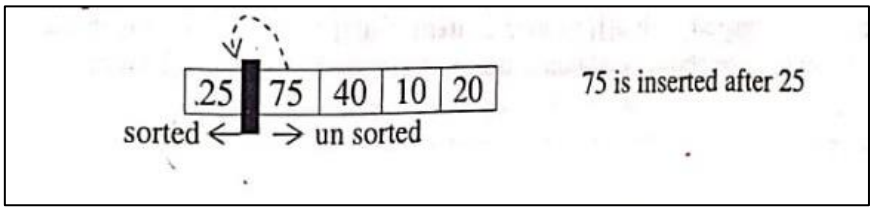


- In the above image, All the elements from zero to J are sorted, and elements from k to n-1 are not sorted
- The k^{th} Element can be inserted into appropriate position from 0 to j, so that element towards left of boundary are sorted.
- As each item is inserted towards the sorted left part. The boundary moves to the right, decreasing the sorted list.
- Finally, once the boundary moves to the right most position, the elements towards the left of boundary represent the sorted list

Working of Insertion sort:

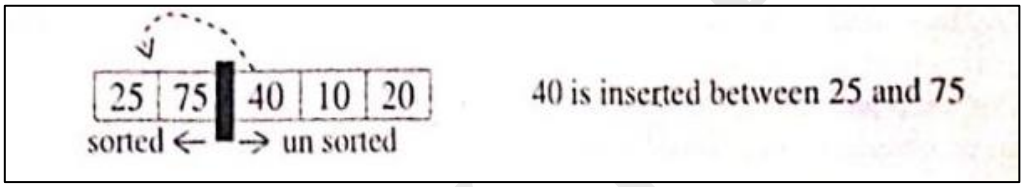
Let's say we have an unsorted array **[25,75,40,10,20]**

Iteration 1:



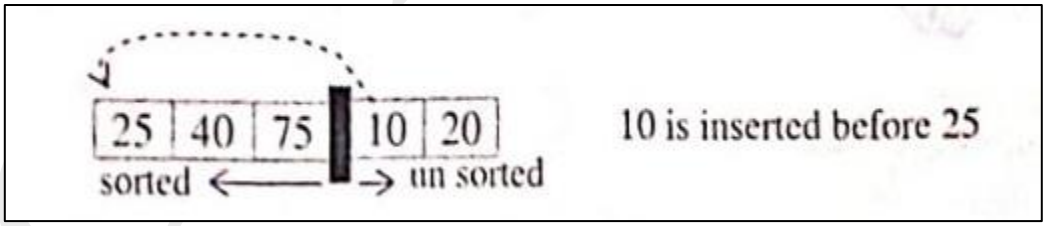
- Pick the first unsorted element, which is 75.
- Compare it with the elements in the sorted part (25). Since 75 > 25, no changes are made.
- Array remains: [25, 75, 40, 10, 20]

Iteration 2:



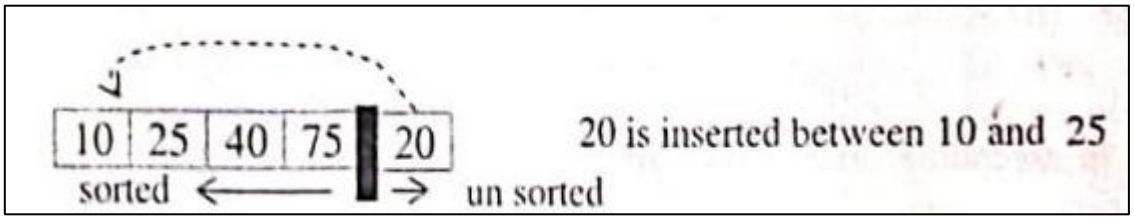
- Pick the second unsorted element, which is 40.
- Compare it with the elements in the sorted part (25, 75). Since 40 < 75, move 75 to the right and insert 40.
- Array becomes: [25, 40, 75, 10, 20]

Iteration 3:



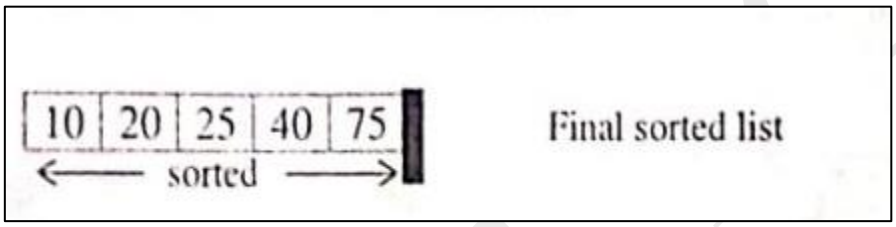
- Pick the third unsorted element, which is 10.
- Compare it with the elements in the sorted part (25, 40, 75).
- Since 10 < 75, move 75 to the right,
  10 < 40 move 40 to the right,
  10 < 25 move 25 to the right and finally insert 10.
- Array becomes: [10, 25, 40, 75, 20]

Iteration 4:



- Pick the fourth unsorted element, which is 20.
- Compare it with the elements in the sorted part (10, 25, 40, 75).
  Since 20 < 75, move 75 to the right,
  20 < 40 move 40 to right
  20 < 25 move 25 to right finally insert 20 at the correct position.
- Array becomes: [10, 20, 25, 40, 75]

Final Sorted Array:



Algorithm for insertion Sort:

```
Algorithm insertionSort(a,n)
// Purpose: Sorts an array of comparable elements in ascending
//          order using the Insertion Sort algorithm.
// Input: a - The array to be sorted.
//        n – the total number of elements in the list to be
//            sorted
// Output: a – the list is sorted
for i <- 1 to n - 1
    item <- arr[i]
     j <- i - 1

    while (j >= 0 and arr[j] > key)
        arr[j + 1] <- arr[j]
        j <- j – 1
   end while
   arr[j + 1] <- key
end for
```

Analysis:

TIME COMPLEXITY OF INSERTION SORT

The insertion sort algorithm encompasses a time complexity of **O(n2)**

Time Complexities:

- **Best Case Complexity:** The insertion sort algorithm has a best-case time complexity of O(n) for the already sorted array because here, only the outer loop is running n times, and the inner loop is kept still.
- **Average Case Complexity:** The average-case time complexity for the insertion sort algorithm is O(n2), which is incurred when the existing elements are in jumbled order, i.e., neither in the ascending order nor in the descending order.
- **Worst Case Complexity**: The worst-case time complexity is also O(n2), which occurs when we sort the ascending order of an array into the descending order.

Topological Sorting

Definition:

The topological sorting of directed acyclic graph (DAG) G = (V,E) is linear ordering of all the vertices such that for every edge(u,v) in graph G, the vertex u appears before the vertex v in the ordering.

A topological sort of graph can be viewed as an ordering of vertices along a horizontal line so that all directed edges go from left to right for cyclic graph, no linear ordering is possible.

Note:

If A depends on B and B depends on A, then it is cyclic, A graph which is cyclic does not have topological sequence.

Example: To get the job, one should have work experience. But to get the work experience, one should have a job

The topological sorting can be done using following two methods:

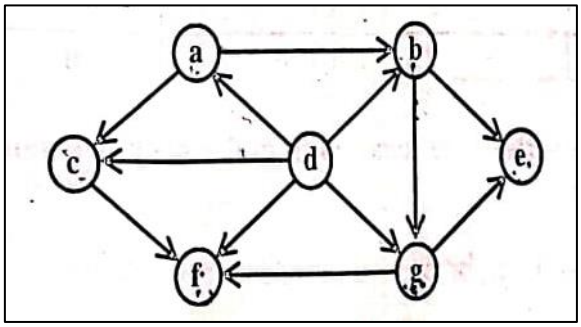
- DFS method
- Source removal method

Topological sort using DFS method:

The topological order using DFS method can be obtained as shown below:

- **Step1:** select any arbitrary vertex
- **Step2:** when a vertex is visited from the first time, it is pushed on to the stack
- **Step3:** When a vertex becomes a dead end, it is removed from the stack
- **Step4:** Repeat step 2 to 3 for all vertices in the graph
- **Step5:** Reverse the order of deleted items to get that topological sequence

Eg: **Apply the DFS based algorithm to solve that apological sorting problem for the following graph**



Note: The solution is similar to DFS Traversal, Since vertex a is the least in alphabetical order, it is selected as the start vertex

Initial step: Push start vertex a onto the stack and add it to yes, indicating the vertex is marked or visited

Stage 1: The various activities that are performed are shown below:

Step1: Stack not empty top of Stackies Vertex a

Step2: Obtain the first node, which is adjacent to vertex a i.e.. vertex b.

Step3: Add Valtex b to S indicating it is visited or marked.

Step4: Not perform since a vertex is added to S. This step is performed if nothing is added to S

On similar lines perform various stages and express the solution as shown in the table below:

below:

	Step1	Step 2	Step 3	Step 4
	Stack	v=adj(s[top])	Nodes visited S	pop(stack)
Initial step	a	-	a	-
Stage 1	a	b	a, b	-
Stage 2	a, b	e	a, b, e	-
Stage 3	a, b, e	-	a, b, e	e
Stage 4	a, b	g	a, b, e, g	-
Stage 5	a, b, g	f	a, b, e, f, g	-
Stage 6	a, b, g, f	-	a, b, e, f, g	f
Stage 7	a, b, g	-	a, b, e, f, g	g
Stage 8	a, b	-	a, b, e, f, g	b
Stage 9	a	c	a, b, c, e, f, g	-
Stage 10	a, c	-	a, b, c, e, f, g	c
Stage 11	a	-	a, b, c, e, f, g	a
	Note: Stack is empty. So, take the next vertex in sequence which is not visited and push it onto stack and add to S.			
Stage 12	d	-	a, b, c, d, e, f, g	-
Stage 13	d	-	a, b, c, d, e, f, g	d

e, f, g, b, c, a, d (popped sequence)

The topological order is obtained by reversing the above popped order, which is given by:

d → a → c → b → g → f → e (Topological order)

Algorithm to traverse the graph Topological sorting Using DFS

```
ALGORITHM DFS(u, n, a)
// Purpose: To obtain the sequence of jobs to be executed resulting in //
            topological order
// Input:
// u - Starting vertex for DFS traversal
// n - Number of vertices in the graph
// a - Adjacency matrix of the given graph
// Global variables:
// s - To keep track of visited and unvisited nodes
// j - Index variable to store the vertices (only those nodes which are dead
ends or those nodes whose nodes are completely explored)
// res - An array which holds the order in which the vertices are popped
// Output:
// res - Indicates the vertices in reverse order that are to be executed
```

```
Step 1: [Visit the vertex u]
    s[u] ← 1
Step 2: [Traverse deeper into the graph until we reach a dead end or until all
vertices are visited]
    for v from 0 to n-1 do
        if (a[u][v] = 1 and s[v] = 0) then
            DFS(v, n, a)
        end if
    end for
Step 3: [Store the dead vertex or which is completely explored]
    j = j + 1
    res[j] ← u

Step 4: [Finished]
    Return
```

Analysis of topology calls out using DFS method (represented as adjacency matrix)

The running time of function DFS is the time required to execute the statement

for v ← 0 **to** n-1 **do**
if (a[u][v] = 1 **and** s[v] = 0) **then**
DFS(v, n, A) // This statement is executed *n* times

So, the time efficiency is given by the following relation:

$$
\begin{aligned}
f(n) &= \sum_{v=0}^{n-1}\sum_{v=0}^{n-1} 1 \\
&= \sum_{v=0}^{n-1} n-1-0+1 \\
&= \sum_{v=0}^{n-1} n \\
&= n\sum_{v=0}^{n-1} 1 \\
&= n(n-1-0+1) \\
&= n^2
\end{aligned}
$$

So, the time complexity of topological sort is given by $\theta(n^2)$. For a given graph G = (V,E) where V is the number of vertices(Here, we are using *n* as the number of vertices),

$$n = |V|$$

Therefore, time complexity of topological sort is given by $\theta(|V^2|)$.

Topological sort using source removal method

This method is based on decrease and conquer technique in this method. a vertex with no incoming edges, (whose in degrees 0) is selected and deleting along with the outgoing edges.

If there is several vertices with no incoming edges, arbitrarily, our vertex is selected.

The order. In which the vertices are visited and deleted one by one result in topological sorting.

For example, Consider the graph shown in figure below, where 1,2,3,4 and represent the task to be completed.

