

COURSE OBJECTIVES:

- Analyze the asymptotic performance of algorithms.
- Write rigorous correctness proofs for algorithms.
- Demonstrate a familiarity with major algorithms and data structures.
- Apply important algorithmic design paradigms and methods of analysis.
- Synthesize efficient algorithms in common engineering design situations.

COURSE OUTCOMES:

- For a given algorithms analyze worst-case running times of algorithms based on asymptotic analysis and justify the correctness of algorithms.
- Describe the greedy paradigm and explain when an algorithmic design situation calls for it. For a given problem develop the greedy algorithms.
- Describe the divide-and-conquer paradigm and explain when an algorithmic design situation calls for it. Synthesize divide-and-conquer algorithms. Derive and solve recurrence relation.
- Describe the dynamic-programming paradigm and explain when an algorithmic design situation calls for it. For a given problems of dynamic-programming and develop the dynamic programming algorithms, and analyze it to determine its computational complexity.
- For a given model engineering problem model it using graph and write the corresponding algorithm to solve the problems.
- Explain the ways to analyze randomized algorithms (expected running time, probability of error).
- Explain what an approximation algorithm is. Compute the approximation factor of an approximation algorithm (PTAS and FPTAS).

UNIT 1:

Introduction: Characteristics of algorithm. Analysis of algorithm: Asymptotic analysis of complexity bounds – best, average and worst-case behavior;

Performance measurements of

Algorithm, Time and space trade-offs, Analysis of recursive algorithms through recurrence relations: Substitution method, Recursion tree method and Masters' theorem.

UNIT 2:

Fundamental Algorithmic Strategies: Brute-Force, Greedy, Dynamic Programming, Branch- and-Bound and Backtracking methodologies for the design of

algorithms; Illustrations
of these techniques for Problem-Solving , Bin Packing, Knap Sack TSP. Heuristics
–
characteristics and their application domains.

UNIT 3:

Graph and Tree Algorithms: Traversal algorithms: Depth First Search (DFS) and Breadth First Search (BFS); Shortest path algorithms, Transitive closure, Minimum Spanning Tree, Topological sorting, Network Flow Algorithm.

UNIT 4:

Tractable and Intractable Problems: Computability of Algorithms, Computability classes – P, NP, NP-complete and NP-hard. Cook's theorem, Standard NP-complete problems and Reduction techniques.

UNIT 5:

Advanced Topics: Approximation algorithms, Randomized algorithms, Class of problems beyond NP – P SPACE

TEXT BOOKS:

1. Introduction to Algorithms, 4TH Edition, Thomas H Cormen, Charles E Lieserson, Ronald L Rivest and Clifford Stein, MIT Press/McGraw-Hill.
2. Fundamentals of Algorithms – E. Horowitz et al.

REFERENCES:

1. Algorithm Design, 1ST Edition, Jon Kleinberg and ÉvaTardos, Pearson.
2. Algorithm Design: Foundations, Analysis, and Internet Examples, Second Edition, Michael T Goodrich and Roberto Tamassia, Wiley.
3. Algorithms -- A Creative Approach, 3RD Edition, UdiManber, Addison-Wesley, Reading, MA.



KARPAGAM ACADEMY OF HIGHER EDUCATION

Faculty of Engineering

Department of Computer Science and Engineering

Lecture Plan

Subject Name: DESIGN AND ANALYSIS OF ALGORITHMS **Subject Code:1618BECS443**

S.No	Topic Name	No.of Periods	Supporting Materials	Teaching Aids
UNIT- I INTRODUCTION				
1	Introduction: Characteristics of algorithm.	1	R[1]-1	BB
2	Analysis of algorithm: Asymptotic analysis of complexity bounds	1	R[1]-1	BB
3	Best, average and worst-case behavior	1	R[1]-5	PPT
4	Performance measurements of Algorithm	1	R[1]-6	PPT
5	Time and space trade-offs	1	R[1]-6	PPT
6	Analysis of recursive algorithms	1	R[1]-95	PPT
7	Analysis of recursive algorithms through recurrence relations	1	R[1]-95	PPT
8	Substitution method	1	R[1]-68	BB
9	Recursion tree method	1	Web	PPT
10	Masters' theorem	1	R[1]-12	BB
11	Tutorial : Analysis of Algorithm	1	Web	BB
Total		11		
UNIT- II FUNDAMENTAL ALGORITHMIC STRATEGIES				
12	Brute-Force methodologies	1	R[1]-200	PPT
13	Greedy methodologies	1	web	PPT
14	Dynamic Programming methodologies	1	R[1] 201	BB
15	Branch- and-Bound methodologies	1	R[1]214	PPT
16	Backtracking methodologies	1	R[1]214	PPT
17	Illustrations of these techniques for Problem-Solving	1	R[1]218	PPT
18	Bin Packing	1	R[1]218	PPT
19	Knap Sack TSP	1	R[1]218	PPT
20	Heuristics – characteristics and their application domains	1	R[1]221	BB
21	Tutorial : Algorithm Strategy	1	R[1]221	PPT
Total		10		

	UNIT- III GRAPH AND TREE ALGORITHMS			
22	Graph and Tree Algorithms	1	web	PPT
23	Traversal algorithms	1	web	PPT
24	Depth First Search (DFS)	1	web	PPT
25	Breadth First Search	1	T[1]-488	BB
26	Shortest path algorithms	1	T[1]-193	PPT
27	Transitive closure	1	T[1]-266	BB
28	Minimum Spanning Tree	1	T[1]-305	PPT
29	Topological sorting	1	T[1]-343	BB
30	Network Flow Algorithm	1	web	PPT
31	Tutorial: Real-time application	1	web	PPT
	Total	10		
	UNIT- IV TRACTABLE AND INTRACTABLE PROBLEMS			
32	Computability of Algorithms	1	R[1]-139	PPT
33	Computability classes	1	R[1]-139	PPT
34	P-Class	1	T[1]-140	PPT
35	NP-Class	1	R[1]-152	BB
36	NP-complete	1	R[1]-159	PPT
37	NP-hard	1	R[1]-162	BB
38	Cook's theorem	1	R[1]-163	PPT
39	Standard NP-complete problems	1	R[1]-133	PPT
40	Reduction techniques	1	web	PPT
41	Tutorial : List of problems in NP hard	1	R[1]-133	BB
	Total	10		
	UNIT- V ADVANCED TOPICS			
42	Approximation algorithms	1	R[1]-248	PPT
43	Approximation algorithms: Problems	1	R[1]-465	BB
44	Class of problems	1	R[1]-465	BB
45	Class of problems :Example Problems	1	R[1]-255	PPT
46	Class of problems: Example Problems	1	R[1]-248	PPT
47	NP SPACE	1	T[1]-1087	PPT
48	beyond NP SPACE	1	T[1]-1087	PPT
49	P SPACE	1	T[1]-690	BB
50	Tutorial: Approximation algorithm	1	T[1]-690	PPT
51	Revisions	1	T[1]-752	BB
52	Discussion on Previous University Question Papers			
	Total	10		
	Total Hours	52		

TEXT BOOKS

S.NO	Title of the book			Year of publication
1	Introduction to Algorithms, 4TH Edition, Thomas H Cormen, Charles E Lieserson, Ronald L Rivest and Clifford Stein, MIT Press/McGraw-Hill.			2012
2	Fundamentals of Algorithms – E. Horowitz et al.			2013

REFERNCE BOOKS

S.NO	Title of the book			Year of publication
1	Algorithm Design, 1ST Edition, Jon Kleinberg and ÉvaTardos, Pearson			2011
2	Algorithm Design: Foundations, Analysis, and Internet Examples, Second Edition, Michael T Goodrich and Roberto Tamassia, Wiley.			2012
3	Algorithms -- A Creative Approach, 3RD Edition, UdiManber, Addison-Wesley, Reading, MA.			2011

WEBSITES

1. <https://www.javatpoint.com/daa-tutorial>
2. https://nptel.ac.in/content/syllabus_pdf/106106131.pdf

UNIT-1

1. Characteristics of an Algorithm

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

- **Unambiguous** – Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
- **Input** – An algorithm should have 0 or more well-defined inputs.
- **Output** – An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- **Finiteness** – Algorithms must terminate after a finite number of steps.
- **Feasibility** – Should be feasible with the available resources.
- **Independent** – An algorithm should have step-by-step directions, which should be independent of any programming code.

2. Analysis of Algorithm

Asymptotic analysis of an algorithm refers to defining the mathematical boundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.

Asymptotic analysis is input bound i.e., if there's no input to the algorithm, it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.

Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, the running time of one operation is computed as $f(n)$ and may be for another operation it is computed as $g(n^2)$. This means the first operation running time will increase linearly with the increase in n and the running time of the second operation will increase exponentially when n increases. Similarly, the running time of both operations will be nearly the same if n is significantly small.

Usually, the time required by an algorithm falls under three types –

- **Best Case** – Minimum time required for program execution.
- **Average Case** – Average time required for program execution.
- **Worst Case** – Maximum time required for program execution.

3. Asymptotic Notations

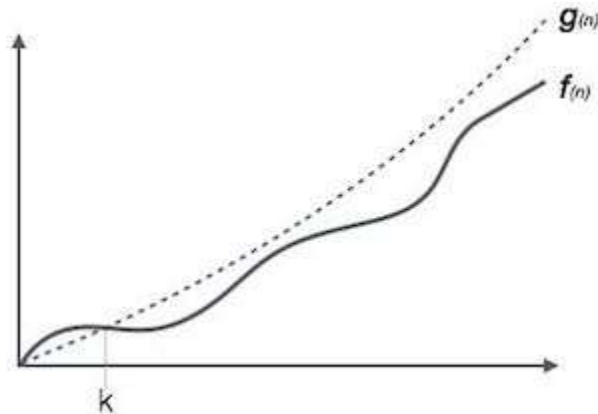
Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- O Notation

- Ω Notation
- θ Notation

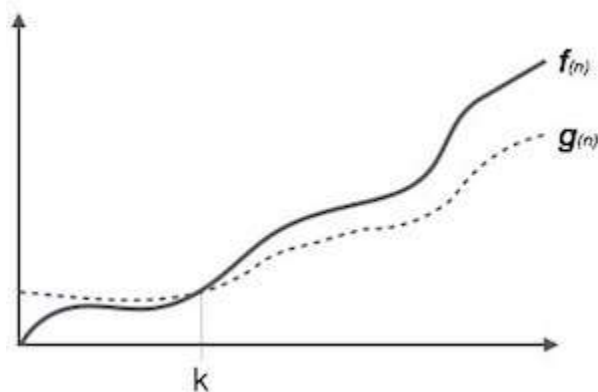
Big Oh Notation, O

The notation $O(n)$ is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.



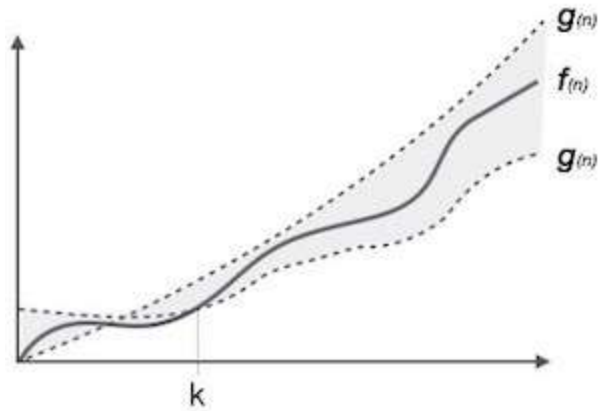
Omega Notation, Ω

The notation $\Omega(n)$ is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.



Theta Notation, θ

The notation $\theta(n)$ is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows –



Common Asymptotic Notations

Following is a list of some common asymptotic notations –

constant	–	$O(1)$
logarithmic	–	$O(\log n)$
linear	–	$O(n)$
$n \log n$	–	$O(n \log n)$
quadratic	–	$O(n^2)$
cubic	–	$O(n^3)$
polynomial	–	$n^{O(1)}$
exponential	–	$2^{O(n)}$

4. Fundamentals of Algorithmic problem solving

- Understanding the problem
- Ascertain the capabilities of the computational device

- Exact /approximate soln.
- Decide on the appropriate data structure
- Algorithm design techniques
- Methods of specifying an algorithm
- Proving an algorithms correctness
- Analysing an algorithm

Understanding the problem: The problem given should be understood completely. Check if it is similar to some standard problems & if a Known algorithm exists. otherwise a new algorithm has to be devised. Creating an algorithm is an art which may never be fully automated. An important step in the design is to specify an in- stance of the problem.

Ascertain the capabilities of the computational device: Once a problem is understood we need to Know the capabilities of the computing device this can be done by Knowing the type of the architecture, speed & memory availability.

Exact /approximate soln.: Once algorithm is devised, it is necessary to show that it computes answer for all the possible legal inputs. The solution is stated in two forms, Exact solution or approximate solution. examples of problems where an exact solution cannot be obtained are

i) Finding a squareroot of number.

ii) Solutions of non linear equations.

Decide on the appropriate data structure: Some algorithms do not demand any in- genuity in representing their inputs. Some others are in fact are predicted on ingenious data structures. A *data type* is a well-defined collection of data with a well-defined set of operations on it. A *data structure* is an actual implementation of a particular abstract data type. The Elementary Data Structures are Arrays These let you access lots of data fast. (good) . You can have arrays of *any* other da ta type. (good) . However, you cannot make arrays bigger if your program decides it needs more space. (bad) .

Records These let you organize non-homogeneous data into logical packages to keep every- thing together. (good) . These packages do not include operations, just data fields (bad, which is why we need objects) . Records do not help you process distinct items in loops (bad, which is why arrays of records are used) **Sets** These let you represent subsets of a set with such operations as intersection, union, and equivalence. (good) . Built-in sets are limited to a certain small size. (bad, but we can build our own *set data type* out of arrays to solve this problem if necessary)

Algorithm design techniques: Creating an algorithm is an art which may never be fully automated. By mastering these design strategies, it will become easier for you to devise new and useful algorithms. Dynamic programming is one such technique. Some of the techniques are especially useful in fields other then computer science such as operation research and electric- al engineering. Some important design techniques are linear, non linear and integer programming

Methods of specifying an algorithm: There are mainly two options for specifying an algorithm: use of natural language or pseudocode & Flowcharts.

A Pseudo code is a mixture of natural language & programming language like constructs. A flowchart is a method of expressing an algorithm by a collection of connected geometric shapes.

Proving an algorithms correctness: Once algorithm is devised, it is necessary to show that it computes answer for all the possible legal inputs .We refer to this process as algorithm validation. The process of validation is to assure us that this algorithm will work correctly independent of issues concerning programming language it will be written in. A proof of correctness requires that the solution be stated in two forms. One form is usually as a program which is annotated by a set of assertions about the input and output variables of a program. These assertions are often expressed in the predicate calculus. The second form is called a specification, and this may also be expressed in the predicate calculus. A proof consists of showing that these two forms are equivalent in that for every given legal input, they describe same out- put. A complete proof of program correctness requires that each statement of programming language be precisely defined and all basic operations be proved correct. All these details may cause proof to be very much longer than the program.

5. (a). Time Complexity of Algorithms

For any defined problem, there can be N number of solution. This is true in general. If I have a problem and I discuss about the problem with all of my friends, they will all suggest me different solutions. And I am the one who has to decide which solution is the best based on the circumstances.

Similarly for any problem which must be solved using a program, there can be infinite number of solutions. Let's take a simple example to understand this. Below we have two different algorithms to find square of a number(for some time, forget that square of any number n is $n*n$):

One solution to this problem can be, running a loop for n times, starting with the number n and adding n to it, every time.

```
/*  
    we have to calculate the square of n  
*/  
for i=1 to n  
    do n = n + n  
// when the loop ends n will hold its square  
return n
```

Or, we can simply use a mathematical operator $*$ to find the square.

```
/*  
    we have to calculate the square of n  
*/
```

```
return n*n
```

In the above two simple algorithms, you saw how a single problem can have many solutions. While the first solution required a loop which will execute for n number of times, the second solution used a mathematical operator $*$ to return the result in one line. So which one is the better approach, of course the second one.

What is Time Complexity?

Time complexity of an algorithm signifies the total time required by the program to run till its completion.

The time complexity of algorithms is most commonly expressed using the **big O notation**. It's an asymptotic notation to represent the time complexity. We will study about it in detail in the next tutorial.

Time Complexity is most commonly estimated by counting the number of elementary steps performed by any algorithm to finish execution. Like in the example above, for the first code the loop will run n number of times, so the time complexity will be n atleast and as the value of n will increase the time taken will also increase. While for the second code, time complexity is constant, because it will never be dependent on the value of n , it will always give the result in 1 step.

And since the algorithm's performance may vary with different types of input data, hence for an algorithm we usually use the **worst-case Time complexity** of an algorithm because that is the maximum time taken for any input size.

Calculating Time Complexity

Now let's tap onto the next big topic related to Time complexity, which is How to Calculate Time Complexity. It becomes very confusing some times, but we will try to explain it in the simplest way.

Now the most common metric for calculating time complexity is Big O notation. This removes all constant factors so that the running time can be estimated in relation to N , as N approaches infinity. In general you can think of it like this :

```
statement;
```

Above we have a single statement. Its Time Complexity will be **Constant**. The running time of the statement will not change in relation to N .

```
for(i=0; i < N; i++)  
{  
    statement;
```

```
}
```

The time complexity for the above algorithm will be **Linear**. The running time of the loop is directly proportional to N. When N doubles, so does the running time.

```
for(i=0; i < N; i++)  
{  
    for(j=0; j < N; j++)  
    {  
        statement;  
    }  
}
```

This time, the time complexity for the above code will be **Quadratic**. The running time of the two loops is proportional to the square of N. When N doubles, the running time increases by N * N.

(b). Space Complexity of Algorithms

Whenever a solution to a problem is written some memory is required to complete. For any algorithm memory may be used for the following:

1. Variables (This include the constant values, temporary values)
2. Program Instruction
3. Execution

Space complexity is the amount of memory used by the algorithm (including the input values to the algorithm) to execute and produce the result.

Sometime **Auxiliary Space** is confused with Space Complexity. But Auxiliary Space is the extra space or the temporary space used by the algorithm during it's execution.

Space Complexity = Auxiliary Space + Input space

Memory Usage while Execution

While executing, algorithm uses memory space for three reasons:

1. **Instruction Space**

It's the amount of memory used to save the compiled version of instructions.

2. Environmental Stack

Sometimes an algorithm(function) may be called inside another algorithm(function). In such a situation, the current variables are pushed onto the system stack, where they wait for further execution and then the call to the inside algorithm(function) is made.

For example, If a function **A()** calls function **B()** inside it, then all the variables of the function **A()** will get stored on the system stack temporarily, while the function **B()** is called and executed inside the function **A()**.

3. Data Space

Amount of space used by the variables and constants.

But while calculating the **Space Complexity** of any algorithm, we usually consider only **Data Space** and we neglect the **Instruction Space** and **Environmental Stack**.

Calculating the Space Complexity

For calculating the space complexity, we need to know the value of memory used by different type of datatype variables, which generally varies for different operating systems, but the method for calculating the space complexity remains the same.

Type	Size
bool, char, unsigned char, signed char, __int8	1 byte
__int16, short, unsigned short, wchar_t, __wchar_t	2 bytes
float, __int32, int, unsigned int, long, unsigned long	4 bytes
double, __int64, long double, long long	8 bytes

Now let's learn how to compute space complexity by taking a few examples:

```
{  
    int z = a + b + c;  
    return(z);  
}
```

In the above expression, variables **a**, **b**, **c** and **z** are all integer types, hence they will take up 4 bytes each, so total memory requirement will be $(4(4) + 4) = 20$ bytes, this additional 4 bytes is for **return value**. And because this space requirement is fixed for the above example, hence it is called **Constant Space Complexity**.

Let's have another example, this time a bit complex one,

```
// n is the length of array a[]  
int sum(int a[], int n)  
{  
    int x = 0;           // 4 bytes for x  
    for(int i = 0; i < n; i++) // 4 bytes for i  
    {  
        x = x + a[i];  
    }  
    return(x);  
}
```

- In the above code, $4*n$ bytes of space is required for the array **a[]** elements.
- 4 bytes each for **x**, **n**, **i** and the return value.

Hence the total memory requirement will be $(4n + 12)$, which is increasing linearly with the increase in the input value **n**, hence it is called as **Linear Space Complexity**.

Similarly, we can have quadratic and other complex space complexity as well, as the complexity of an algorithm increases.

But we should always focus on writing algorithm code in such a way that we keep the space complexity **minimum**.

UNIT-II

1. Brute Force Algorithm

The brute force algorithm consists in checking, at all positions in the text between 0 and $n-m$, whether an occurrence of the pattern starts there or not. Then, after each attempt, it shifts the pattern by exactly one position to the right.

The brute force algorithm requires no preprocessing phase, and a constant extra space in addition to the pattern and the text. During the searching phase the text character comparisons can be done in any order.

The time complexity of this searching phase is $O(mn)$ (when searching for $a^{m-1}b$ in a^n for instance). The expected number of text character comparisons is $2n$.

Main Features

- no preprocessing phase
- constant extra space needed
- always shifts the window by exactly 1 position to the right
- comparisons can be done in any order
- searching phase in $O(mn)$ time complexity
- $2n$ expected text characters comparisons

EXAMPLE

Bubble sort is one of the simple sorting algorithms and also popularly known as a **Brute Force Approach**. The logic of the algorithm is very simple as it works by repeatedly iterating through a list of elements, comparing two elements at a time and swapping them if necessary until all the elements are swapped to an order.

For e.g. if we have a list of 10 elements, bubble sort starts by comparing the first two elements in the list. If the second element is smaller than the first element then it exchanges them. Then it compares the current second element with the third element in the list. This continues until the second last and the last element is compared which completes one iteration through the list. By the time it completes the first iteration the largest element in the list comes to the rightmost position.

The algorithm gets its name as we start from lowest point and “bubble up” the higher elements to the highest point in the list. We can also follow other approach where we start from highest point and “bubble down” lowest elements in the list. Since it only uses comparisons to operate on elements, it is a comparison sort.

05	01	04	02	08
----	----	----	----	----

First Iteration

05	01	04	02	08
----	----	----	----	----

Compare 5 with 1 and $5 > 1$

01	05	04	02	08
----	----	----	----	----

Swap 1 and 5

01	05	04	02	08
----	----	----	----	----

Compare 5 with 4 and $5 > 4$

01	04	05	02	08
----	----	----	----	----

Swap 4 and 5

01	04	05	02	08
----	----	----	----	----

Compare 5 with 2 and $5 > 2$

01	04	02	05	08
----	----	----	----	----

Swap 2 and 5

01	04	02	05	08
----	----	----	----	----

Compare 5 with 8 and $5 < 8$

Second Iteration

01	04	02	05	08
----	----	----	----	----

Compare 1 with 4 and $1 < 4$

01	04	02	05	08
----	----	----	----	----

Compare 4 with 2 and $4 > 2$

01	02	04	05	08
----	----	----	----	----

Swap 2 and 4

01	02	04	05	08
----	----	----	----	----

Compare 4 with 5 and $4 < 5$

01	02	04	05	08
----	----	----	----	----

Compare 5 with 8 and $5 < 8$

Third Iteration

01	02	04	05	08
----	----	----	----	----

Compare 1 with 2 and $1 < 2$

01	02	04	05	08
----	----	----	----	----

Compare 2 with 4 and $2 < 4$

01	02	04	05	08
----	----	----	----	----

Compare 4 with 5 and $4 < 5$

01	02	04	05	08
----	----	----	----	----

Compare 5 with 8 and $5 < 8$

Fourth Iteration

01	02	04	05	08	Compare 1 with 2 and $1 < 2$
01	02	04	05	08	Compare 2 with 4 and $2 < 4$
01	02	04	05	08	Compare 4 with 5 and $4 < 5$
01	02	04	05	08	Compare 5 with 8 and $5 < 8$

Fifth Iteration

01	02	04	05	08	Compare 1 with 2 and $1 < 2$
01	02	04	05	08	Compare 2 with 4 and $2 < 4$
01	02	04	05	08	Compare 4 with 5 and $4 < 5$
01	02	04	05	08	Compare 5 with 8 and $5 < 8$

As we can see in above example the list sorted by third iteration and it is useless to go for 4th and 5th iterations. So we use a flag which determines whether a swap operation is done in last iteration or not. If a swap operation is not done in last iteration we will stop remaining iterations since the list is already in sorted order. For the above example we will stop iterating once third iteration is done as the whole list is sorted by then.