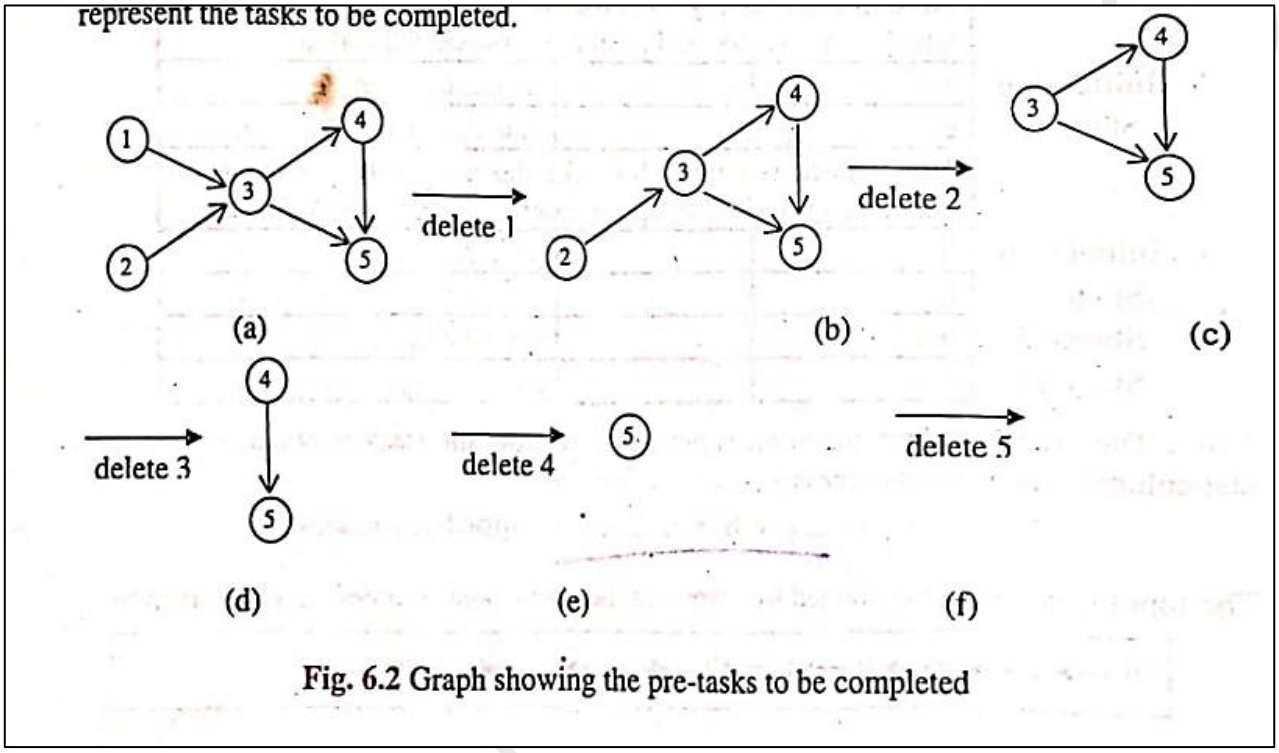


Fig. 6.2 Graph showing the pre-tasks to be completed

- It is clear from figure (a) that tasks 1 and 2 are independent and one of them can be selected and processed and graph shown in figure (b) is obtained.
- Now, the next independent task is 2 (figure b) and can be deleted.
- The sequence of vertices that are processed and deleted are shown in figure (a) through (f).
- Thus, the order in which each task has to be completed is given by 1, 2, 3, 4 and 5. The topological sequence is shown below:

1 → 2 → 3 → 4 → 5 (Topological order)

Working of Topological sorting using source removal method:

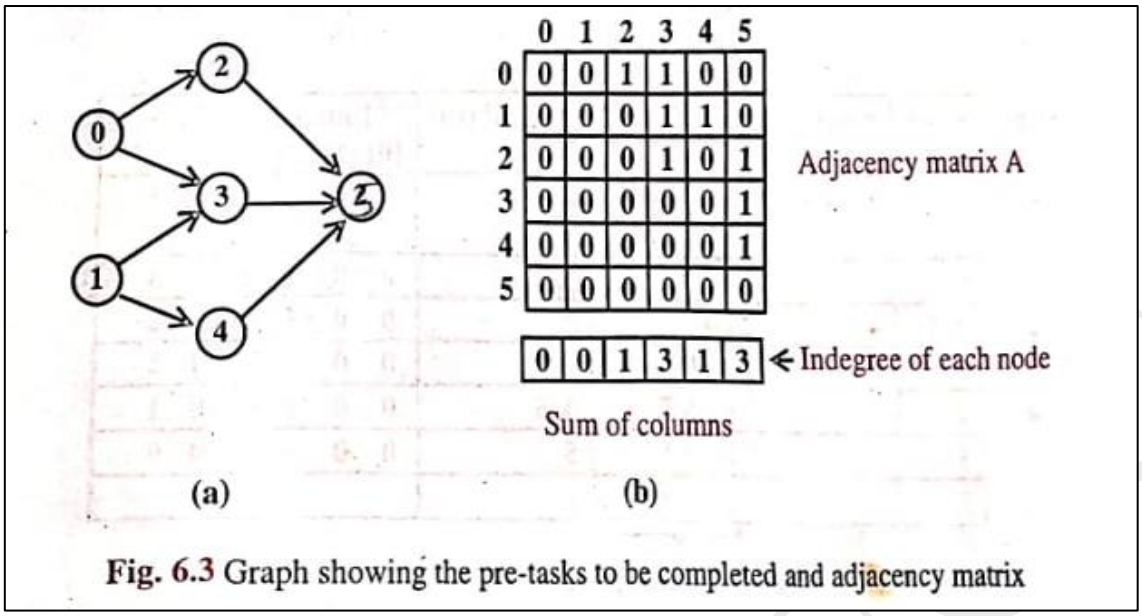


Fig. 6.3 Graph showing the pre-tasks to be completed and adjacency matrix

- By adding the columns of adjacency matrix, then a degree of each vertex is obtained
- The vertices for which the inner degree 0 are independent task, which do not depend on any task, and can be computed Independently

Observe the following facts:

- Indegree[0] = Indegree[1] = 0. Since, indegree of jobs 0 and 1 is 0, these jobs are considered independent jobs.
- Indegree[2] = 1 indicates that job 2 depends on one job (namely job 0).
- Indegree[3] = 3 indicates that job 3 depends on 3 jobs (namely, 0,1 and 2)
- Indegree[4] = 1 indicates that job 4 depends on only one job (i.e., 1)
- Indegree[5] = 3 which indicates that job 5 depends on three jobs (namely 2, 3 and 4).

So, the various steps involved in the design have to be performed repeatedly till the stack is empty:

- Find the vertices whose indegree is zero and place them on the stack (shown in second column of table). These vertices denote the jobs which do not depend on any other job and can be performed independently.
- Pop a vertex u and it is the task to be done (third column of table)
- Add the vertex u to the solution vector (fourth column of table). This vertex represents the next job to be considered for executing.
- Find the vertices v adjacent to the vertex u. The vertices v represents the jobs which depend on job u (fifth column in the table)
- Decrement indegree[v] by one thereby reducing the number dependencies on v by one. Since, indegree[v] gives the number of jobs to be completed before job v and

job u is completed, the dependent jobs of v are reduced by one (shown in sixth column of table)

Steps	stack	u = pop( )	Solution T	v = adj (u)	Indegree of jobs [0] [1] [2] [3] [4] [5]
Initial step	Find indegree of each node →				0 0 1 3 1 3
1	0, 1	1	1	3, 4	0 0 1 2 0 3
2	0, 4	4	1 4	5	0 0 1 2 0 2
3	0	0	1 4 0	2 , 3	0 0 0 1 0 2
4	2	2	1 4 0 2	3, 5	0 0 0 0 0 1
5	3	3	1 4 0 2 3	5	0 0 0 0 0 0
6	5	5	1 4 0 2 3 5	-	-

Topological sequence

Here are the detailed steps for each row in the table:

Step 0:

- Stack: 0, 1
- Solution vector v: -[ ]
- Indegree of jobs: 0 1 3 13

Step 1:

- Node u: 1
- Solution vector v: [1]
- Adj(u): 3, 4
- Indegree of jobs: 0 0 1 0 2 3
    - Jobs 0 and 1 are independent (indegree is zero), so they are pushed onto the stack.
    - Job 1 is popped from the stack and added to the solution vector.
    - The nodes adjacent to 1 (jobs 3 and 4) have their indegrees reduced by one.
    - Indegree[3] and Indegree[4] are now 0 and 1, respectively.
    - Job 4 is pushed onto the stack.
    - Job 4 is popped from the stack and added to the solution vector.
    - The node adjacent to 4 (job 5) has its indegree reduced by one.
    - Indegree[5] is now 0.

Step 2:

- Node u: 4
- Solution vector v: 14
- Adj(u): 5
- Indegree of jobs: 0 0 1 2 0 2
    - Job 5 is pushed onto the stack.
    - Job 5 is popped from the stack and added to the solution vector.
    - The node adjacent to 5 (none in this case) is processed.

Step 3:

- Node u: 0
- Solution vector v: 140
- Adj(u): 2, 3
- Indegree of jobs: 0 0 0 1 0 2
    - Job 2 is pushed onto the stack.
    - Job 2 is popped from the stack and added to the solution vector.
    - The nodes adjacent to 2 (jobs 3) have their indegrees reduced by one.
    - Indegree[3] is now 0.

Step 4:

- Node u: 2
- Solution vector v: 1402
- Adj(u): 3, 5
- Indegree of jobs: 0 0 0 0 0 1
    - Jobs 3 and 5 are pushed onto the stack.
    - Job 3 is popped from the stack and added to the solution vector.
    - The nodes adjacent to 3 (none in this case) are processed.
    - Job 5 is popped from the stack and added to the solution vector.
    - The node adjacent to 5 (none in this case) is processed.

Step 5:

- Node u: 3
- Solution vector v: 14023
- Adj(u): 5
- Indegree of jobs: 0 0 0 0 0 0
    - The nodes adjacent to 3 (none in this case) are processed.

Step 6:

- Finaly the Topological Sequence : **1 → 4 → 0 → 2 → 3 → 5**

Divide-and-Conquer

Divide and conquer method of designing the algorithm is one of the best known method of solving a problem.

Definition: Divide and conquer method is a top-down technique for designing algorithms which consist of dividing the problem into smaller sub problems hoping that the solutions of the sub problems are easier to find.

The solutions of all smaller problems are then combined to get a solution for the original problem.

The divide and conquer technique of solving a problem involves three steps at each level of the recursion:

Divide: The problem is divided into a number of sub problems

Conquer: If the sub problems are smaller in size, the problem can be solved using straightforward method. If the sub problems are larger in size, they are divided into number of sub-problems of the same type and size. Each of the sub-problems is solved recursively.

Combine: The solutions of sub problems are combined to get the solution for the larger problem.

The divide and conquer technique is represented as shown in below figure :

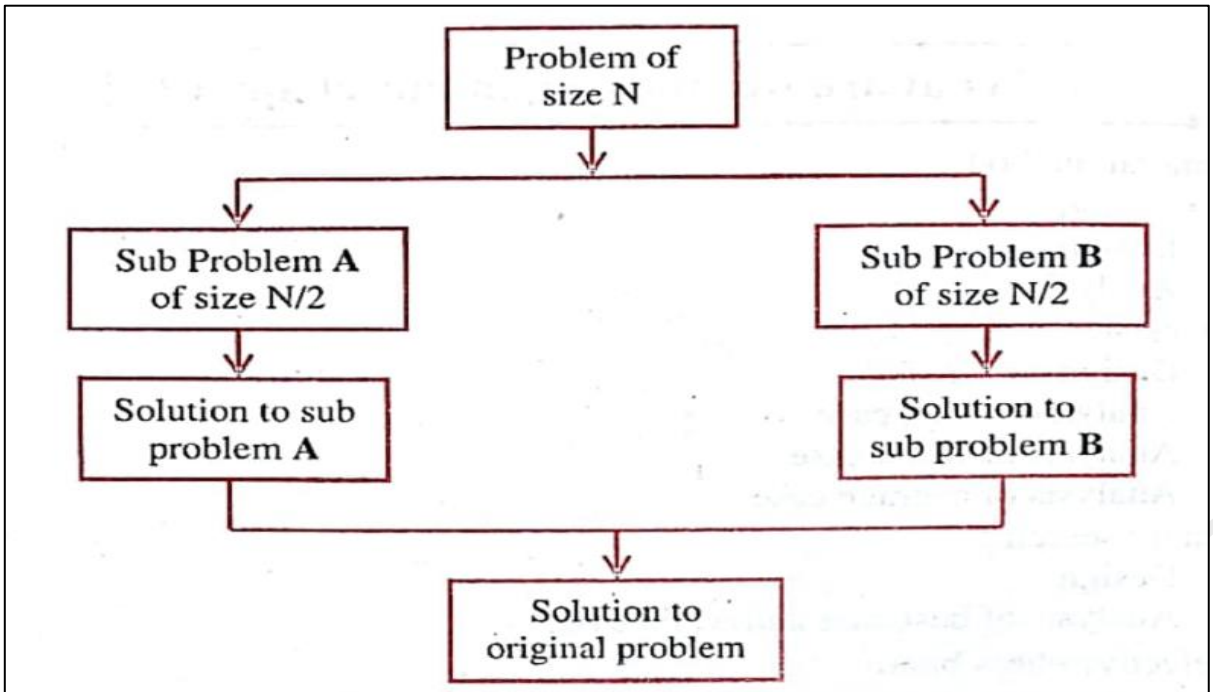


Figure 3.1 Divide and Conquer technique

Merge Sort

Merge sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array

The merge sort works very well on large data set

How does Merge Sort work?

Merge sort is a recursive algorithm that continuously splits the array in half until it cannot be further divided i.e., the array has only one element left (an array with one element is always sorted). Then the sorted subarrays are merged into one sorted array.

The various tips that are involved while sorting using merge sort are shown below:

- **Divide**: Divide the given array consisting of n elements into two parts of n/2 elements each until it cannot further divide.
- **Conquer**: Sort the left part of the array and right part of the array recursively using merge sort.
- **Combine**: Majda sorted left part and sorted rate part to get the single sorted array

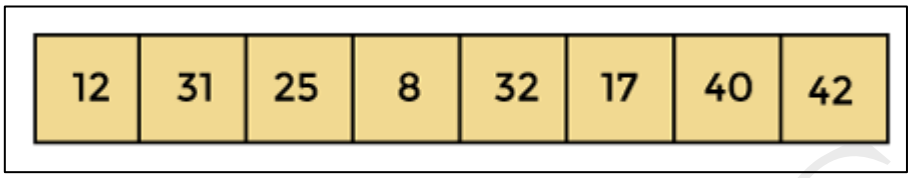
The key operation is merging sort is combining the sorted left part and sorted right part into a single sorted array. This process of merging of two sorted vectors into single sorted vector is called simple merge. This only necessary condition for the problem is that both array should be sorted.