Pseudo-code

```
procedure bubbleSort( A : list of sortable items )
  repeat
    swapped = false
    for i = 0 to length(A) - 1
      if A[i] > A[i+1] then
        swap( A[i], A[i+1] )
        swapped = true
      end if
    end for
  until not swapped
end procedure
```

2. Greedy Method

- Greedy method is the most straightforward designed technique.
- As the name suggest they are short sighted in their approach taking decision on the basis of the information immediately at the hand without worrying about the effect these decision may have in the future.

DEFINITION:

- A problem with N inputs will have some constraints .any subsets that satisfy these constraints are called a feasible solution.
- A feasible solution that either maximize can minimize a given objectives function is called an optimal solution.

Control algorithm for Greedy Method:

```
1.Algorithm Greedy (a,n)
2.//a[1:n] contain the 'n' inputs
3. {
4.solution =0;//Initialise the solution.
5.For i=1 to n do
6.{
7.x=select(a);
8.if(feasible(solution,x))then
9.solution=union(solution,x);
10.}
11.return solution;
12.}
```

* The function select an input from a[] and removes it. The select input value is assigned to X.

- Feasible is a Boolean value function that determines whether X can be included into the solution vector.
- The function Union combines X with The solution and updates the objective function.
- The function Greedy describes the essential way that a greedy algorithm will once a particular problem is chosen ends the function subset, feasible & union are properly implemented.

Example

Knapsack Problem

- We are given n objects and knapsack or bag with capacity M object I has a weight W_i where I varies from 1 to N.
- The problem is we have to fill the bag with the help of N objects and the resulting profit has to be maximum.
- Formally the problem can be stated as

Maximize $\sum x_i p_i$ subject to $\sum x_i W_i \leq M$
Where x_i is the fraction of object and it lies between 0 to 1.

- There are so many ways to solve this problem, which will give many feasible solution for which we have to find the optimal solution.
- But in this algorithm, it will generate only one solution which is going to be feasible as well as optimal.

- First, we find the profit & weight rates of each and every object and sort it according to the descending order of the ratios.
- Select an object with highest p/w ratio and check whether its height is lesser than the capacity of the bag.
- If so place 1 unit of the first object and decrement .the capacity of the bag by the weight of the object you have placed.
- Repeat the above steps until the capacity of the bag becomes less than the weight of the object you have selected .in this case place a fraction of the object and come out of the loop.
- Whenever you selected.

ALGORITHM:

```

1.Algorithm Greedy knapsack (m,n)
2//P[1:n] and the w[1:n]contain the profit
3.// & weight res'.of the n object ordered.
4.//such that  $p[i]/w[i] \geq p[i+1]/W[i+1]$ 
5.//n is the Knapsack size and x[1:n] is the solution vertex.
6.{
7.for I=1 to n do a[I]=0.0;
8.U=n;
9.For I=1 to n do
10.{
11.if (w[i]>u)then break;
13.x[i]=1.0;U=U-w[i]
14.}
15.if(i<=n)then x[i]=U/w[i];
16.}

```

Example:

Capacity=20
N=3 ,M=20
Wi=18,15,10
Pi=25,24,15

$P_i/W_i = 25/18 = 1.36, 24/15 = 1.6, 15/10 = 1.5$

Descending Order → $P_i/W_i \rightarrow 1.6 \quad 1.5 \quad 1.36$
 $P_i = 24 \quad 15 \quad 25$
 $W_i = 15 \quad 10 \quad 18$
 $X_i = 1 \quad 5/10 \quad 0$

$P_i X_i = 1*24 + 0.5*15 \rightarrow 31.5$

The optimal solution is $\rightarrow 31.5$

<i>X1</i>	<i>X2</i>	<i>X3</i>	<i>WiXi</i>	<i>PiXi</i>
1/2	1/3	1/4	16.6	24.25
1	2/5	0	20	18.2
0	2/3	1	20	31
0	1	1/2	20	31.5

Of these feasible solution Solution 4 yield the Max profit .As we shall soon see this solution is optimal for the given problem instance.

3. Dynamic program general method

- It is an algorithm design method that can be used when the solution to a problem can be viewed as the result of a sequence of decisions.
- The idea of dynamic programming is thus quit simple: avoid calculating the same thing twice, usually by keeping a table of known result that fills up a sub instances are solved.
- Divide and conquer is a top-down method. When a problem is solved by divide and conquer, we immediately attack the complete instance, which we then divide into smaller and smaller sub-instances as the algorithm progresses.
- Dynamic programming on the other hand is a bottom-up technique. We usually start with the smallest and hence the simplest sub- instances. By combining their solutions, we obtain the answers to sub-instances of increasing size, until finally we arrive at the solution of the original instances.
- The essential difference between the greedy method and dynamic programming is that the greedy method only one decision sequence is ever generated. In dynamic programming, many decision sequences may be generated. However, sequences containing sub-optimal sub-sequences cannot be optimal and so will not be generated.

Because of principle of optimality, decision sequences containing subsequences that are suboptimal are not considered. Although the total number of different decision sequences is exponential in the number of decisions(if there are d choices for each of the n decisions to be made then there are d^n possible decision sequences),Dynamic programming algorithms often have a polynomial complexity.

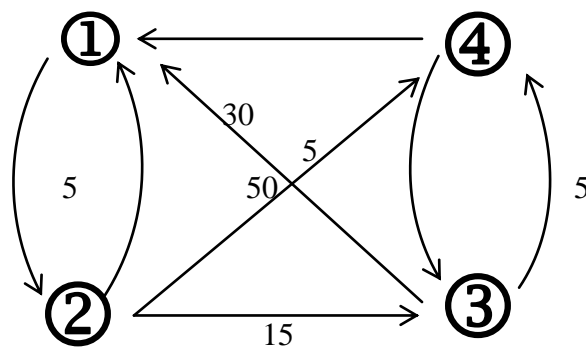
Example

All Pair Shortest Path Algorithm

- Let $G(V,E)$ be a directed graph with n vertices , ‘ E ’ is the set of edges & V is the set of n vertices.
- Each edge has an associated non-negative length.
- We want to calculate the length of the shortest path between each pair of nodes.
- i.e)Shortest path between every vertex to all other vertices.
- Suppose the nodes of G are numbered from 1 to n , so $n=\{1,2,...n\}$
- $cost(i,j)$ is length of edge $\langle i,j \rangle$ and it is called as cost adjacency matrix.
 - $cost(i,i)=0$ for $i=1,2,...n$,
 - $cost(i,j)=cost$ of edge $\langle i,j \rangle$ for all $i \& j$
 - $cost(i,j)=infinity$, if the edge (i,j) does not exist.

- The all pairs shortest path problem is to determine a matrix A such that $A(i,j)$ is the length of a shortest path from i to j .
- The principle of optimality:
If k is the node on the shortest path from i to j then the part of the path from i to k and the part from k to j must also be optimal, that is shorter.
- First create a cost adjacency matrix for the given graph.
- Copy the above matrix to matrix D , which will give the direct distance between nodes.
- We have to perform $n*n$ iterations for each iteration of k . The matrix D will give you the distance between nodes with only $(1,2,...,k)$ as intermediate nodes.
- At the iteration k , we have to check for each pair of nodes (i,j) whether or not there exists a path from i to j passing through node k .
- Likewise we have to find the value for n iterations (ie) for n nodes.

15



15

Fig: floyd's algorithm and work

ALGORITHM :

Algorithm Allpaths(cost,A,n)

//cost[1...n,1...n] is the cost adjacency matrix of graph with n vertices

//A[i,j] is the cost of shortest path from vertex i to j

//cost[i,i]=0.0 for $1 \leq i \leq n$

{

 for $i:=1$ to n do

 for $j:=1$ to n do

$A[i,j] := \text{cost}[i,j]$; //copy cost into A

 for $k:=1$ to n do

 for $i:=1$ to n do

 for $j:=1$ to n do

$A[i,j] := \min\{A[i,j], A[i,k] + A[k,j]\}$;

}

COST ADJACENCY MATRIX :

❖ At 0th iteration it nil give you the direct distances between any 2 nodes

$$\text{cost}(i,j) = A(i,j) = D_0 = \begin{array}{c|c} \begin{array}{cccc} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 15 & \infty & 5 & 0 \end{array} & \begin{array}{cccc} & & & \\ 30 & \infty & 0 & 15 \end{array} \end{array}$$

It gives the direct distances between the two nodes.

❖ At 1st iteration we have to check the each pair(i,j) whether there is a path through node 1.if so we have to check whether it is minimum than the previous value and if i is so than the distance through 1 is the value of d1(i,j).at the same time we have to solve the intermediate node in the matrix position p(i,j).

for k= 1 to 4

When k=1

for i= 1 to 4

when i=1

for j= 1 to 4

when j=1

$A[1,1] = \min(A[1,1], A[1,1] + A[1,1]) = \min(0, 0) = 0$

When j=2

$A[1,2] = \min(A[1,2], A[1,1] + A[1,2]) = \min(5, 0 + 5) = 5$

When j=3

$A[1,3] = \min(A[1,3], A[1,1] + A[1,3]) = \min(\infty, 0 + \infty) = \infty$

When j=4

$A[1,4] = \min(A[1,4], A[1,1] + A[1,4]) = \min(\infty, 0 + \infty) = \infty$

When i=2

for j= 1 to 4

when j=1

$A[2,1] = \min(A[2,1], A[2,1] + A[1,1]) = \min(50, 50 + 0) = 50$

When j=2

$A[2,2] = \min(A[2,2], A[2,1] + A[1,2]) = \min(0, 50 + 5) = 0$

When j=3

$A[2,3] = \min(A[2,3], A[2,1] + A[1,3]) = \min(15, 50 + \infty) = 15$

When j=4

$A[2,4] = \min(A[2,4], A[2,1] + A[1,4]) = \min(5, 50 + \infty) = 5$

When i=3

for j= 1 to 4

when j=1

$A[3,1] = \min(A[3,1], A[3,1] + A[1,1]) = \min(30, 30 + 0) = 30$

When j=2

$A[3,2] = \min(A[3,2], A[3,1] + A[1,2]) = \min(\infty, 30 + 5) = 35$

When j=3

$A[3,3] = \min(A[3,3], A[3,1] + A[1,3]) = \min(0, 30 + \infty) = 0$

When j=4

$$A[3,4]=\min(A[3,4],A[3,1]+A[1,4])=\min(15,30+\infty)=15$$

When i=4

for j= 1 to 4

when j=1

$$A[4,1]=\min(A[4,1],A[4,1]+A[1,1])=\min(15,15+0)=15$$

When j=2

$$A[4,2]=\min(A[4,2],A[4,1]+A[1,2])=\min(\infty,15+5)=20$$

When j=3

$$A[4,3]=\min(A[4,3],A[4,1]+A[1,3])=\min(5,15+\infty)=5$$

When j=4

$$A[4,4]=\min(A[4,4],A[4,1]+A[1,4])=\min(0,15+\infty)=0$$

	i	j	A[i,j]
1	1	1	0
		2	5
		3	∞
		4	∞
	2	1	50
		2	0
		3	15
		4	5
	3	1	30
		2	35
		3	0
		4	15
	4	1	15
		2	20
		3	5
		4	0

$$\begin{array}{c}
 50 \ 0 \ 15 \ 5 \\
 P[3,2]=1 \\
 D_1=
 \end{array}
 \left| \begin{array}{cccc}
 0 & 5 & \infty & \infty \\
 30 & \mathbf{35} & 0 & 15 \\
 15 & \mathbf{20} & 5 & 0
 \end{array} \right|
 \begin{array}{c}
 0 \quad 0 \\
 P[4,2]=1 \quad P_1=
 \end{array}
 \left| \begin{array}{cccc}
 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 \\
 0 & 1 & 0 & 0 \\
 0 & 1 & 0 & 0
 \end{array} \right|$$

When k=2, we have obtained the matrix D_2 and P_2

$$\begin{array}{cc}
 & \begin{array}{c|c|c|c|c}
 & 0 & 5 & 20 & 10 \\
 \hline
 & 50 & 0 & 15 & 5 \\
 \hline
 & & & &
 \end{array} \\
 \begin{array}{c} 30 \ 35 \ 0 \ 15 \\ 15 \ 20 \ 5 \ 0 \end{array} & D2=
 \end{array}
 \begin{array}{c}
 P[1,3]=2 \\
 P[1,4]=2
 \end{array}
 \begin{array}{c}
 P_2= \\
 0
 \end{array}
 \begin{array}{c|c|c|c|c}
 & 0 & 0 & 2 & 2 \\
 \hline
 & 0 & 0 & 0 & 0 \\
 \hline
 & 1 & 0 & 1 & 1 \\
 \hline
 & 0 & 1 & 0 & 0
 \end{array}$$

$$D3= \begin{array}{c|c|c|c|c}
 & 0 & 5 & 20 & 10 \\
 \hline
 & 45 & 0 & 15 & 5 \\
 \hline
 & 30 & 35 & 0 & 15 \\
 \hline
 & 15 & 20 & 5 & 0
 \end{array}
 \begin{array}{c}
 P[2,1]=3
 \end{array}$$

$$\begin{array}{c} 15 \ 20 \ 5 \ 0 \end{array}
 D4= \begin{array}{c|c|c|c|c}
 & 0 & 5 & 15 & 10 \\
 \hline
 & 20 & 0 & 10 & 5 \\
 \hline
 & 30 & 35 & 0 & 15 \\
 \hline
 & & & &
 \end{array}
 \begin{array}{c}
 P[1,3]=4 \\
 P[2,3]=4
 \end{array}$$

- D4 will give the shortest distance between any pair of nodes.
- If you want the exact path then we have to refer the matrix p. The matrix will be,

$$P= \begin{array}{c|c|c|c|c}
 & 0 & 0 & 4 & 2 \\
 \hline
 & 3 & 0 & 4 & 0 \\
 \hline
 & 0 & 1 & 0 & 0 \\
 \hline
 & 0 & 1 & 0 & 0
 \end{array}
 \xrightarrow{0} \text{ direct path}$$

- Since, $p[1,3]=4$, the shortest path from 1 to 3 passes through 4.
- Looking now at $p[1,4]$ & $p[4,3]$ we discover that between 1 & 4, we have to go to node 2 but that from 4 to 3 we proceed directly.
- Finally we see the trips from 1 to 2, & from 2 to 4, are also direct.
- The shortest path from 1 to 3 is 1,2,4,3.

ALGORITHM :

Function Floyd (L[1..r,1..r]):array[1..n,1..n]
array D[1..n,1..n]


```

D = L
For k = 1 to n do
For i = 1 to n do
For j = 1 to n do

D [ i , j ] = min (D[ i, j ], D[ i, k ] + D[ k, j ] )

Return D

```

ANALYSIS:

This algorithm takes a time of $\theta(n^3)$

4. Backtracking method

- It is one of the most general algorithm design techniques.
- Many problems which deal with searching for a set of solutions or for an optimal solution satisfying some constraints can be solved using the backtracking formulation.
- To apply backtracking method, the desired solution must be expressible as an n-tuple $(x_1 \dots x_n)$ where x_i is chosen from some finite set S_i .
- The problem is to find a vector, which maximizes or minimizes a criterion function $P(x_1 \dots x_n)$.
- The major advantage of this method is, once we know that a partial vector (x_1, \dots, x_i) will not lead to an optimal solution that $(m_{i+1} \dots m_n)$ possible test vectors may be ignored entirely.
- Many problems solved using backtracking require that all the solutions satisfy a complex set of constraints.
- These constraints are classified as:

- i) Explicit constraints.
- ii) Implicit constraints.

1) Explicit constraints:

Explicit constraints are rules that restrict each X_i to take values only from a given set.

Some examples are,

$X_i \geq 0$ or $S_i = \{\text{all non-negative real nos.}\}$

$X_i = 0$ or 1 or $S_i = \{0, 1\}$.

$L_i \leq X_i \leq U_i$ or $S_i = \{a: L_i \leq a \leq U_i\}$

- All tuples that satisfy the explicit constraint define a possible solution space for I.

2) Implicit constraints:

The implicit constraint determines which of the tuples in the solution space I can actually satisfy the criterion functions.

Algorithm:

Algorithm IBacktracking (n)

// This schema describes the backtracking procedure .All solutions are generated in X[1:n]

//and printed as soon as they are determined.

```
{
  k=1;
  While (k ≠ 0) do
  {
    if (there remains all untried
     $X[k] \in T(X[1], [2], \dots, X[k-1])$  and  $B_k(X[1], \dots, X[k])$  is true ) then
    {
      if( $X[1], \dots, X[k]$  )is the path to the answer node)
      Then write( $X[1:k]$ );
      k=k+1;           //consider the next step.
    }
  }
  else k=k-1;           //consider backtracking to the previous set.
}
```

- All solutions are generated in X[1:n] and printed as soon as they are determined.
- $T(X[1], \dots, X[k-1])$ is all possible values of X[k] gives that $X[1], \dots, X[k-1]$ have already been chosen.
- $B_k(X[1], \dots, X[k])$ is a boundary function which determines the elements of X[k] which satisfies the implicit constraint.
- Certain problems which are solved using backtracking method are,
 1. Sum of subsets.
 2. Graph coloring.
 3. Hamiltonian cycle.
 4. N-Queens problem.

Example

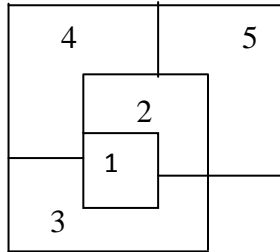
Graph coloring

Let 'G' be a graph and 'm' be a given positive integer. If the nodes of 'G' can be colored in such a way that no two adjacent nodes have the same color. Yet only 'M' colors are used. So it's called M-color ability decision problem.

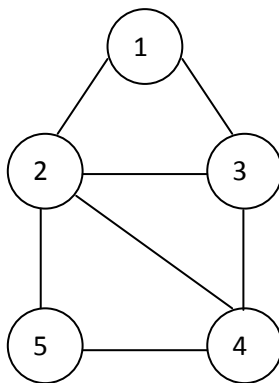
- The graph G can be colored using the smallest integer 'm'. This integer is referred to as chromatic number of the graph.

- A graph is said to be planar iff it can be drawn on plane in such a way that no two edges cross each other.
- Suppose we are given a map then, we have to convert it into planar. Consider each and every region as a node. If two regions are adjacent then the corresponding nodes are joined by an edge.

Consider a map with five regions and its graph.



- 1 is adjacent to 2, 3, 4.
 2 is adjacent to 1, 3, 4, 5
 3 is adjacent to 1, 2, 4
 4 is adjacent to 1, 2, 3, 5
 5 is adjacent to 2, 4



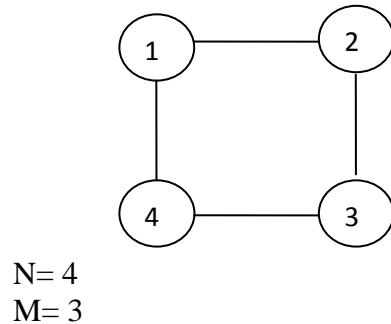
Steps to color the Graph:

- ❖ First create the adjacency matrix $graph(1:m, 1:n)$ for a graph, if there is an edge between i, j then $C(i, j) = 1$ otherwise $C(i, j) = 0$.
- ❖ The Colors will be represented by the integers $1, 2, \dots, m$ and the solutions will be stored in the array $X(1), X(2), \dots, X(n)$, $X(\text{index})$ is the color, index is the node.
- ❖ The formula is used to set the color is,

$$X(k) = (X(k) + 1) \% (m + 1)$$
- ❖ First one chromatic number is assigned, after assigning a number for 'k' node, we have to check whether the adjacent nodes have got the same values if so then we have to assign the next value.

- ❖ Repeat the procedure until all possible combinations of colors are found.
- ❖ The function which is used to check the adjacent nodes and same color is,
If((Graph (k,j) == 1) and X(k) = X(j))

Example:



Adjacency Matrix:

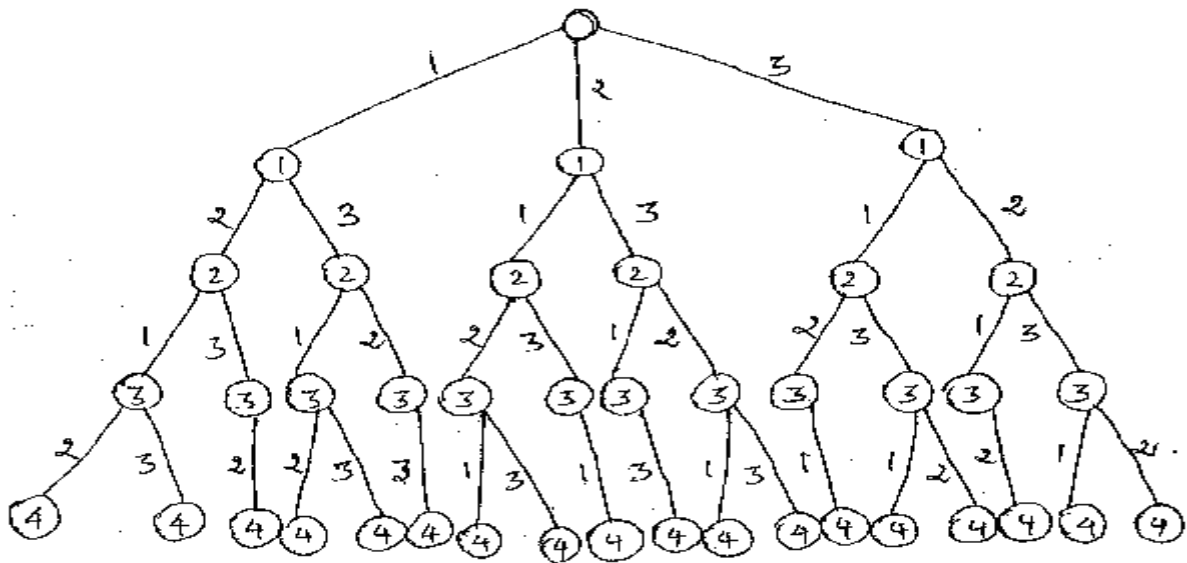
$$\begin{vmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{vmatrix}$$

→ Problem is to color the given graph of 4 nodes using 3 colors.

→ Node-1 can take the given graph of 4 nodes using 3 colors.

→ The state space tree will give all possible colors in that ,the numbers which are inside the circles are nodes ,and the branch with a number is the colors of the nodes.

State Space Tree:



Algorithm:

Algorithm mColoring(k)

// the graph is represented by its Boolean adjacency matrix $G[1:n, 1:n]$. All assignments // of 1, 2, ..., m to the vertices of the graph such that adjacent vertices are assigned // distinct integers are printed. 'k' is the index of the next vertex to color.