Working of Merge sort Algorithm

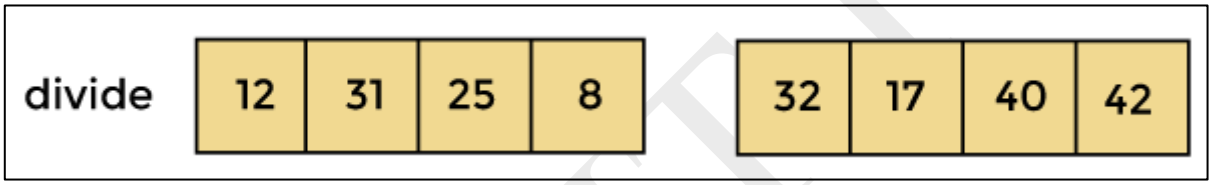
Now, let's see the working of merge sort Algorithm.

- To understand the working of the merge sort algorithm, let's take an unsorted array. It will be easier to understand the merge sort via an example.

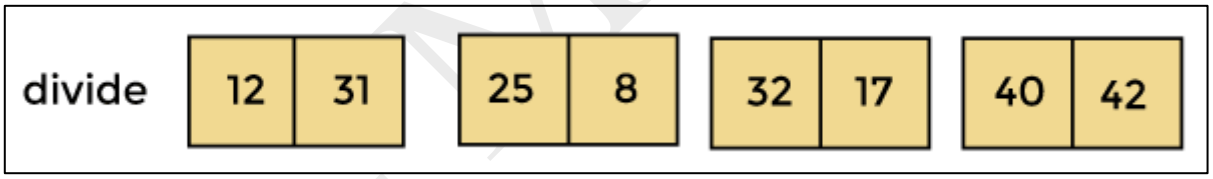
Let the elements of array are -



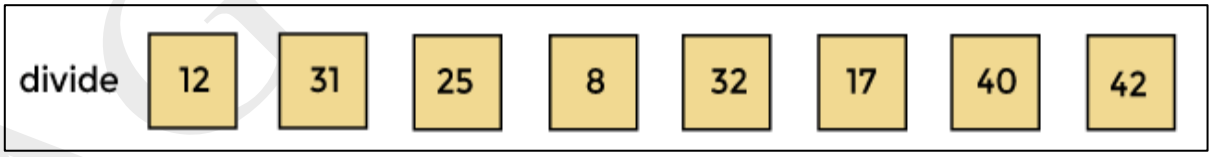
- According to the merge sort, first divide the given array into two equal halves. Merge sort keeps dividing the list into equal parts until it cannot be further divided.
- As there are eight elements in the given array, so it is divided into two arrays of size 4.



- Now, again divide these two arrays into halves. As they are of size 4, so divide them into new arrays of size 2.

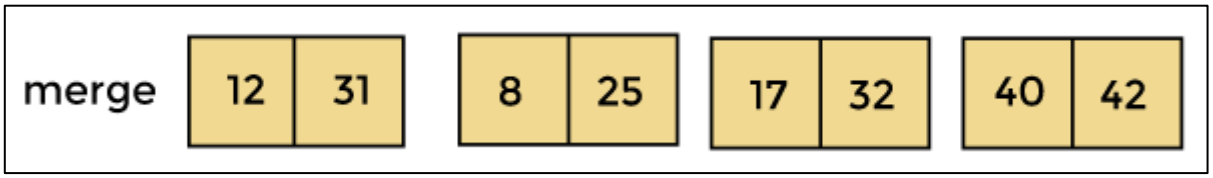


- Now, again divide these arrays to get the atomic value that cannot be further divided.

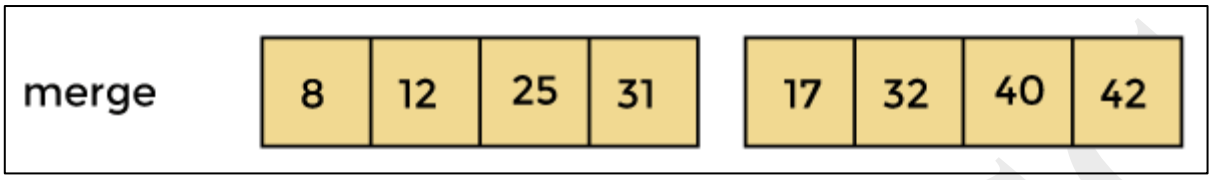


- Now, combine them in the same manner they were broken.
- In combining, first compare the element of each array and then combine them into another array in sorted order.

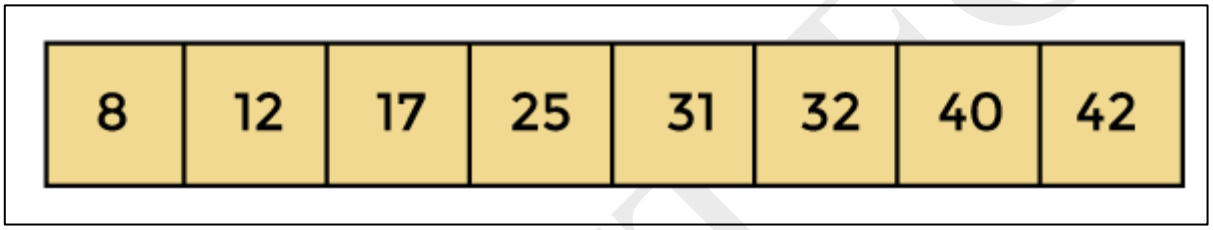
  - So, first compare 12 and 31, both are in sorted positions.
  - Then compare 25 and 8, and in the list of two values, put 8 first followed by 25.
  - Then compare 32 and 17, sort them and put 17 first followed by 32. After that,
  - compare 40 and 42, and place them sequentially.



- In the next iteration of combining, now compare the arrays with two data values and merge them into an array of found values in sorted order.



- Now, there is a final merging of the arrays. After the final merging of above arrays, the array will look like -



Example for Trace of Merge Sort:

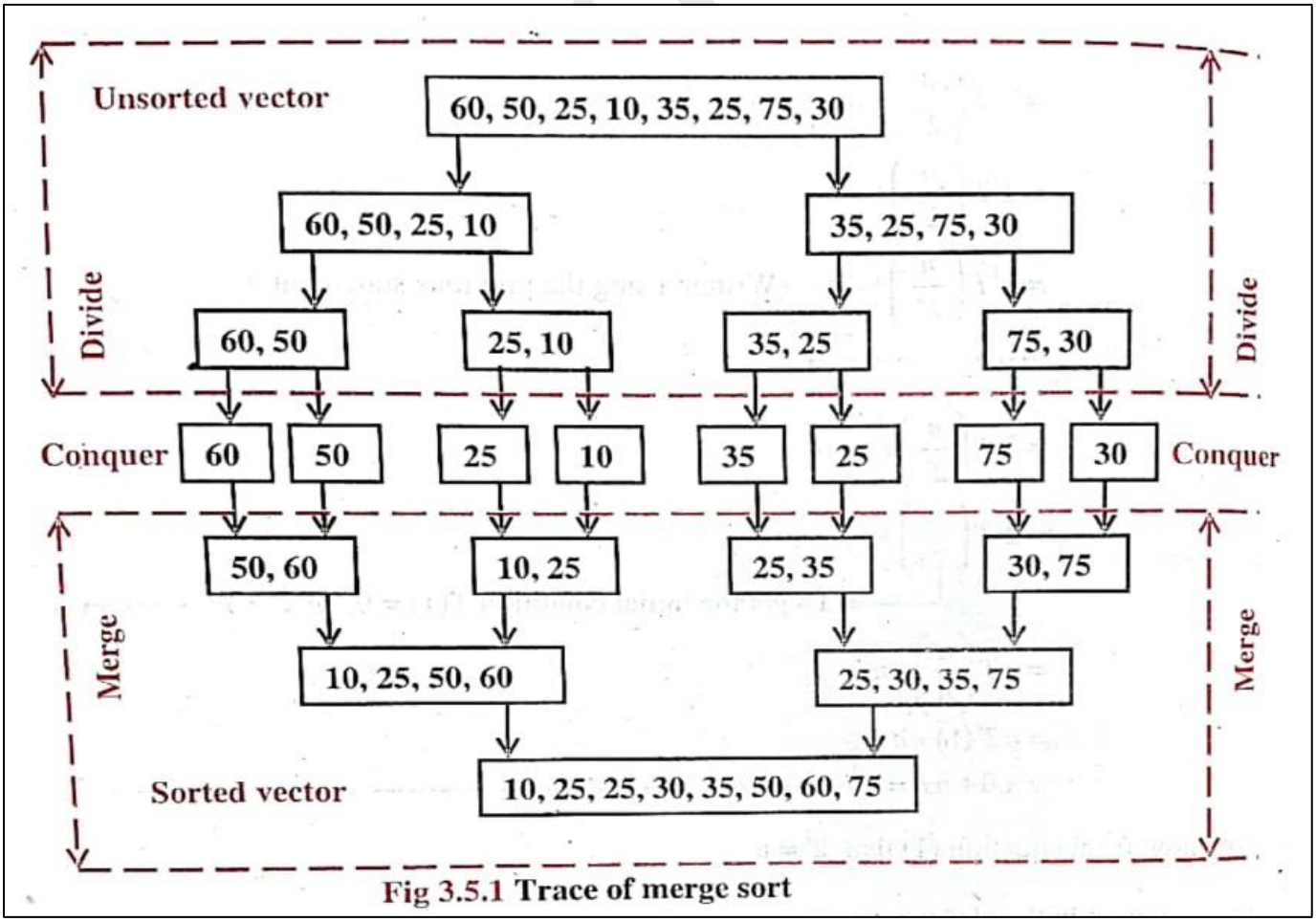


Fig 3.5.1 Trace of merge sort

Analysis of Merge Sort:

Let T (n) be the total time taken by the Merge Sort algorithm.

- Sorting two halves will take at the most $2T\frac{n}{2}$ time.
- When we merge the sorted lists, we come up with a total n-1 comparison because the last element which is left will need to be copied down in the combined list, and there will be no comparison.

Thus, the relational formula will be

$$T(n) = 2T\left(\frac{n}{2}\right) + n - 1$$

But we ignore '-1' because the element will take some time to be copied in merge lists.

So $T(n) = 2T\left(\frac{n}{2}\right) + n$...equation 1

Note: Stopping Condition T (1) =0 because at last, there will be only 1 element left that need to be copied, and there will be no comparison.

Putting $n=\frac{n}{2}$ in place of n in .....equation 1

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{2^2}\right) + \frac{n}{2}$$.....equation2

Put 2 equation in 1 equation

$$T(n) = 2\left[2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right] + n$$

$$= 2^2T\left(\frac{n}{2^2}\right) + \frac{2n}{2} + n$$

$$T(n) = 2^2\,T\left(\frac{n}{2^2}\right) + 2n$$.....equation 3

Putting $n=\frac{n}{2^2}$ in equation 1

$$T\left(\frac{n}{2^3}\right) = 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}$$.....equation4

Putting 4 equation in 3 equation

$T(n) = 2^2 \left[2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\right] + 2n$

$T(n) = 2^3\ T\left(\frac{n}{2^3}\right) + n + 2n$

$T(n) = 2^3\ T\left(\frac{n}{2^3}\right) + 3n$.....equation5

From eq 1, eq3, eq 5.....we get

$T(n) = 2^i\ T\left(\frac{n}{2^i}\right) + in$.....equation6

From Stopping Condition:

$\frac{n}{2^i} = 1$ And $T\left(\frac{n}{2^i}\right)=0$

$n = 2^i$

Apply log both sides:

$\log n=\log_2 i$
$\log n=i\log 2$
$\frac{\log n}{\log 2} = i$

$\log_2 n=i$

From 6 equation

$T(n) = 2^i\ T\left(\frac{n}{2^i}\right) + in$

$= 2^i \times 0 + \log_2 n \,.\, n$

$= T(n) = n.\log n$

Best Case Complexity: The merge sort algorithm has a best-case time complexity of **O(n*log n)** for the already sorted array.

Average Case Complexity: The average-case time complexity for the merge sort algorithm is **O(n*log n)**, which happens when 2 or more elements are jumbled, i.e., neither in the ascending order nor in the descending order.

Worst Case Complexity: The worst-case time complexity is also **O(n*log n)**, which occurs when we sort the descending order of an array into the ascending order.

Space Complexity: The space complexity of merge sort is **O(n)**.

Algorithm for Merge Sort:

```
Algorithm MergeSort(A, low, high)
//    Purpose: Sort the elements of the array between the lower bound and upper
      bound
//    Input: A is an unsorted vector with low and high as lower bound and upper
      bound
//    Output: A is a sorted vector

if (low > high) then
    return
end if
mid ← (low + high) / 2          // Divide the array into equal parts
MergeSort(A, low, mid)          // Sort the left part of the array
MergeSort(A, mid + 1, high)     // Sort the right part of the array
SimpleMerge(A, low, mid, high)  // Merge the left part and right part

Algorithm SimpleMerge(A, low, mid, high)
//    Purpose: Merge two sorted arrays where the first array starts from low to
      mid and the second starts from mid+1 to high
//    Input: A is sorted from index low to mid, A is sorted from index mid+1 to
      high
//    Output: A is sorted from index low to high.
i ← low
j ← mid + 1
k ← low

while (i <= mid and j <= high)
    if (A[i] < A[j]) then
        C[k] ← A[i]  // Copy the lowest element from the first part of A to C
        i ← i + 1    // Point to the next item in the left part of A
    else
        C[k] ← A[j]  // Copy the lowest element from the second part of A to C
        j ← j + 1    // Point to the next item in the second part of A
    end if
k ← k + 1
    // Point to the next item in C
end while
```

```
while (i <= mid)
    // Copy the remaining items from the left part of A to C
    C[k] ← A[i]
    k ← k + 1
    i ← i + 1
end while

while (j <= high)
    // Copy the remaining items from the right part of A to C
    C[k] ← A[j]
    k ← k + 1
    j ← j + 1
end while

for i = low to high
    A[i] ← C[i]
    // Copy the elements from vector C to vector A
end for
```

Quick Sort

Quick Sort is a sorting algorithm based on the Divide and Conquer algorithm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

This algorithm follows the divide and conquer approach. Divide and conquer is a technique of breaking down the algorithms into subproblems, then solving the subproblems, and combining the results back together to solve the original problem.

Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

Conquer: Recursively, sort two subarrays with Quicksort.

Combine: Combine the already sorted array.

Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.

After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.

Choosing the pivot

Picking a good pivot is necessary for the fast implementation of quicksort. However, it is typical to determine a good pivot. Some of the ways of choosing a pivot are as follows -

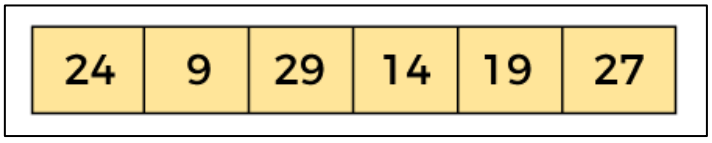
- Pivot can be random, i.e. select the random pivot from the given array.
- Pivot can either be the rightmost element of the leftmost element of the given array.
- Select median as the pivot element.

Working of Quick Sort Algorithm

Now, let's see the working of the Quicksort Algorithm.

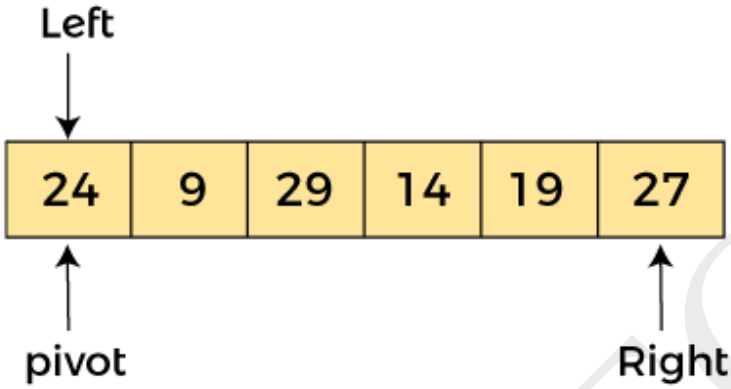
To understand the working of quick sort, let's take an unsorted array. It will make the concept more clear and understandable.

Let the elements of array are -



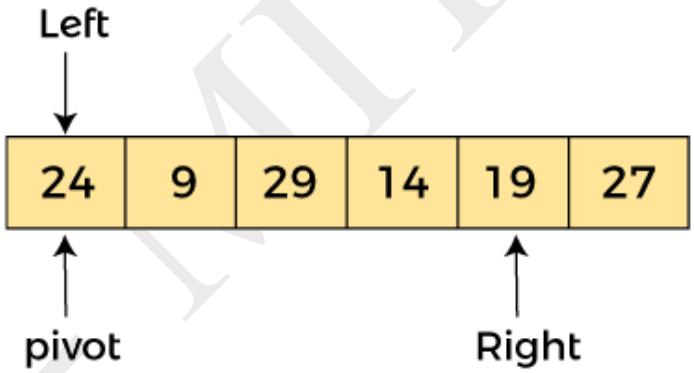
Step 1:

- Choose the leftmost element 24 as the pivot.
- Set left to the leftmost index [0] = 24, and right to the rightmost index [5] = 27.
- Compare a[pivot] = 24 with a[right]=27 . Since 24 < 27, move the right pointer one position to the left.



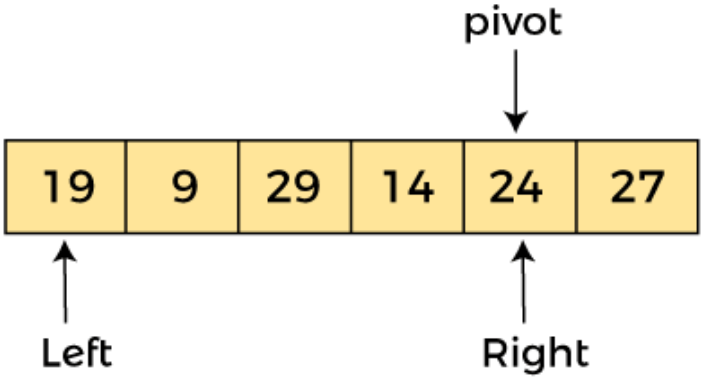
Step 2:

- Now, compare a[pivot]= 24 with a[right] = 19 . Since 24 > 19, swap a[pivot] and a[right] and move the pivot to the right.



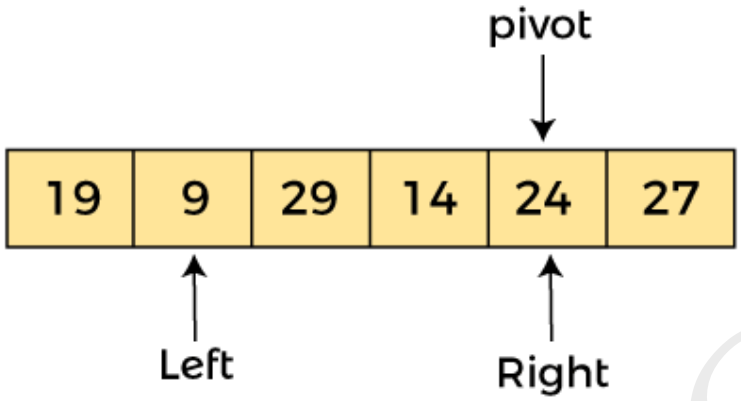
Step3:

- Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right.
- As a[pivot] > a[left], so algorithm moves one position to right as :



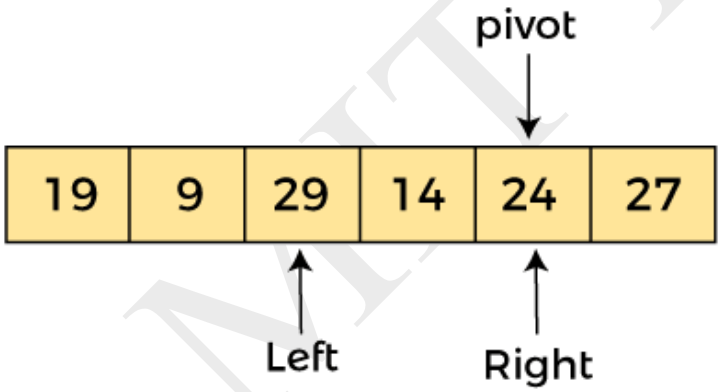
Step 4:

- Now, a[left] = 9, a[right] = 24, and a[pivot] = 24.
- As a[pivot] > a[left] 24 >9 , so algorithm moves one position to right as –.



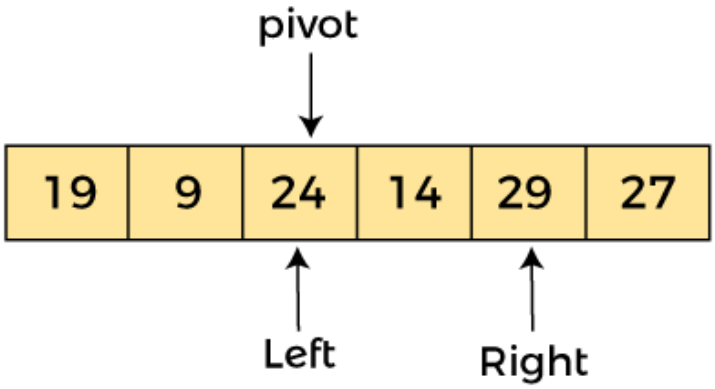
Step5:

- Now, a[left] = 29, a[right] = 24, and a[pivot] = 24
- As a[pivot] < a[left] 24 < 29 so, swap a[pivot] and a[left]



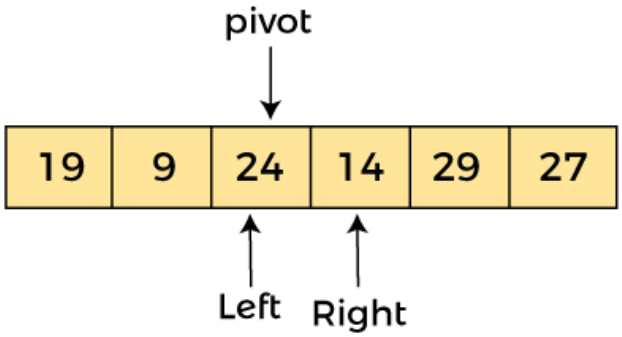
Step6:

- Now pivot is at left, i.e. –
- Now, a[pivot] = 24, a[left] = 24, and a[right] = 29



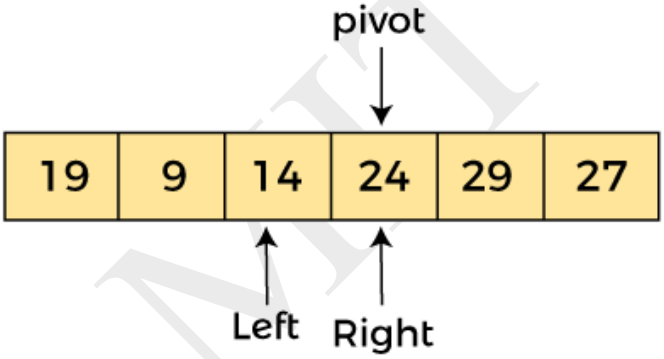
Step7:

- As a[pivot] < a[right] = 24 < 29 , so algorithm moves one position to left, as
- Now, a[pivot] = 24, a[left] = 24, and a[right] = 14. As a[pivot] > a[right], so, swap a[pivot]24 and a[right]14



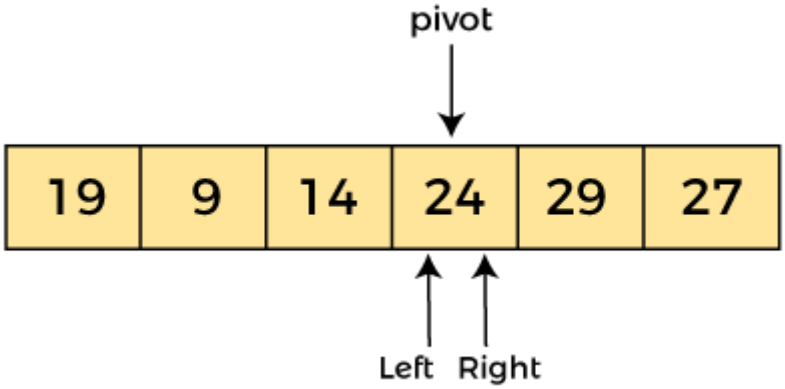
Step8:

- Now pivot is at right
- a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.

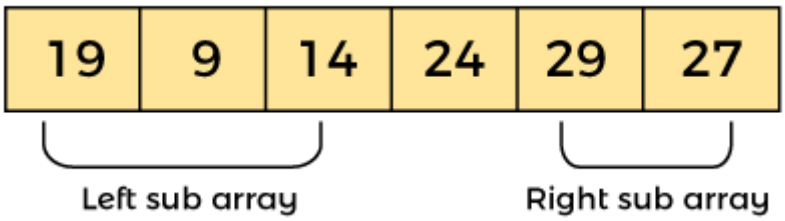


Step9:

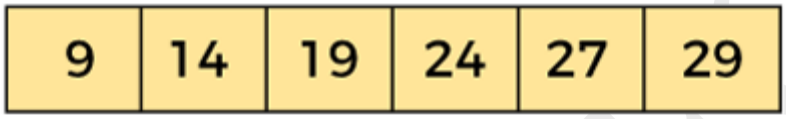
- Now, a[pivot] = 24, a[left] = 24, and a[right] = 24.
- So, pivot, left and right are pointing the same element. It represents the termination of procedure.



- Element 24, which is the pivot element is placed at its exact position.
- Elements that are right side of element 24 are greater than it, and the elements that are left side of element 24 are smaller than it.



Now, in a similar manner, quick sort algorithm is separately applied to the left and right sub-arrays. After sorting gets done, the array will be -



Algorithm for Quick Sort:

```
Algorithm QuickSort(a, low, high)

//   Purpose: Sort the elements of the array using quicksort

//   Inputs: low - The position of the first element of array a

//           high - The position of the last element of array a

              a   - An array consisting of unsorted elements

//   Output: a - An array consisting of sorted elements

if (low > high) then

    return

end if

k ← partition(a, low, high)

QuickSort(a, low, k - 1)

QuickSort(a, k + 1, high)


Algorithm partition(A, low, high)
```

```
//   Purpose: Divide the array into two parts such that elements to
     the left of the pivot element are <= to the pivot,and elements
     to the right of the pivot are >= to the pivot.
//   Inputs: low - The position of the first element of array A,
//           high - The position of the last element of array A,
//           a - An array consisting of unsorted elements
//   Output: a - Partitioned array such that elements to the left of
     the pivot are ≤ pivot and elements to the right are ≥ pivot
key ← A[low]
i ← low
j ← high + 1
while (i <= j)
     do i ← i + 1 while (key >= a[i])
     do j ← j - 1 while (key <a[j])
     if (i < j) exchange(a[i], a[j])
end while
exchange (a[low], a[j])
return j
```

Binary Search

The various searching techniques are linear search and Binary search. Linear search works on both sorted and unsorted list. But if the list is either in ascending or descending order, the search item can be drastically reduced by using binary search.