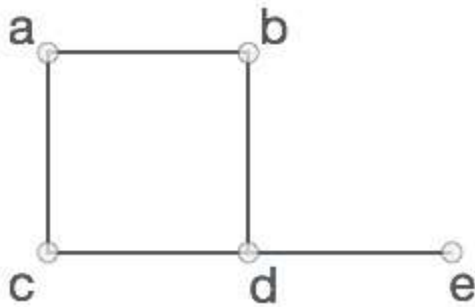
```
{
repeat
{
    // generate all legal assignment for X[k].
    Nextvalue(k); // Assign to X[k] a legal color.
    If (X[k]=0) then return; // No new color possible.
    If (k=n) then // Almost 'm' colors have been used to color the 'n' vertices
        Write(x[1:n]);
    Else mcoloring(k+1);
}until(false);
}
```

UNIT-III

GRAPH BASIC TERMINOLOGIES

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**.

Formally, a graph is a pair of sets (V, E) , where V is the set of vertices and E is the set of edges, connecting the pairs of vertices. Take a look at the following graph –



In the above graph,

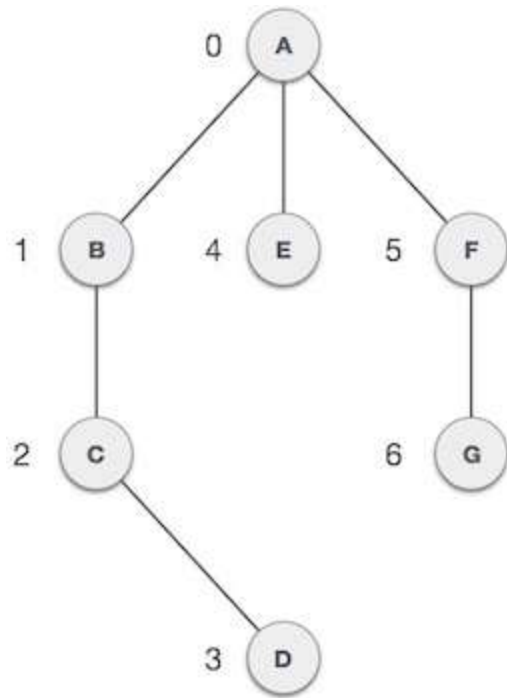
$$V = \{a, b, c, d, e\}$$

$$E = \{ab, ac, bd, cd, de\}$$

Graph Data Structure

Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms –

- **Vertex** – Each node of the graph is represented as a vertex. In the following example, the labeled circle represents vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1 and so on.
- **Edge** – Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represents edges. We can use a two-dimensional array to represent an array as shown in the following image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.
- **Adjacency** – Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.
- **Path** – Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.



Basic Operations

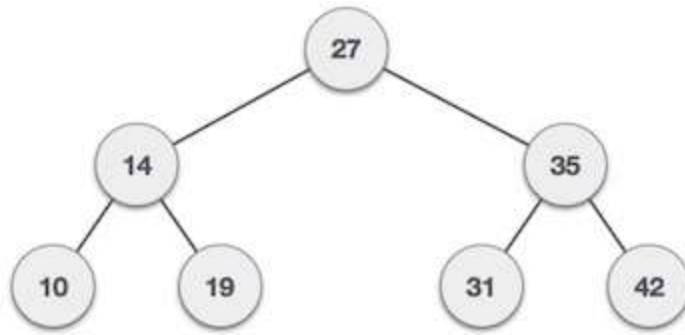
Following are basic primary operations of a Graph –

- **Add Vertex** – Adds a vertex to the graph.
- **Add Edge** – Adds an edge between the two vertices of the graph.
- **Display Vertex** – Displays a vertex of the graph.

TREE BASIC TERMINOLOGIES

Tree represents the nodes connected by edges. We will discuss binary tree or binary search tree specifically.

Binary Tree is a special data structure used for data storage purposes. A binary tree has a special condition that each node can have a maximum of two children. A binary tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list.



We're going to implement tree using node object and connecting them through references.

Tree Node

The code to write a tree node would be similar to what is given below. It has a data part and references to its left and right child nodes.

```
struct node {  
    int data;  
    struct node *leftChild;  
    struct node *rightChild;  
};
```

In a tree, all nodes share common construct.

BST Basic Operations

The basic operations that can be performed on a binary search tree data structure, are the following –

- **Insert** – Inserts an element in a tree/create a tree.
- **Search** – Searches an element in a tree.
- **Preorder Traversal** – Traverses a tree in a pre-order manner.
- **Inorder Traversal** – Traverses a tree in an in-order manner.
- **Postorder Traversal** – Traverses a tree in a post-order manner.

We shall learn creating (inserting into) a tree structure and searching a data item in a tree in this chapter. We shall learn about tree traversing methods in the coming chapter.

Insert Operation

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

Algorithm

```
If root is NULL
  then create root node
return

If root exists then
  compare the data with node.data

  while until insertion position is located

    If data is greater than node.data
      goto right subtree
    else
      goto left subtree

  endwhile

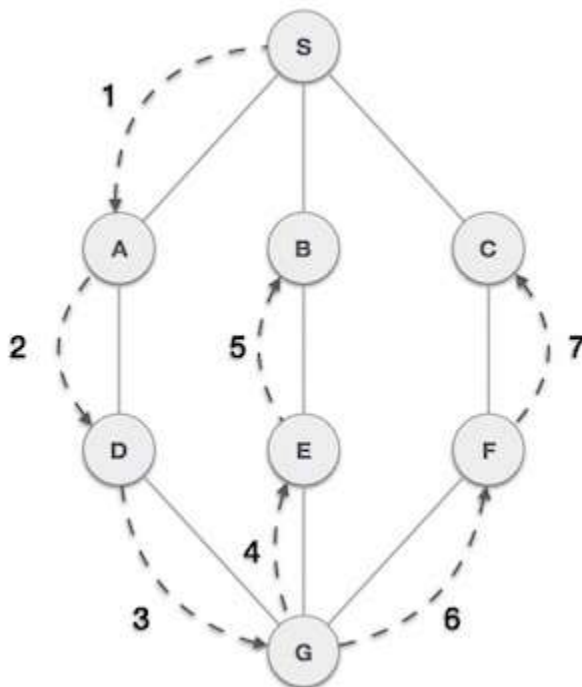
  insert data

end If
```

1. Traversal Algorithm

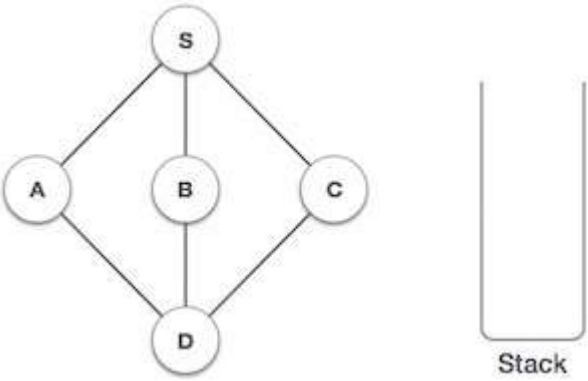
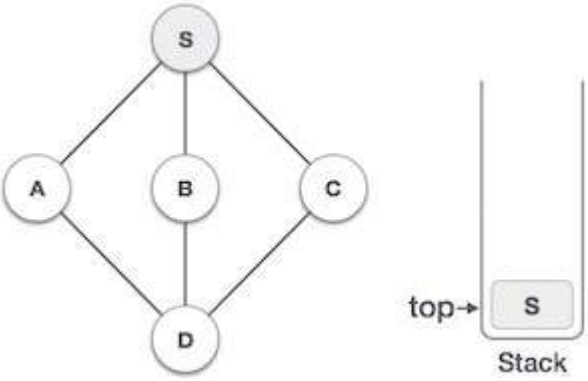
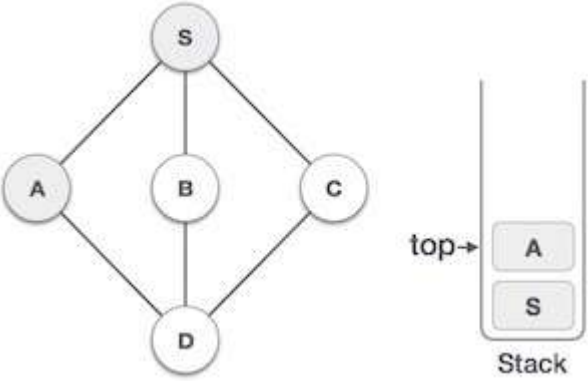
Depth First Search

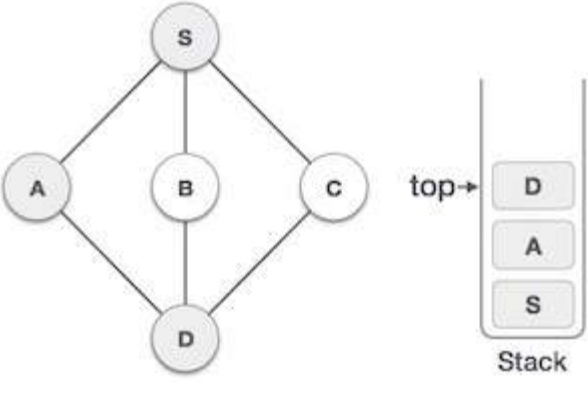
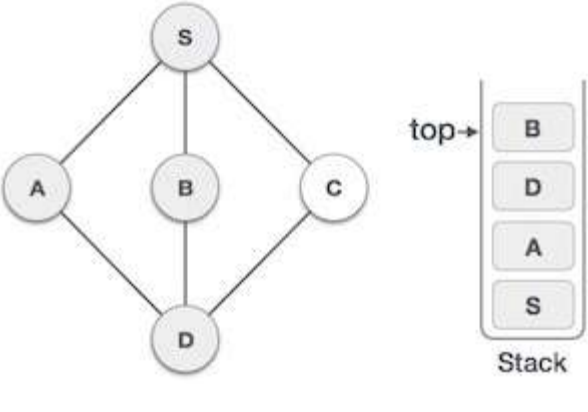
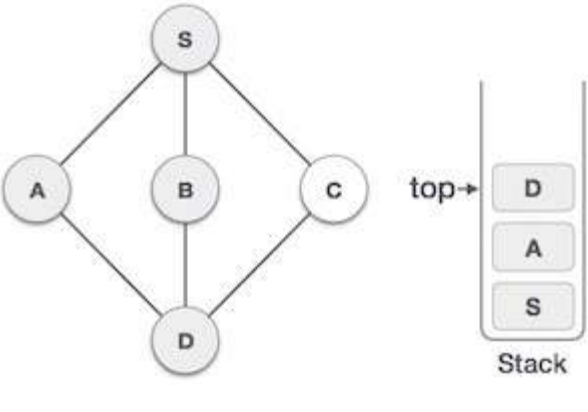
Depth First Search (DFS) algorithm traverses a graph in a depth wise motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



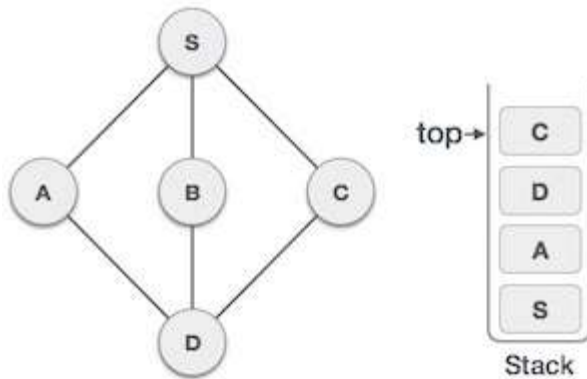
As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** – Repeat Rule 1 and Rule 2 until the stack is empty.