Definition: A binary search is a simple searching technique which can be applied if that item not be compared, or either in ascending order or descending order.

Conditions for when to apply Binary Search

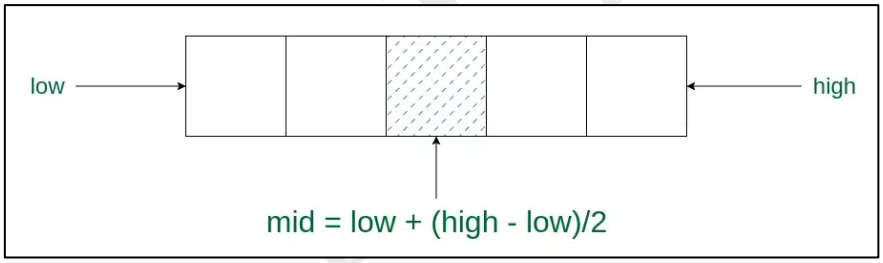
To apply Binary Search algorithm:

- The data structure must be sorted.
- Access to any element of the data structure takes constant time.

Working of Binary Search:

In this algorithm,

- Divide the search space into two halves by finding the middle index ***mid***



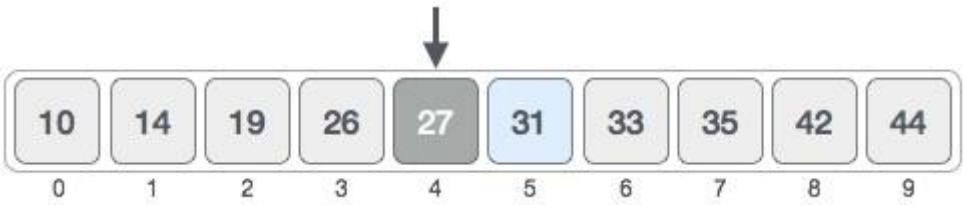
- Compare the middle element of the search space with the key.
- If the key is found at middle element, the process is terminated.
- If the key is not found at middle element, choose which half will be used as the next search space.
    - If the key is smaller than the middle element, then the left side is used for next search.
    - If the key is larger than the middle element, then the right side is used for next search.
- This process is continued until the key is found or the total search space is exhausted.

Example:

For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.

First, we shall determine half of the array by using this formula −

- mid = low + (high - low) / 2
- Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

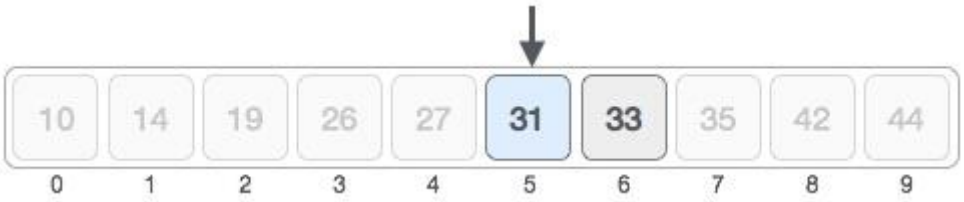
We change our low to mid + 1 and find the new mid value again.

- low = mid + 1
- mid = low + (high - low) / 2

Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.

Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.

We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

Algorithm for Binary Search:

```
BinarySearch(array, target):
// Purpose: Search for an item in the list a using binary search
// Input:
//         item – the element to be searched
//         a – the list of element where the searching takes place
//         n – the number of elements in the list
// Output: Returns the index of the target element if found, otherwise returns
-1.
low ← 0
high ← n-1
    while (low <= high)
        mid ← (left + right) / 2
        if (item = a[mid])
           return mid + 1
       if (item < array[mid])
```

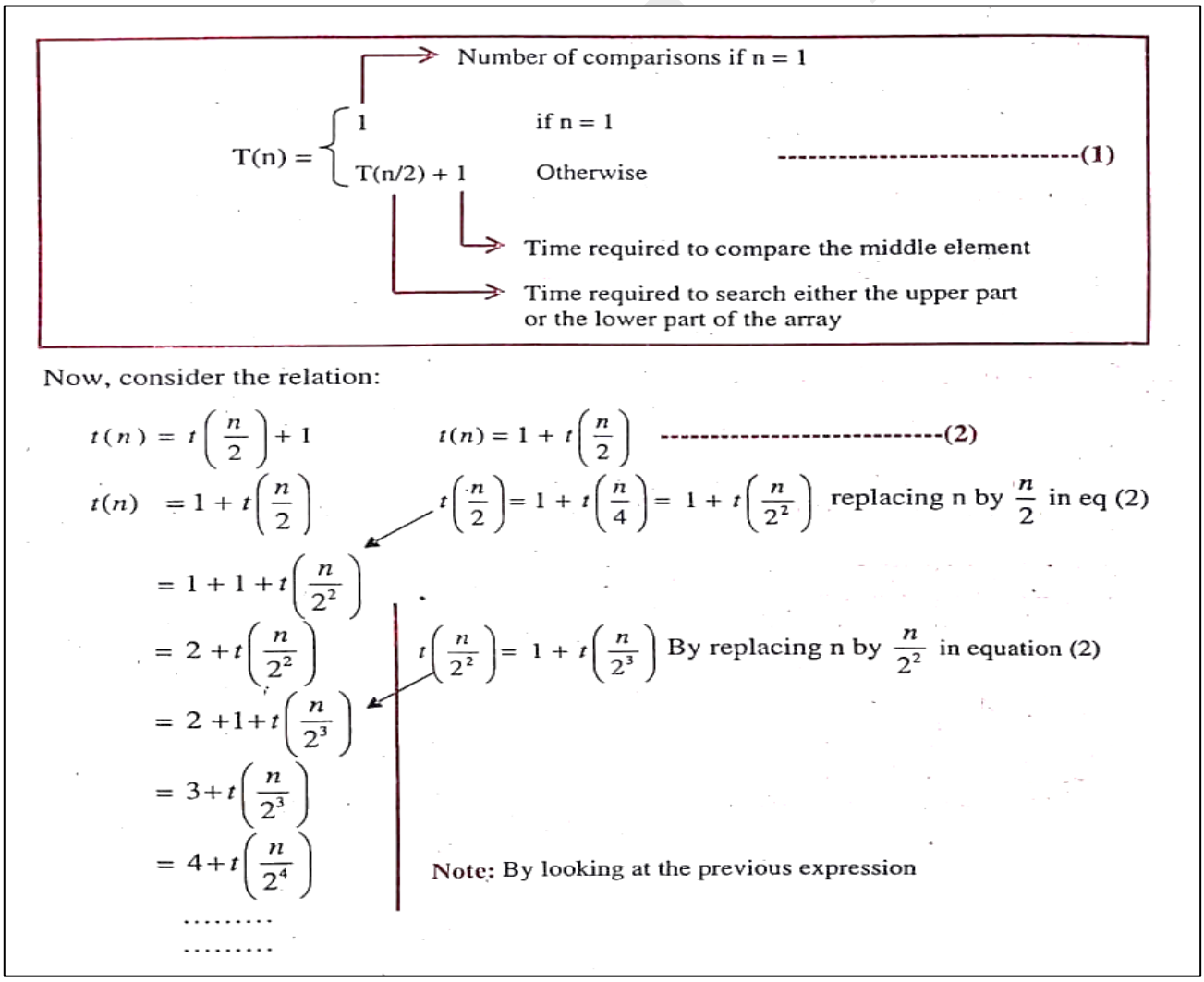
```
            high ← middle - 1
        else
            low ← middle + 1
      end if
end while
return-1
```

Analysis:

Analysis (best case): the best case occur when the item to be searched is present in the middle of the array. So, that the total number of comparison required will be 1.

So that time complexity of Binary search in the best case is given by Ω(1)

Analysis (Worst case): The worst case occurs when maximum number of element comparison are required and the time publicity is given by the following recurrence relation:



In general,

$$= i + t\left(\frac{n}{2^i}\right)$$

⋮ ⟶ Finally, to get initial condition t(1), let $2^i = n$ ----------(3)

$t(1) = 1$ from equation (1)

$$= i + t(1)$$

$t(n) \approx i$ ----------(4)

From equation (3) we have $2^i = n$. Taking log on both sides, we have

$$i * \log_2 2 = \log_2 n$$
$$i = \log_2 n$$

Substituting the value of i in equation (4) we have,

$$t(n) = \log_2 n$$

So, time complexity is given by $t(n) \in \theta(\log_2 n)$

Binary Tree traversals and related properties

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree

- Pre-order Traversal
- Post-order Traversal
- In-order Traversal

Preorder traversal

This technique follows the ***'root-left-right' policy***. It means that, first root node is visited after that the left subtree is traversed recursively, and finally, right subtree is recursively traversed. As the root node is traversed before (or pre) the left and right subtree, it is called preorder traversal.

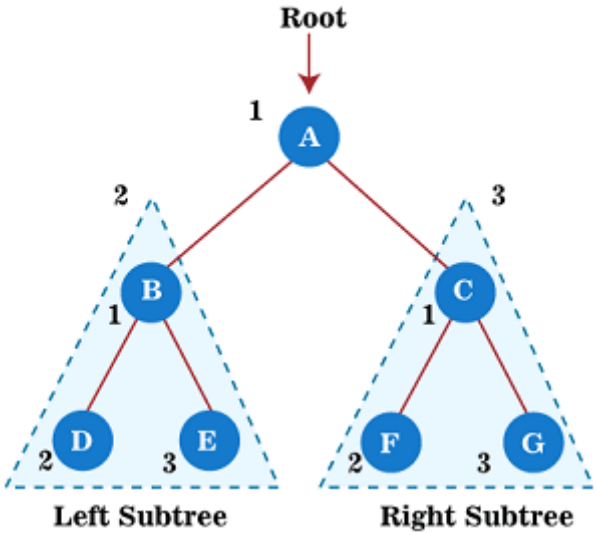
So, in a preorder traversal, each node is visited before both of its subtrees.

The applications of preorder traversal include –

- It is used to create a copy of the tree.
- It can also be used to get the prefix expression of an expression tree.

Example

Now, let's see the example of the preorder traversal technique.



- **Visit Node A (Root):** A

- **Traverse Left Subtree (Node B):**
  Visit Node B: A → B

- **Traverse Left Subtree (Node D):**
  - Visit Node D: A → B → D
  - Traverse Left Subtree (No left child for D)
  - Traverse Right Subtree (Node E):
    - Visit Node E: A → B → D → E
    - Traverse Left Subtree (No left child for E)
    - Traverse Right Subtree (No right child for D)
- **Traverse Right Subtree (Node C):**
  - Visit Node C: A → B → D → E → C
  - Traverse Left Subtree (Node F):
    - Visit Node F: A → B → D → E → C → F
    - Traverse Left Subtree (No left child for F)
    - Traverse Right Subtree (No right child for F)

- **Traverse Right Subtree (Node G):**
  - Visit Node G: A → B → D → E → C → F → G

- Traverse Left Subtree (No left child for G)
- Traverse Right Subtree (No right child for G)

So, the preorder traversal of the given tree is: $A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$.

Post order traversal

This technique follows the ***'left-right root' policy***. It means that the first left subtree of the root node is traversed, after that recursively traverses the right subtree, and finally, the root node is traversed. As the root node is traversed after (or post) the left and right subtree, it is called post order traversal.

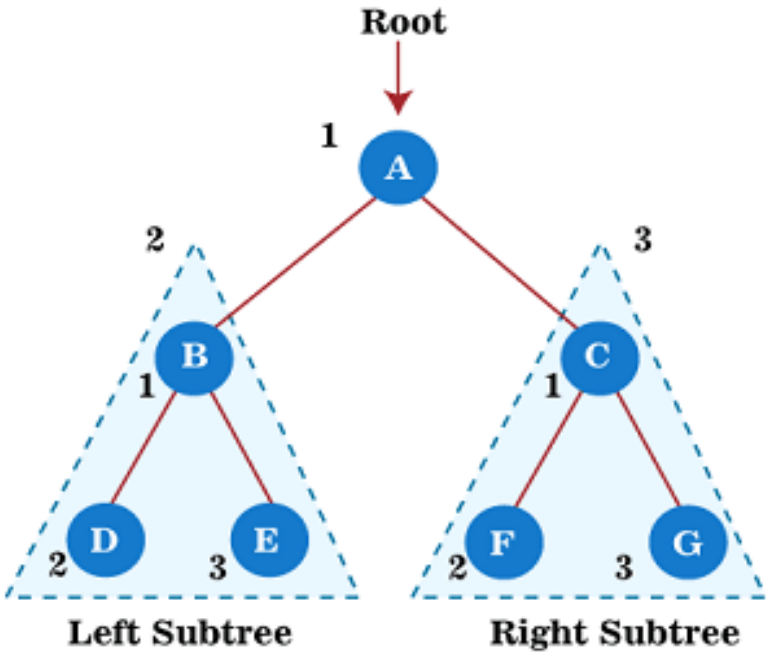
So, in a post order traversal, each node is visited after both of its subtrees.

The applications of post order traversal include -

- It is used to delete the tree.
- It can also be used to get the postfix expression of an expression tree.

Example:

Now, let's see the example of the post order traversal technique.



1. **Traverse Left Subtree (Node B):**
    - Traverse Left Subtree (Node D):
        - Traverse Left Subtree (No left child for D)
        - Traverse Right Subtree (No right child for D)
    - Traverse Right Subtree (Node E):
        - Traverse Left Subtree (No left child for E)
        - Traverse Right Subtree (No right child for E)
    - Visit Node B: D → E → B
2. **Traverse Right Subtree (Node C):**
    - Traverse Left Subtree (Node F):
        - Traverse Left Subtree (No left child for F)
        - Traverse Right Subtree (No right child for F)
    - Traverse Right Subtree (Node G):
        - Traverse Left Subtree (No left child for G)
        - Traverse Right Subtree (No right child for G)
    - Visit Node C: F → G → C
3. **Visit Root Node A:**
    - A

So, the post order traversal of the given tree is: ***D → E → B → F → G → C → A***.

In-order Traversal

This technique follows the ***'left root right' policy***. It means that first left subtree is visited after that root node is traversed, and finally, the right subtree is traversed. As the root node is traversed between the left and right subtree, it is named inorder traversal.

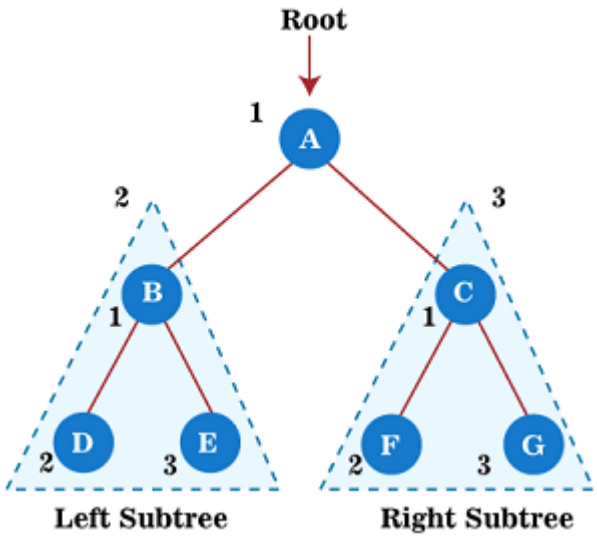
So, in the in order traversal, each node is visited in between of its subtrees.

The applications of In order traversal includes -

- It is used to get the BST nodes in increasing order.
- It can also be used to get the prefix expression of an expression tree.

Example

Now, let's see the example of the In order traversal technique.



1. **Traverse Left Subtree (Node B):**
    - Traverse Left Subtree (Node D):
        - Traverse Left Subtree (No left child for D)
        - Visit Node D: D
        - Traverse Right Subtree (No right child for D)
    - Visit Node B: D → B
    - Traverse Right Subtree (Node E):
        - Traverse Left Subtree (No left child for E)
        - Visit Node E: D → B → E
        - Traverse Right Subtree (No right child for E)
2. **Visit Root Node A:**
    - Visit Node A: D → B → E → A
3. **Traverse Right Subtree (Node C):**
    - Traverse Left Subtree (Node F):
        - Traverse Left Subtree (No left child for F)
        - Visit Node F: D → B → E → A → F
        - Traverse Right Subtree (No right child for F)

- Visit Node C: D → B → E → A → F → C
- Traverse Right Subtree (Node G):
    - Traverse Left Subtree (No left child for G)
    - Visit Node G: D → B → E → A → F → C → G
    - Traverse Right Subtree (No right child for G)

So, the in order traversal of the given tree is: ***D → B → E → A → F → C → G***

Properties of Binary Tree:

- The highest node in a tree that has no parent nodes is considered the root of the tree. Every tree has a single root node.
- **Parent Node**: A node's parent one is the node that came before it in the tree of nodes.
- **Child Node**: The node that is a node's direct successor is referred to as a node's child node.
- **Siblings** are the children of the same parent node.
- **Edge:** Edge serves as a connecting node between the parent and child nodes.
- **Leaf:** A node without children is referred to as a leaf node. It is the tree's last node. A tree may have several leaf nodes.
- A node's subtree is the tree that views that specific node as the root node.
- **Depth:** The depth of a node is the separation between it and the root node.
- **Height:** The height of a node is the distance between it and the subtree's deepest node.
- The maximum height of any node is referred to as the tree's height. The height of the root node is the same as this.
- **Level:** In the tree, a level is the number of parents that correspond to a particular node.
- **Node degree:** A node's degree is determined by how many children it has.

UNIT – 5

Greedy Technique: Introduction, Prim's Algorithm, Kruskal's Algorithm, Dijkstra's Algorithm, Lower-Bound Arguments, Decision Trees, P Problems, NP Problems, NP Complete Problems, Challenges of Numerical Algorithms.

Greedy Technique

Introduction

Among all the algorithmic approaches, the simplest and straightforward approach is the Greedy method. In this approach, the decision is taken on the basis of current available information without worrying about the effect of the current decision in future.

- Greedy algorithms build a solution step by step, selecting the next part for immediate benefit.
- Choices made are not reconsidered later in the process.
- Mainly used for solving optimization problems.
- Aims to find the best solution by making locally optimal choices at each step.
- Easy to implement and efficient in many cases.
- Follows a heuristic/ shortcut approach, focusing on immediate gains.
- Greedy algorithm is an algorithmic paradigm based on shortcut/ heuristics.
- It aims to find a global optimal solution by making locally optimal choices at each step.
- Often produces near-optimal solutions in a reasonable time.
- While not guaranteed to be optimal, it provides a practical solution to many problems.

Greedy approach is used to solve many problems, such as

- Finding the shortest path between two vertices using Dijkstra's algorithm.
- Finding the minimal spanning tree in a graph using Prim's /Kruskal's algorithm, etc.

Spanning Tree

Spanning tree is a tree in which all notes are connected without forming a closed path of circuit. Formally, a spanning tree of a graph G is defined as a sub graph $G^{|} = (V^{|},E^{|})$ With the following properties:

- $V^{|} = V$
- $G^{|}$ is Connected
- $G^{|}$ is a cyclic

Properties of a Spanning Tree:

- A Spanning tree does not exist for a disconnected graph.
- For a connected graph having N vertices then the number of edges in the spanning tree for that graph will be N-1.
- A Spanning tree does not have any cycle.
- We can construct a spanning tree for a complete graph by removing E-N+1 edges, where E is the number of Edges and N is the number of vertices.

For example, consider the graph shown below the various spanning trees of the graph are shown in the figure

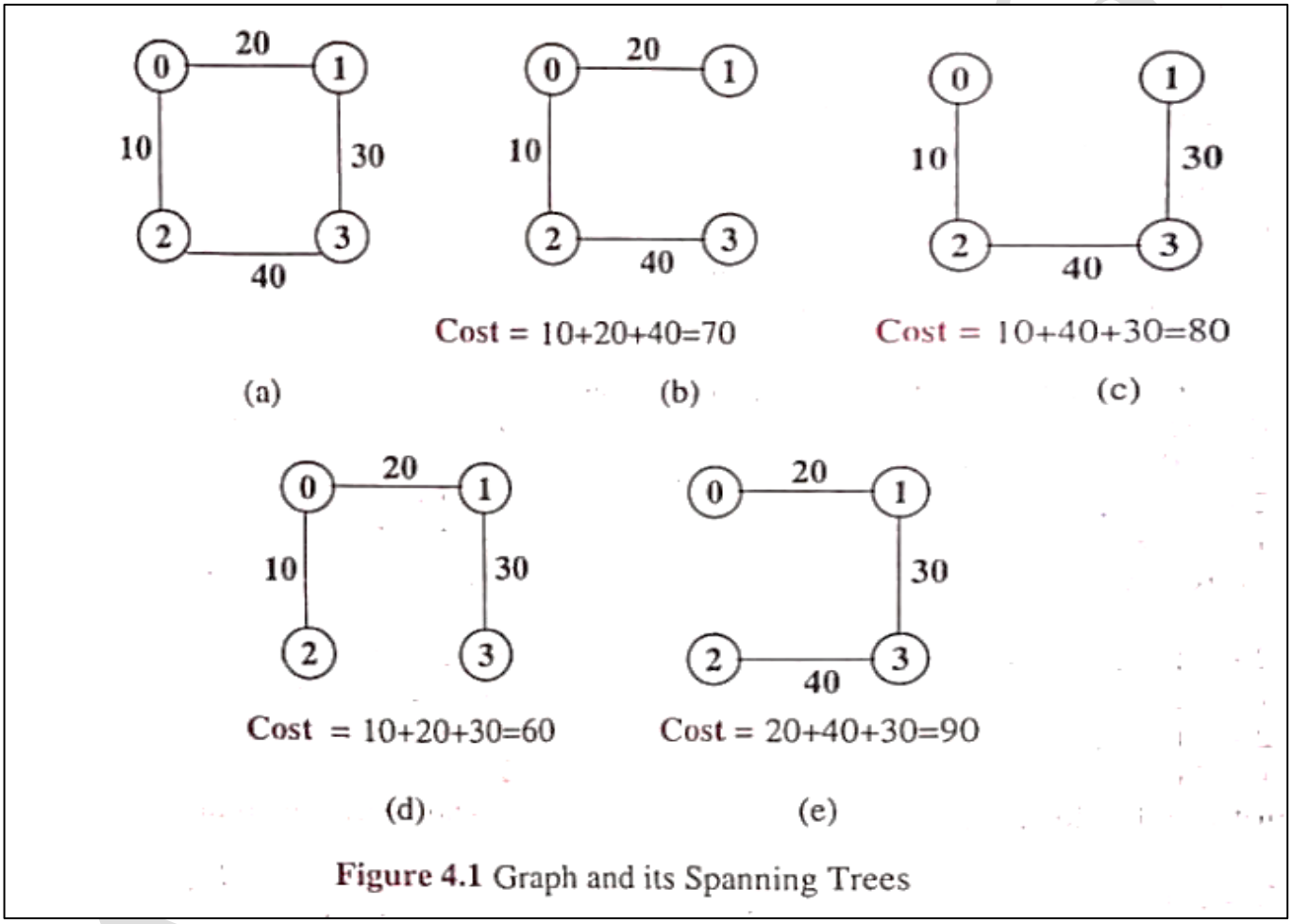


Figure 4.1 Graph and its Spanning Trees

Minimum Spanning Tree:

A ***minimum spanning tree (MST)*** is defined as a spanning tree that has the minimum weight among all the possible spanning trees.

For the graph shown in the figure, the minimum Spanning tree is shown in figure because the cost of that span increased minimum

Now, the question is Given a graph G, will it have only one minimum spanning tree or more than one spanning tree?" A graph will have only one minimum spanning tree if and only if the weights associated will all the edges in the graph are distinct. For example, the graph shown in figure a, there exists only one minimum spanning as shown in figure d.

But, if some of the edges have same weights, then the graph will have more than one minimum spanning tree For example, consider the graph shown in figure a. It has four minimum spanning trees whose cost remains same. The minimum spanning trees are shown in figures b to e.

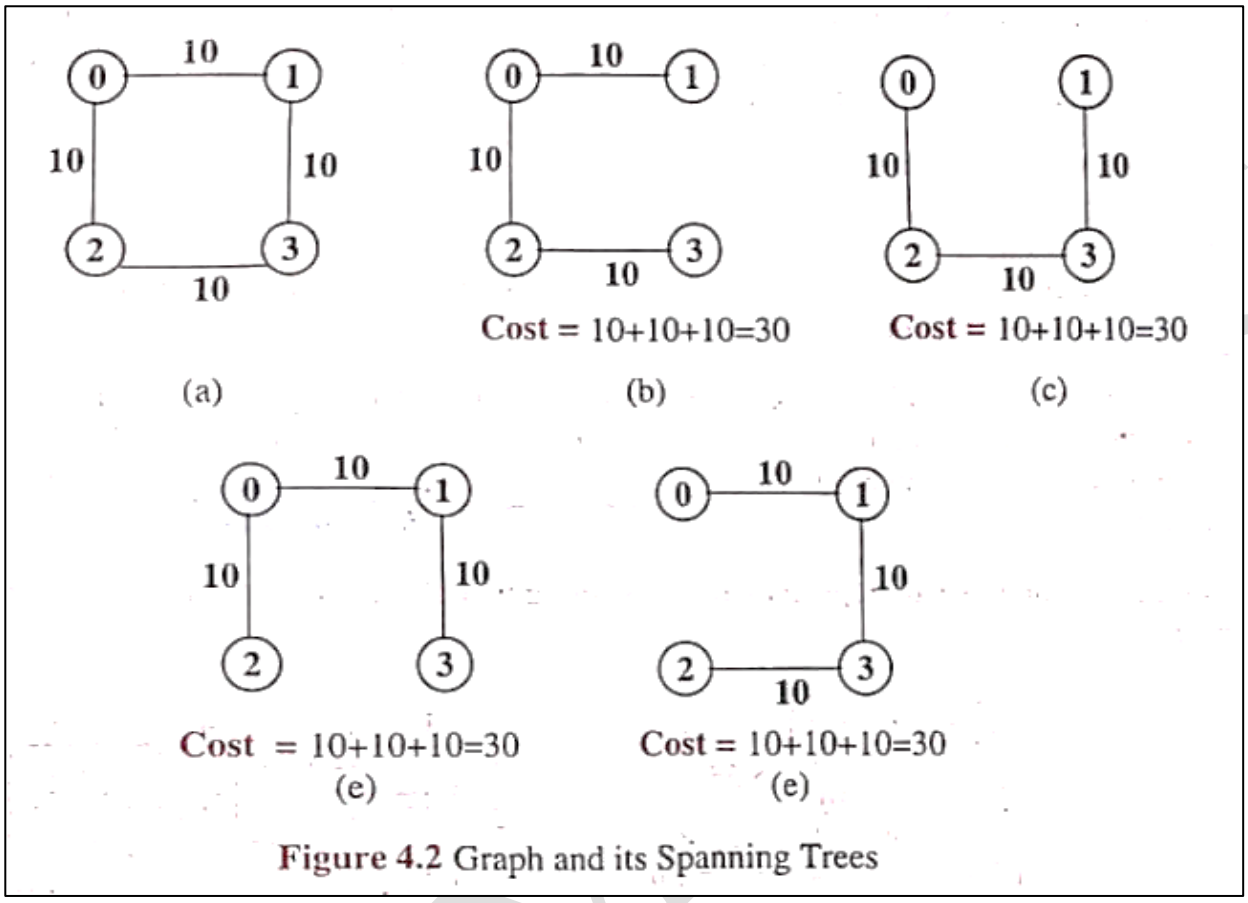


Figure 4.2 Graph and its Spanning Trees

Real World Applications of A Spanning Tree:

- Several path finding algorithms, such as Dijkstra's algorithm and A* search algorithm, internally build a spanning tree as an intermediate step.
- Building Telecommunication Network.
- Image Segmentation to break an image into distinguishable components.
- Computer Network Routing Protocol

Prim's Algorithm

Before starting the main topic, we should discuss the basic and important terms such as spanning tree and minimum spanning tree.