Step	Traversal	Description
1		Initialize the stack.
2		Mark S as visited and put it onto the stack. Explore any unvisited adjacent node from S . We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.
3		Mark A as visited and put it onto the stack. Explore any unvisited adjacent node from A . Both S and D are adjacent to A but we are concerned for unvisited nodes only.

4		<p>Visit D and mark it as visited and put onto the stack. Here, we have B and C nodes, which are adjacent to D and both are unvisited. However, we shall again choose in an alphabetical order.</p>
5		<p>We choose B, mark it as visited and put onto the stack. Here B does not have any unvisited adjacent node. So, we pop B from the stack.</p>
6		<p>We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find D to be on the top of the stack.</p>

7

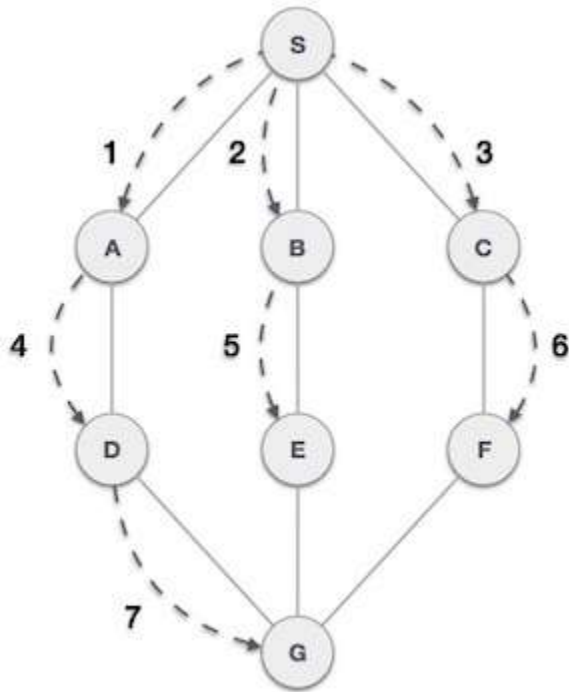


Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack.

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

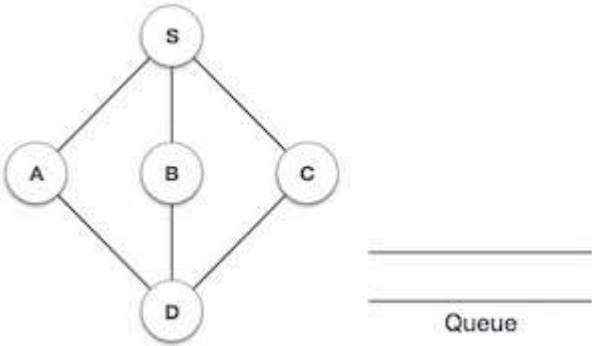
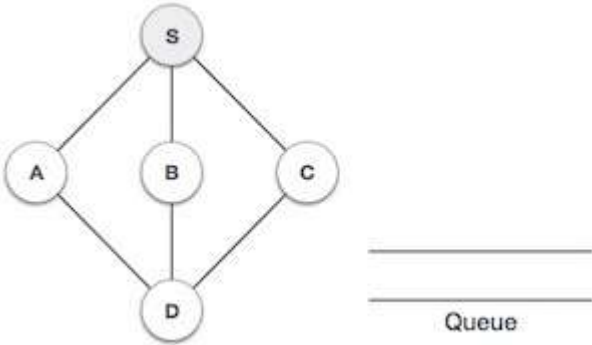
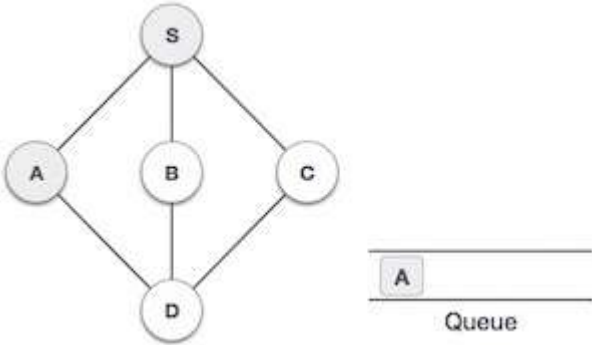
Breadth First Search (BFS)

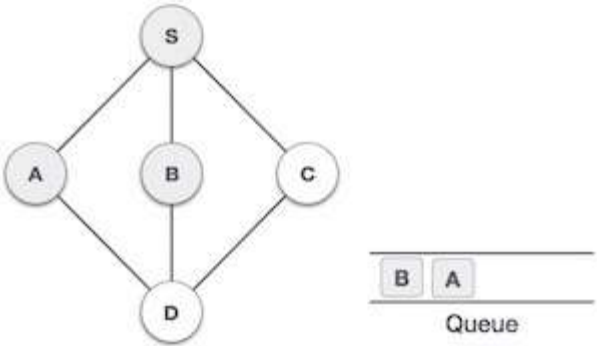
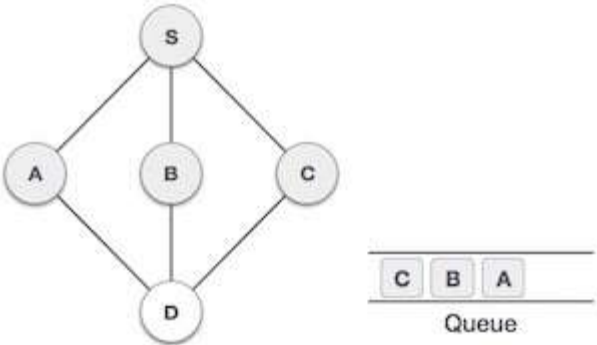
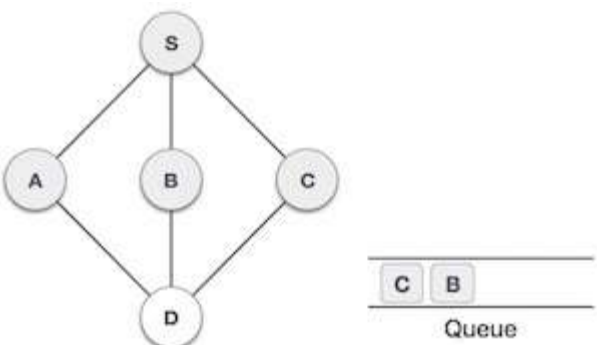
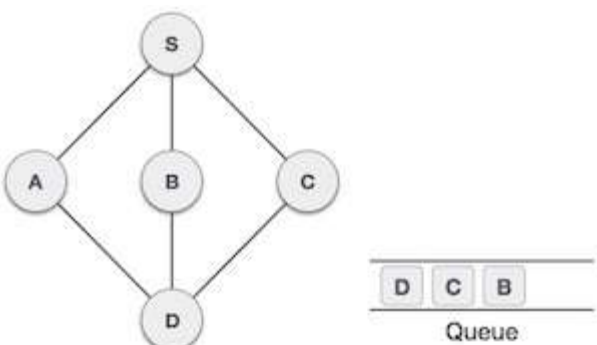
Breadth First Search (BFS) algorithm traverses a graph in a breadth wise motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- **Rule 2** – If no adjacent vertex is found, remove the first vertex from the queue.
- **Rule 3** – Repeat Rule 1 and Rule 2 until the queue is empty.

Step	Traversal	Description
1		Initialize the queue.
2		We start from visiting S (starting node), and mark it as visited.
3		We then see an unvisited adjacent node from S . In this example, we have three nodes but alphabetically we choose A , mark it as visited and enqueue it.

4		<p>Next, the unvisited adjacent node from S is B. We mark it as visited and enqueue it.</p>
5		<p>Next, the unvisited adjacent node from S is C. We mark it as visited and enqueue it.</p>
6		<p>Now, S is left with no unvisited adjacent nodes. So, we dequeue and find A.</p>
7		<p>From A we have D as unvisited adjacent node. We mark it as visited and enqueue it.</p>

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

2. Shortest Path Algorithm

The shortest path problem is about finding a path between 2 vertices in a graph such that the total sum of the edges weights is minimum.

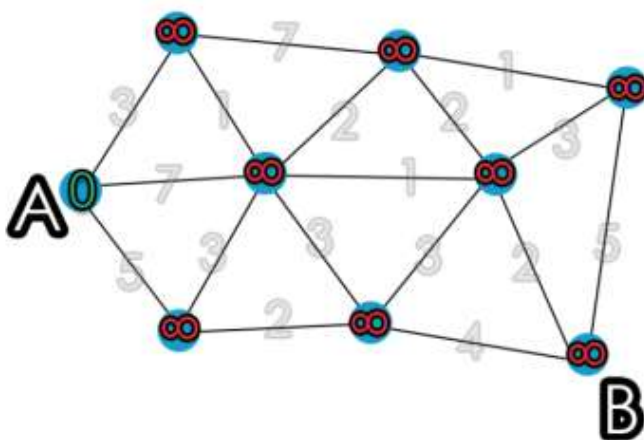
Dijkstra's Algorithm

Dijkstra's algorithm has many variants but the most common one is to find the shortest paths from the source vertex to all other vertices in the graph.

Algorithm Steps:

- Set all vertices distances = infinity except for the source vertex, set the source distance = 0.
- Push the source vertex in a min-priority queue in the form (distance , vertex), as the comparison in the min-priority queue will be according to vertices distances.
- Pop the vertex with the minimum distance from the priority queue (at first the popped vertex = source).
- Update the distances of the connected vertices to the popped vertex in case of "current vertex distance + edge weight < next vertex distance", then push the vertex with the new distance to the priority queue.
- If the popped vertex is visited before, just continue without using it.
- Apply the same algorithm again until the priority queue is empty.

Dijkstra's algorithm finds a shortest path tree from a single source node, by building a set of nodes that have minimum distance from the source.



The graph has the following:

- vertices, or nodes, denoted in the algorithm by vv or uu ;

- weighted edges that connect two nodes: $(u, v, w(u, v))$ denotes an edge, and $w(u, v)$ denotes its weight. In the diagram on the right, the weight for each edge is written in gray.

This is done by initializing three values:

- **dist**, an array of distances from the source node S to each node in the graph, initialized the following way: $\text{dist}(S) = 0$; and for all other nodes v , $\text{dist}(v) = \infty$. This is done at the beginning because as the algorithm proceeds, the **dist** from the source to each node v in the graph will be recalculated and finalized when the shortest distance to v is found
- **Q**, a [queue](#) of all nodes in the graph. At the end of the algorithm's progress, **Q** will be empty.
- **S**, an empty [set](#), to indicate which nodes the algorithm has visited. At the end of the algorithm's run, **S** will contain all the nodes of the graph.

The algorithm proceeds as follows:

1. While **Q** is not empty, pop the node v , that is not already in **S**, from **Q** with the smallest **dist** (v). In the first run, source node S will be chosen because $\text{dist}(S)$ was initialized to 0. In the next run, the next node with the smallest **dist** value is chosen.
2. Add node v to **S**, to indicate that v has been visited
3. Update **dist** values of adjacent nodes of the current node v as follows: for each new adjacent node u ,
 - if $\text{dist}(v) + \text{weight}(u, v) < \text{dist}(u)$, there is a new minimal distance found for u , so update **dist** (u) to the new minimal distance value;
 - otherwise, no updates are made to **dist** (u).

The algorithm has visited all nodes in the graph and found the smallest distance to each node. **dist** now contains the shortest path tree from source S .

Note: The weight of an edge $(u, v, w(u, v))$ is taken from the value associated with $(u, v, w(u, v))$ on the graph.

Implementation

This is pseudocode for Dijkstra's algorithm, mirroring Python syntax. It can be used in order to implement the algorithm in any language.

```
function Dijkstra(Graph, source):
```

```
    dist[source] := 0           // Distance from source to source is set to 0
```

```
    for each vertex v in Graph: // Initializations
```

```
        if v ≠ source
```

```
            dist[v] := infinity // Unknown distance function from source to each node set to infinity
```

```
    add v to Q                  // All nodes initially in Q
```

```
while Q is not empty:         // The main loop
```

```
    v := vertex in Q with min dist[v] // In the first run-through, this vertex is the source node
```

```
    remove v from Q
```

```
    for each neighbor u of v: // where neighbor u has not yet been removed from Q.
```

```
        alt := dist[v] + length(v, u)
```

```
        if alt < dist[u]:      // A shorter path to u has been found
```

```
            dist[u] := alt     // Update distance of u
```

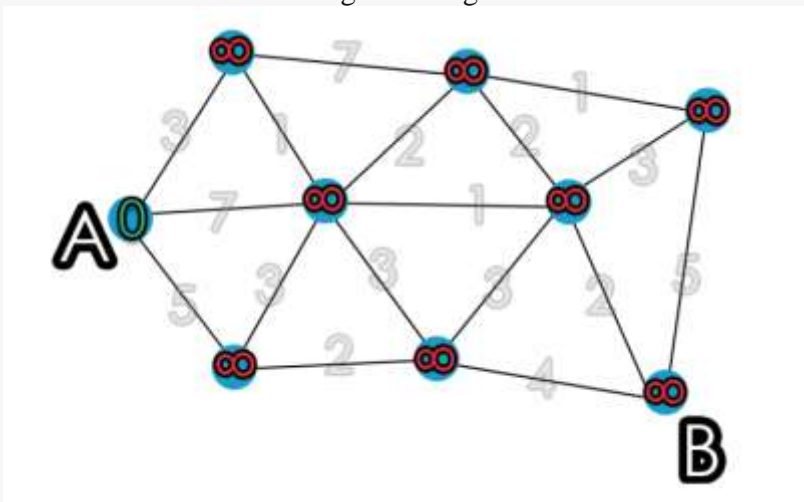
```
return dist[]
```

```
end function
```

Examples

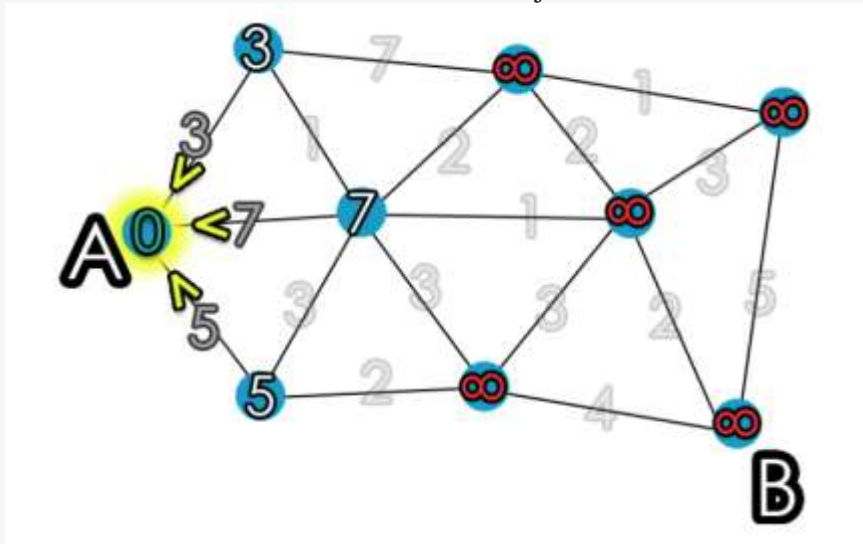
We step through Dijkstra's algorithm on the graph used in the algorithm above:

1. Initialize distances according to the algorithm.

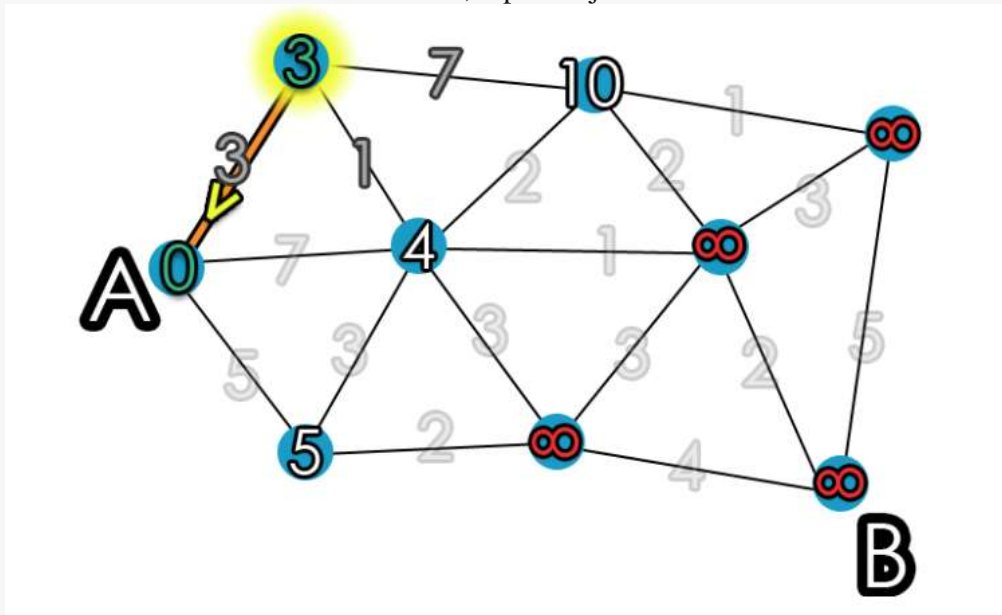


[31]

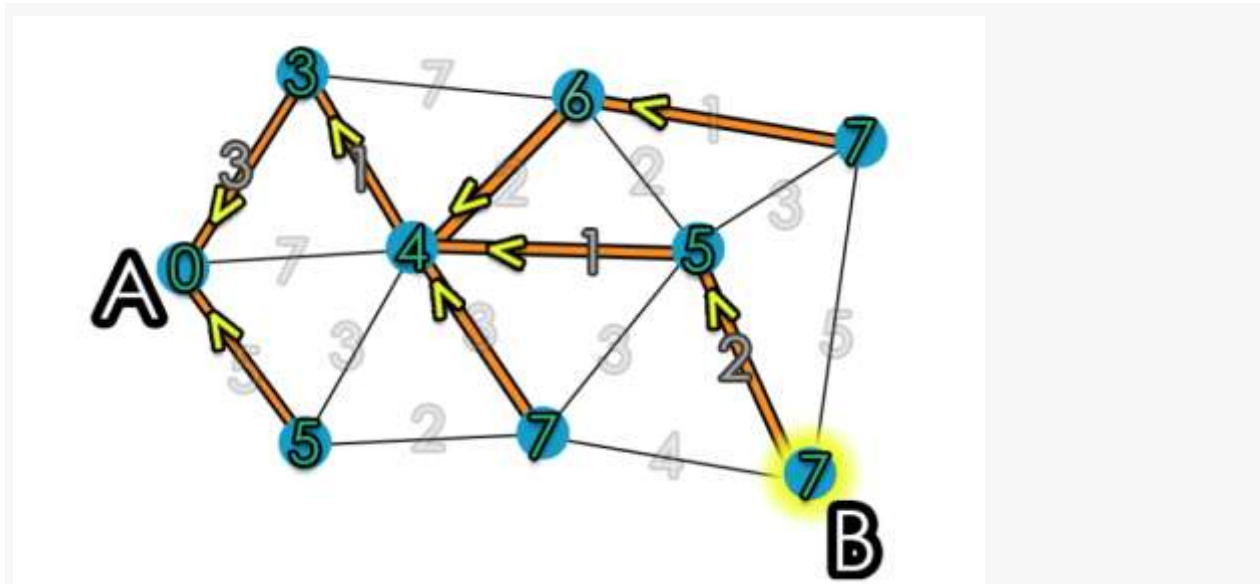
2. Pick first node and calculate distances to adjacent nodes.



3. Pick next node with minimal distance; repeat adjacent node distance calculations.



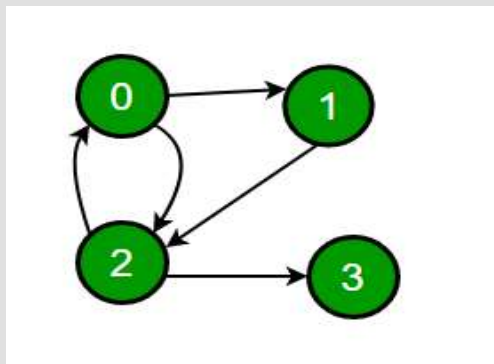
4. Final result of shortest-path tree^[6]



Transitive Closure of a Graph using DFS

Given a directed graph, find out if a vertex v is reachable from another vertex u for all vertex pairs (u, v) in the given graph. Here reachable mean that there is a path from vertex u to v . The reach-ability matrix is called transitive closure of a graph.

For example, consider below graph



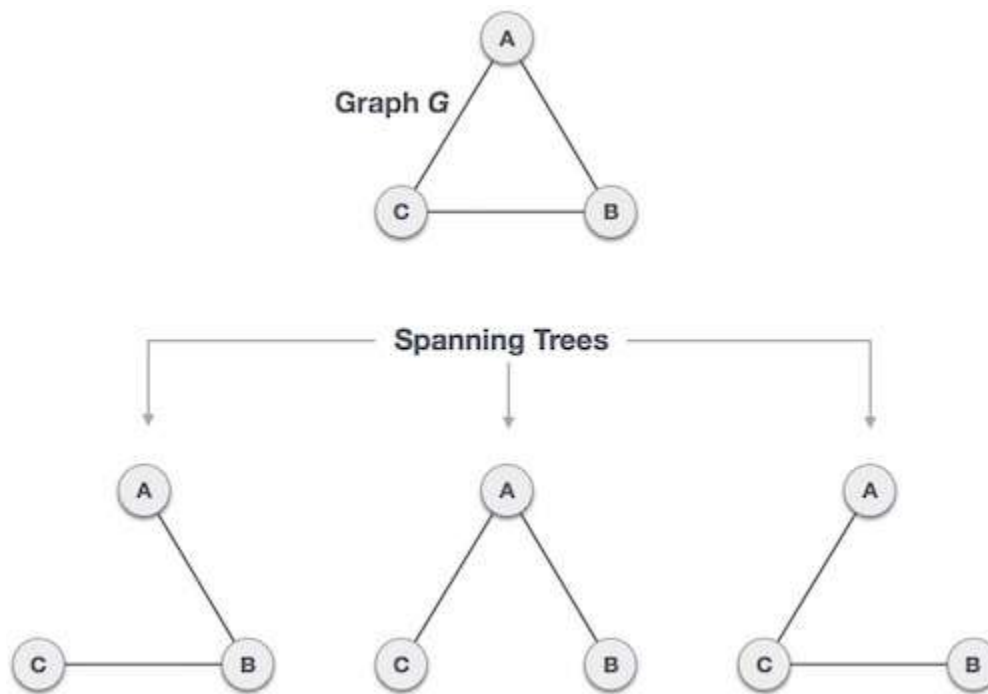
Transitive closure of above graphs is

```
1 1 1 1
1 1 1 1
1 1 1 1
0 0 0 1
```

Minimum Spanning Tree

A spanning tree is a subset of Graph G , which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected..

By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.



We found three spanning trees off one complete graph. A complete undirected graph can have maximum n^{n-2} number of spanning trees, where n is the number of nodes. In the above addressed example, n is 3, hence $3^{3-2} = 3$ spanning trees are possible.

General Properties of Spanning Tree

We now understand that one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph G –

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G , have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is **maximally acyclic**.

Mathematical Properties of Spanning Tree

- Spanning tree has $n-1$ edges, where n is the number of nodes (vertices).
- From a complete graph, by removing maximum $e - n + 1$ edges, we can construct a spanning tree.
- A complete graph can have maximum n^{n-2} number of spanning trees.

Thus, we can conclude that spanning trees are a subset of connected Graph G and disconnected graphs do not have spanning tree.