Spanning tree - A spanning tree is the subgraph of an undirected connected graph.

Minimum Spanning tree - Minimum spanning tree can be defined as the spanning tree in which the sum of the weights of the edge is minimum. The weight of the spanning tree is the sum of the weights given to the edges of the spanning tree.

Now, let's start the main topic.

Prim's Algorithm is a greedy algorithm that is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.

Prim's algorithm starts with the single node and explores all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

How does the prim's algorithm work?

Prim's algorithm is a greedy algorithm that starts from one vertex and continue to add the edges with the smallest weight until the goal is reached. The steps to implement the prim's algorithm are given as follows -

- First, we have to initialize an MST with the randomly chosen vertex.
- Now, we have to find all the edges that connect the tree in the above step with the new vertices. From the edges found, select the minimum edge and add it to the tree.
- Repeat step 2 until the minimum spanning tree is formed.

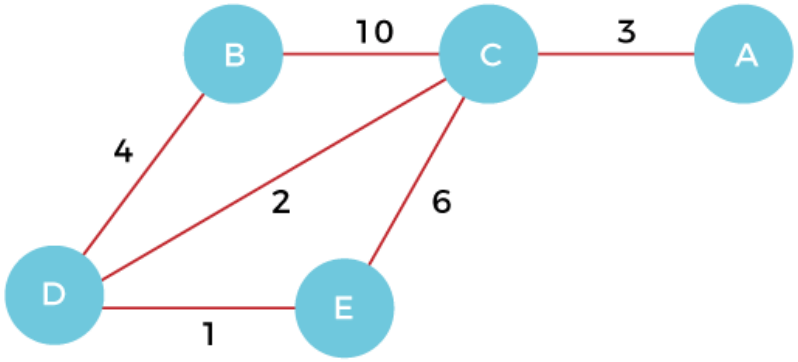
The applications of prim's algorithm are -

- Prim's algorithm can be used in network designing.
- It can be used to make network cycles.
- It can also be used to lay down electrical wiring cables.

Example of prim's algorithm

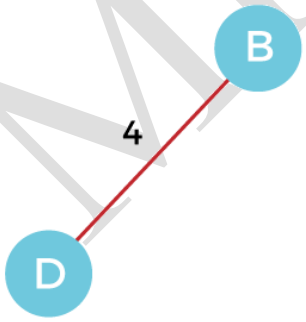
Now, let's see the working of prim's algorithm using an example. It will be easier to understand the prim's algorithm using an example.

Suppose, a weighted graph is –

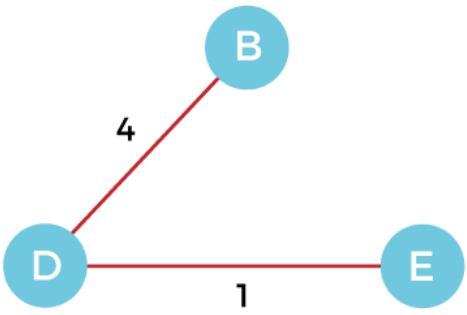


Step 1 - First, we have to choose a vertex from the above graph. Let's choose B.

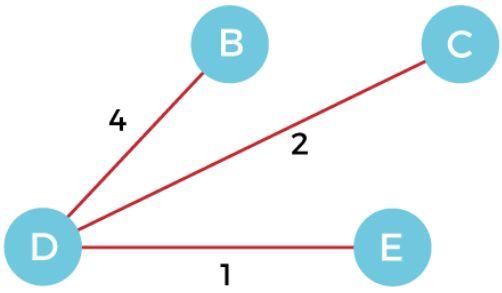
Step 2 - Now, we have to choose and add the shortest edge from vertex B. There are two edges from vertex B that are B to C with weight 10 and edge B to D with weight 4. Among the edges, the edge BD has the minimum weight. So, add it to the MST.



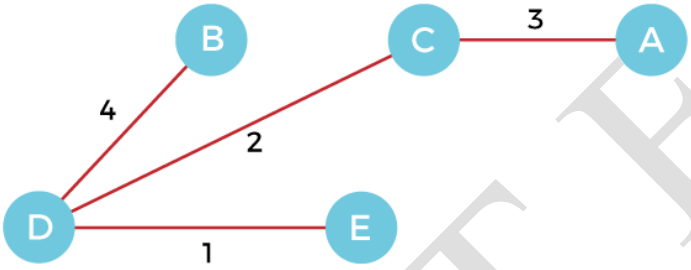
Step 3 - Now, again, choose the edge with the minimum weight among all the other edges. In this case, the edges DE and CD are such edges. Add them to MST and explore the adjacent of C, i.e., E and A. So, select the edge DE and add it to the MST.



Step 4 - Now, select the edge CD, and add it to the MST.



Step 5 - Now, choose the edge CA. Here, we cannot select the edge CE as it would create a cycle to the graph. So, choose the edge CA and add it to the MST.



So, the graph produced in step 5 is the minimum spanning tree of the given graph. The cost of the MST is given below -

Cost of MST = 4 + 2 + 1 + 3 = 10 units.

Kruskal's Algorithm

Here we will discuss **Kruskal's algorithm** to find the MST of a given weighted graph.

In Kruskal's algorithm, sort all edges of the given graph in increasing order. Then it keeps on adding new edges and nodes in the MST if the newly added edge does not form a cycle. It picks the minimum weighted edge at first and the maximum weighted edge at last. Thus we can say that it makes a locally optimal choice in each step in order to find the optimal solution. Hence this is a **Greedy Algorithm**.

How to find MST using Kruskal's algorithm?

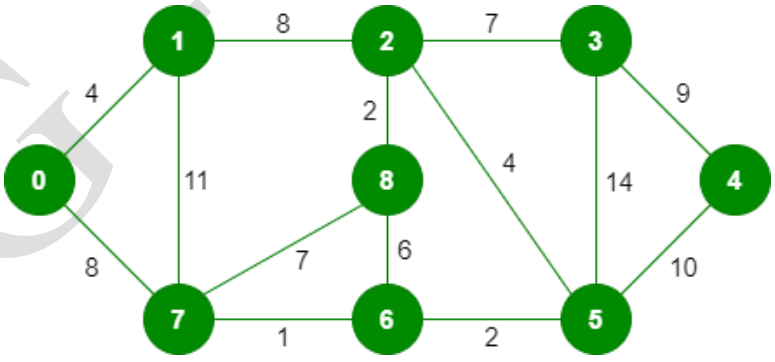
Below are the steps for finding MST using Kruskal's algorithm:

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If the cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are (V-1) edges in the spanning tree.

Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. The Greedy Choice is to pick the smallest weight edge that does not cause a cycle in the MST constructed so far. Let us understand it with an example:

Below is the illustration of the above approach:

Input Graph:



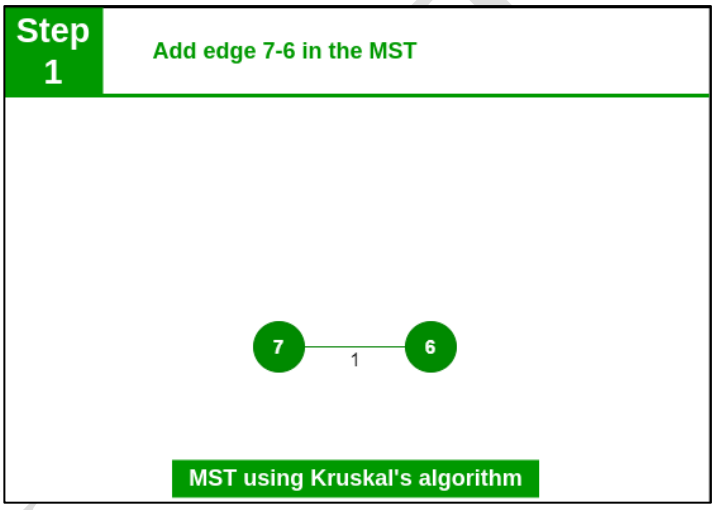
- The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having (9 – 1) = 8 edges.

After sorting:

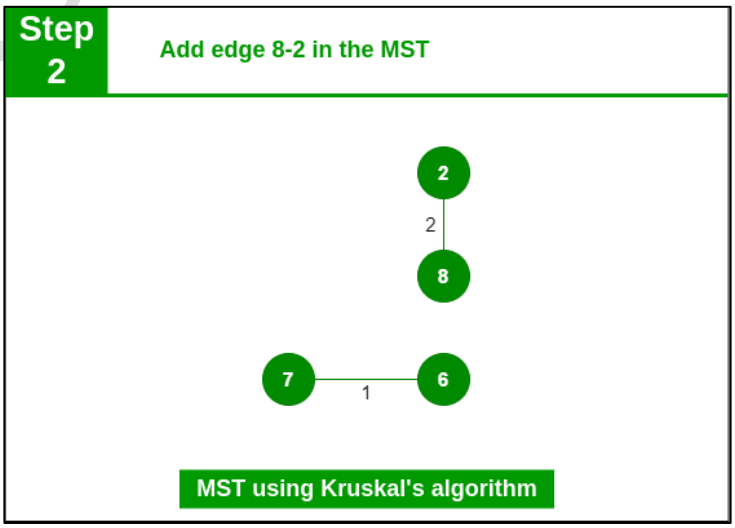
Weight	Source	Destination
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

Now pick all edges one by one from the sorted list of edges

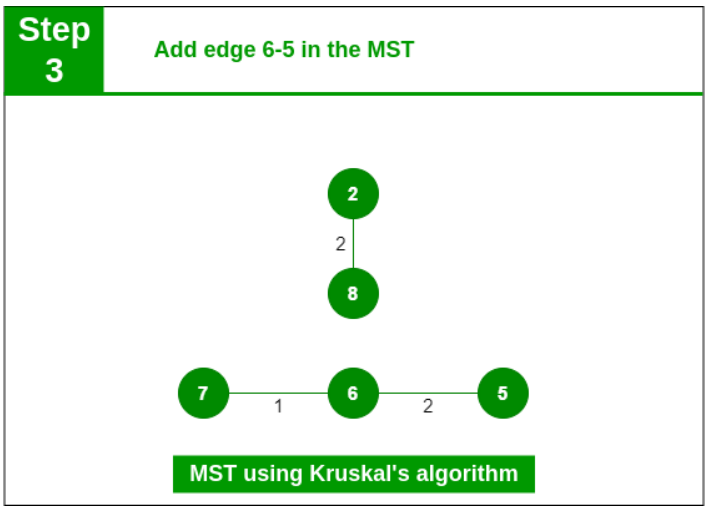
Step 1: Pick edge 7-6. No cycle is formed, include it.



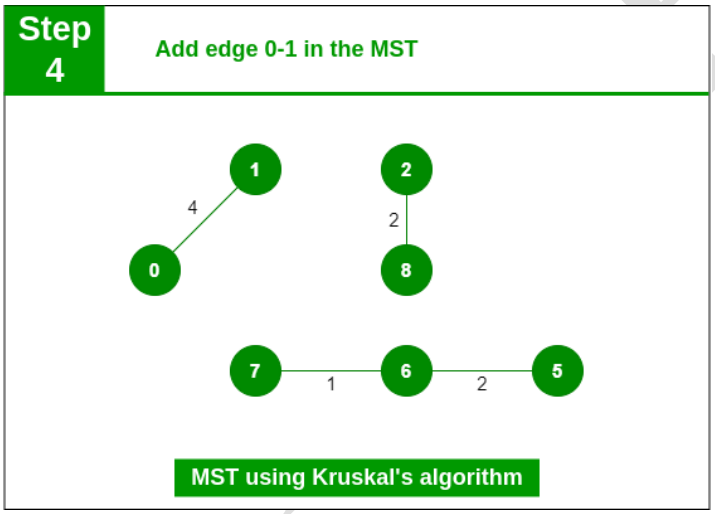
Step 2: Pick edge 8-2. No cycle is formed, include it.



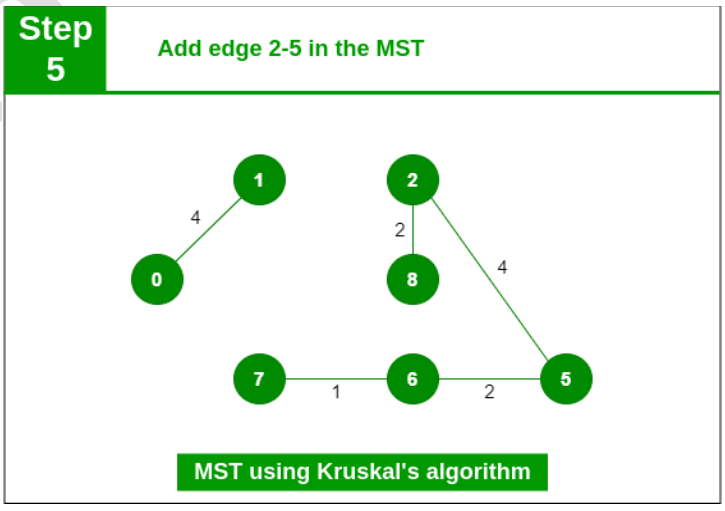
Step 3: Pick edge 6-5. No cycle is formed, include it.



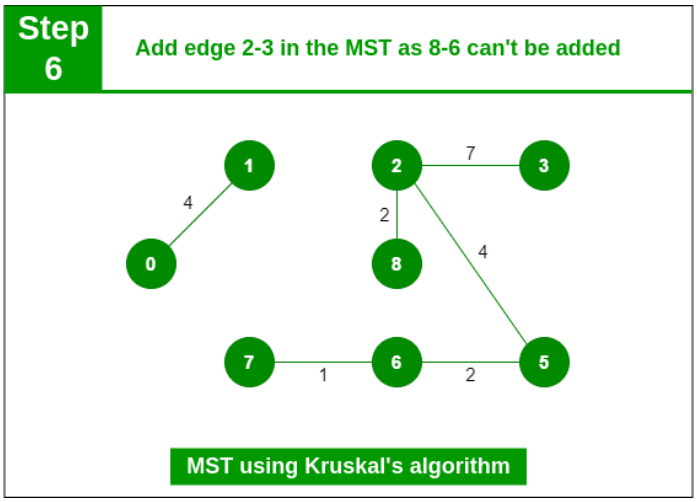
Step 4: Pick edge 0-1. No cycle is formed, include it.



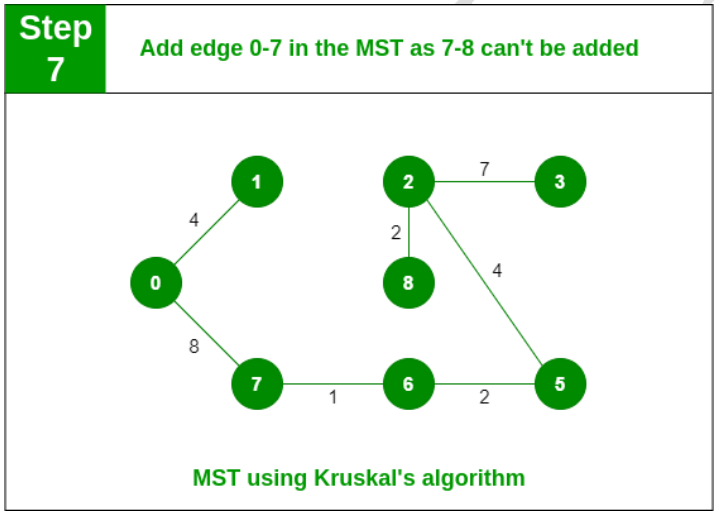
Step 5: Pick edge 2-5. No cycle is formed, include it.



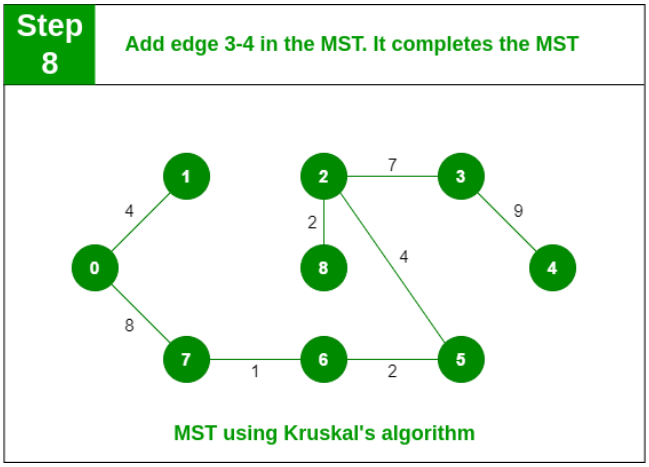
Step 6: Pick edge 8-6. Since including this edge results in the cycle, discard it. Pick edge 2-3: No cycle is formed, include it.



Step 7: Pick edge 7-8. Since including this edge results in the cycle, discard it. Pick edge 0-7. No cycle is formed, include it.



Step 8: Pick edge 1-2. Since including this edge results in the cycle, discard it. Pick edge 3-4. No cycle is formed, include it.



The cost of the spanning tree is the sum of the cost of the edges selected, which is given by:

$$4 + 8 + 1 + 2 + 4 + 2 + 7 + 9 = 37$$

Dijkstra's Algorithm

Definition: The singer sores shortest path problem is the one wherein we compute the shortest distance from a given source vertex V to all the other vertices in the graph.

In short, in single-source short path problem, we find the shortest distance from a given vertex to all other vertex. *This is possible using Dijkstra's algorithm.*

Understanding Dijkstra's Algorithm with an Example

The following is the step that we will follow to implement Dijkstra's Algorithm:

Step 1: First, we will mark the source node with a current distance of 0 and set the rest of the nodes to INFINITY.

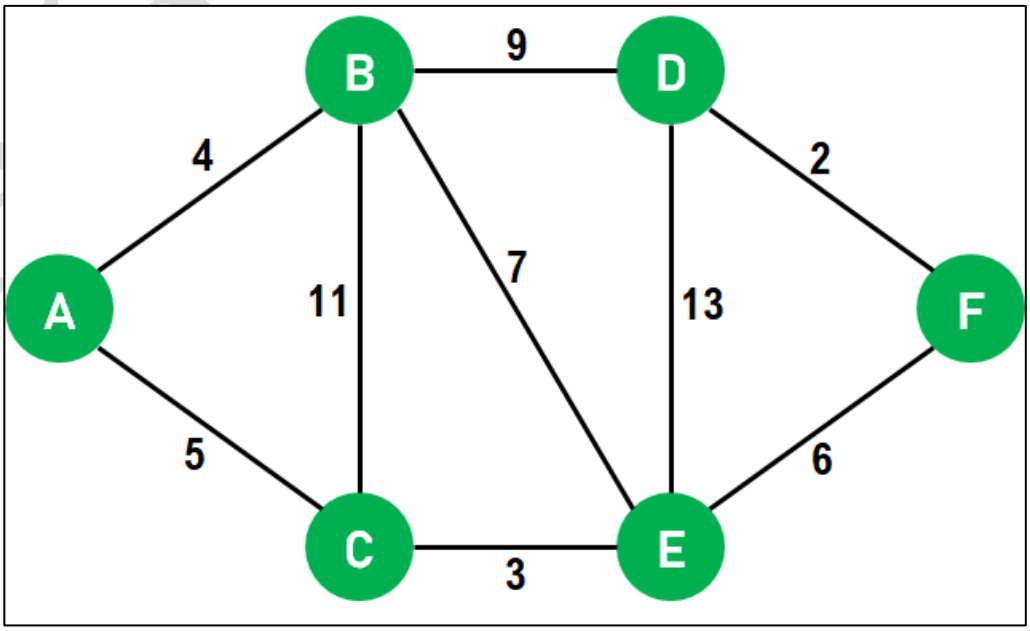
Step 2: We will then set the unvisited node with the smallest current distance as the current node, suppose X.

Step 3: For each neighbor N of the current node X: We will then add the current distance of X with the weight of the edge joining X-N. If it is smaller than the current distance of N, set it as the new current distance of N.

Step 4: We will then mark the current node X as visited.

Step 5: We will repeat the process from **'Step 2'** if there is any node unvisited left in the graph.

Let us now understand the implementation of the algorithm with the help of an example:



The Above Given Graph

1. We will use the above graph as the input, with node **A** as the source.
2. First, we will mark all the nodes as unvisited.
3. We will set the path to **0** at node **A** and **INFINITY** for all the other nodes.
4. We will now mark source node **A** as visited and access its neighboring nodes. **Note:** We have only accessed the neighboring nodes, not visited them.
5. We will now update the path to node **B** by **4** with the help of relaxation because the path to node **A** is **0** and the path from node **A** to **B** is **4**, and the **minimum((0 + 4), INFINITY)** is **4**.
6. We will also update the path to node **C** by **5** with the help of relaxation because the path to node **A** is **0** and the path from node **A** to **C** is **5**, and the **minimum((0 + 5), INFINITY)** is **5**. Both the neighbors of node **A** are now relaxed; therefore, we can move ahead.
7. We will now select the next unvisited node with the least path and visit it. Hence, we will visit node **B** and perform relaxation on its unvisited neighbors. After performing relaxation, the path to node **C** will remain **5**, whereas the path to node **E** will become **11**, and the path to node **D** will become **13**.
8. We will now visit node **E** and perform relaxation on its neighboring nodes **B, D**, and **F**. Since only node **F** is unvisited, it will be relaxed. Thus, the path to node **B** will remain as it is, i.e., **4**, the path to node **D** will also remain **13**, and the path to node **F** will become **14 (8 + 6)**.
9. Now we will visit node **D**, and only node **F** will be relaxed. However, the path to node **F** will remain unchanged, i.e., **14**.
10. Since only node **F** is remaining, we will visit it but not perform any relaxation as all its neighboring nodes are already visited.
11. Once all the nodes of the graphs are visited, the program will end.

Hence, the final paths we concluded are:

A = 0

B = 4 (A -> B)

C = 5 (A -> C)

D = 4 + 9 = 13 (A -> B -> D)

E = 5 + 3 = 8 (A -> C -> E)

F = 5 + 3 + 6 = 14 (A -> C -> E -> F)

Lower Bound Arguments:

- A lower bound refers to the minimum amount of time or space complexity required to solve a computational problem.
- Lower bound proofs establish theoretical limits on the best possible algorithm for a problem, independent of a computer's hardware.
- Proving good lower bounds is difficult. Common techniques used are contradiction, adversary arguments, information theory etc.
- Lower bounds prove the optimality of algorithms. If an algorithm meets the lower bound, we know there cannot be a faster algorithm.
- For example, comparison sorting algorithms have a lower bound of Ω(n log n). Merge sort and Heapsort match this, so we know these are asymptotically optimal comparison sorts.
- Lower bounds help rule out certain approaches to a problem. If an approach cannot beat the lower bound, we know not to pursue that path.
- Problems belong to complexity classes based on the best lower bounds. Problems with linear lower bounds are in P. Problems with exponential lower bounds are often NP-Complete.
- There are still many open problems in computer science without tight lower bounds. Proving lower bounds remains an active area of research in algorithms.

Lower bound proofs are an indispensable tool in DAA to formally establish the performance limits of algorithms for a given problem. Matching an algorithm to a lower bound proves its optimality.

Decision trees:

- A decision tree is a flowchart-like tree structure where each internal node represents a test or decision on an attribute value, each branch represents an outcome of the test, and each leaf node represents a class label or decision.
- It is a method for approaching the problem of supervised learning and classification. The goal is to create a model that predicts the value of a target variable based on several input variables.
- To build a decision tree, we start with the root node which contains the whole dataset.
- At each internal node, we split the data into subsets based on the most significant attribute/predictor. This splitting is done recursively on each derived subset until the leaf node is pure and contains data entries belonging to the same class.

- Decision trees can handle both categorical and continuous attributes as input variables.
- Some commonly used algorithms for building decision trees are ID3, C4.5 and CART.

Example1:

Let's say we want to classify a set of data points representing different fruits based on their attributes like color, shape, taste etc. We would start with the entire dataset at the root. We then split it based on the most significant predictor, say color, into red and not red subsets. Further splits can be done on each subset based on remaining attributes like shape until we reach leaf nodes with pure classes – apple, banana etc. The final decision tree model will help classify new datapoints**.**

Example2:

Suppose we want to build a model to predict whether a student gets admitted into a university based on their exam score and interview performance. We have historical data for previously admitted students with their scores and admission decision (yes/no).

The root node will consider the entire dataset. The algorithm will identify exam score as the attribute that best splits the data into purified branches/subsets.

For example, the data could be split into 2 branches:

- Exam score < 60

- Exam score >= 60

The left branch would be further split on interview performance into final admission classifications:

- Interview score < 80 -> Not Admitted

- Interview score >= 80 -> Admitted

The right branch is pure after the first split and leads to the leaf node:

- Exam score >=60 -> Admitted

To make a prediction on new student with Exam=65 and Interview=90, we start at the root and follow the branches/decisions appropriately until we reach a leaf node to get the prediction – Admitted.

The decision tree built on the training data is used to make predictions on new unseen data by following the branch decisions leading to a final classification. The rules are easy to interpret compared to other black-box models.

Key aspects are:

- Recursively splitting data into purer subsets

- Using historical decisions to create branch rules

- Making predictions by traversing the tree from root to leaf node

P, NP, and NP-complete problems

- P refers to the set of problems that can be solved in polynomial time by a deterministic Turing machine. These are problems for which an efficient algorithm exists that can produce a solution in polynomial time.

Example: Shortest path problem can be solved in polynomial time using Dijkstra's algorithm.

- NP refers to the set of problems that can be verified in polynomial time by a non-deterministic Turing machine.

Example: Subset sum problem – given a set of integers, determine if some subset sums to a given number. The solution can be verified quickly but finding it may take exponential time.

- NP-complete problems are the hardest problems in NP. No polynomial time solution is known for them.

Example: Traveling salesman problem of finding the shortest route visiting all cities exactly once.

- Problems in NP that are not NP-complete are called NP-hard problems. Their solution can be verified in polynomial time but does not necessarily lead to solutions for all problems in NP.

Example: Graph coloring problem of assigning colors to graph vertices so no two adjacent vertices share the same color.

- Showing a problem is NP-complete involves proving that it is in NP and also NP-hard. Typically done by reducing a known NP-complete problem to the problem being considered.

- Impact is that if any NP-complete problem can be solved in polynomial time, all can be. But no polynomial time solution known for any NP-complete problem yet.