Application of Spanning Tree

Spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common application of spanning trees are –

- **Civil Network Planning**
- **Computer Network Routing Protocol**
- **Cluster Analysis**

Let us understand this through a small example. Consider, city network as a huge graph and now plans to deploy telephone lines in such a way that in minimum lines we can connect to all city nodes. This is where the spanning tree comes into picture.

Minimum Spanning Tree (MST)

In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph. In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

Minimum Spanning-Tree Algorithm

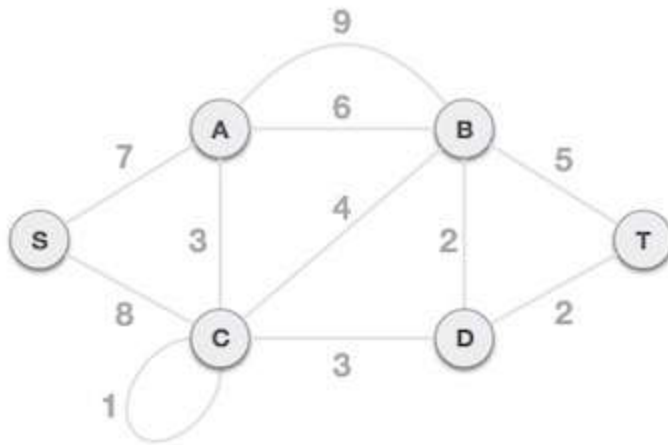
We shall learn about two most important spanning tree algorithms here –

- Kruskal's Algorithm
- Prim's Algorithm

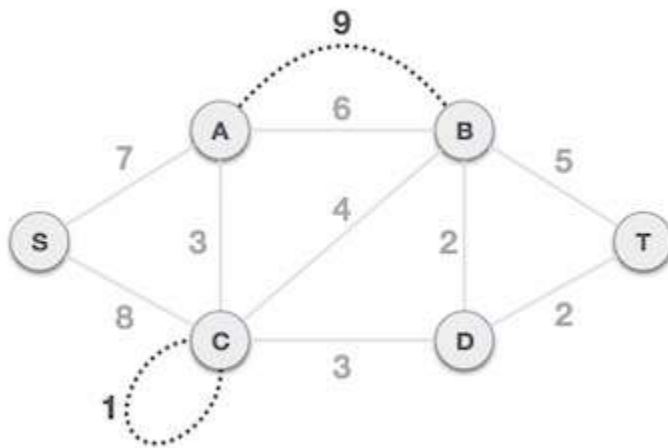
Kruskal's algorithm

Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.

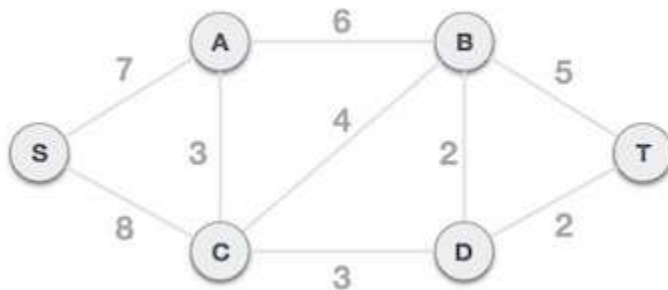
- To understand Kruskal's algorithm let us consider the following example –



- **Step 1 - Remove all loops and Parallel Edges**
- Remove all loops and parallel edges from the given graph.



- In case of parallel edges, keep the one which has the least cost associated and remove all others.

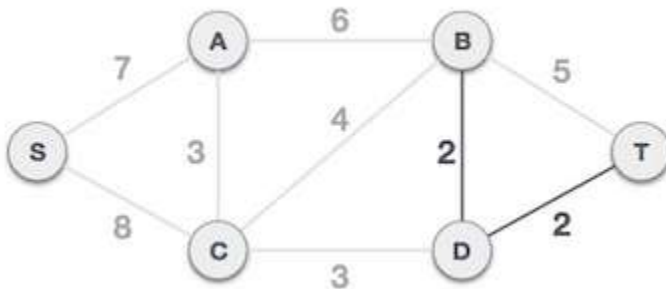


- **Step 2 - Arrange all edges in their increasing order of weight**
- The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

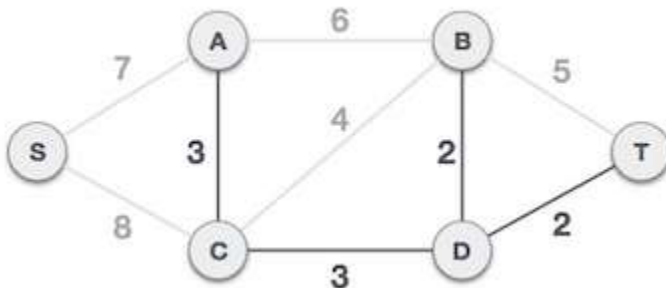
B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

- **Step 3 - Add the edge which has the least weightage**

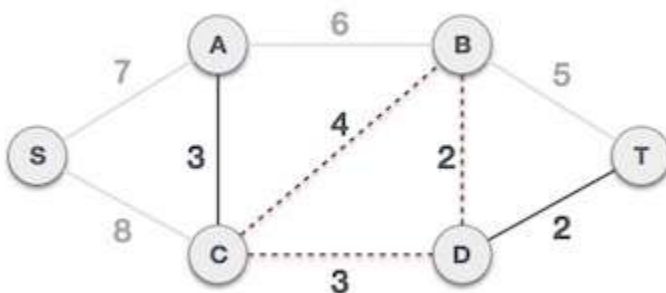
- Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.



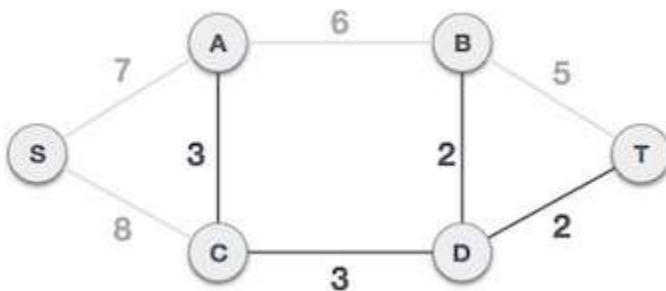
- The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.
- Next cost is 3, and associated edges are A,C and C,D. We add them again –



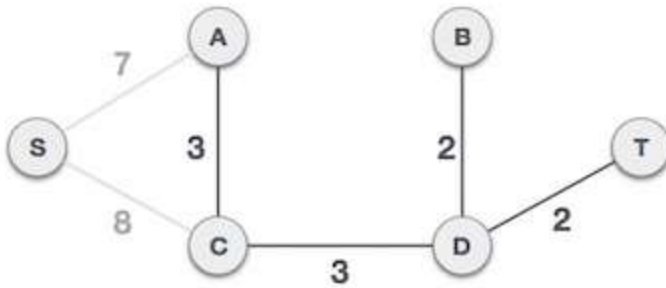
- Next cost in the table is 4, and we observe that adding it will create a circuit in the graph.



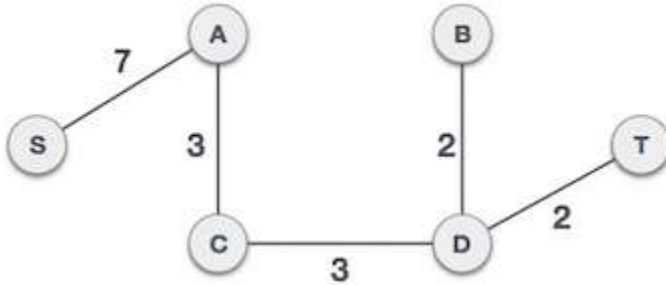
- We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



- We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.



- Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.



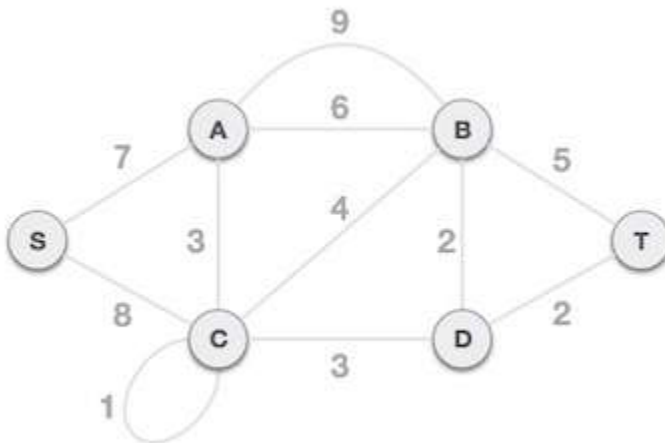
- By adding edge S,A we have included all the nodes of the graph and we now have minimum cost spanning tree.

Prim's Algorithm

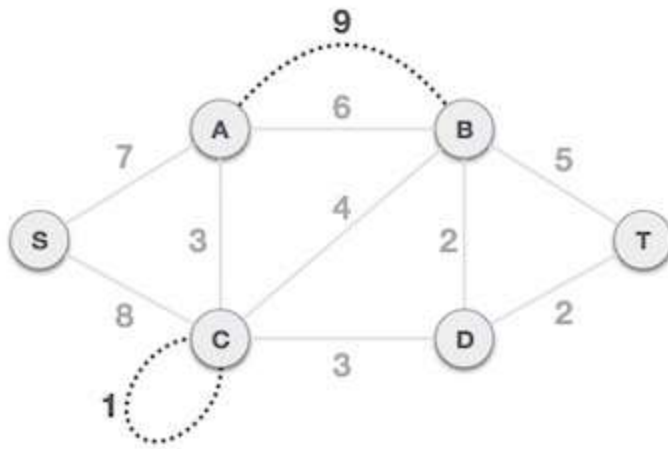
Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach. Prim's algorithm shares a similarity with the **shortest path first** algorithms.

Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.

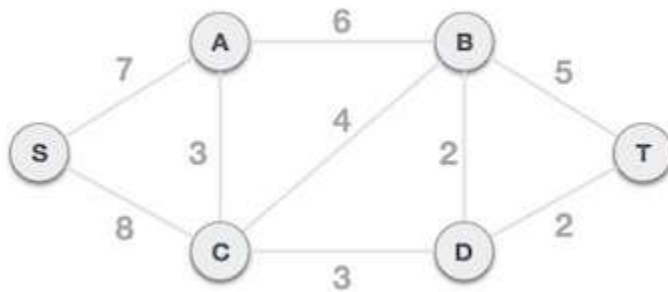
To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example –



Step 1 - Remove all loops and parallel edges



Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

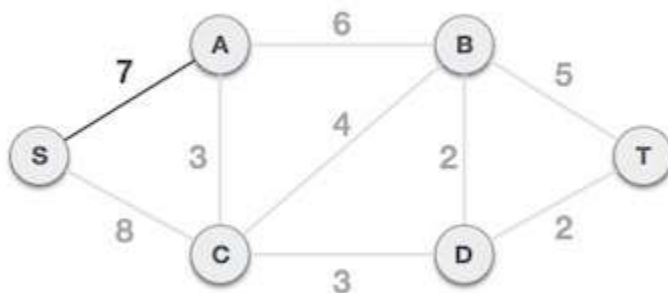


Step 2 - Choose any arbitrary node as root node

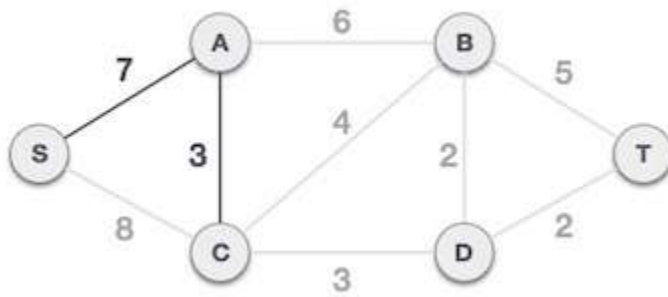
In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any node can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

Step 3 - Check outgoing edges and select the one with less cost

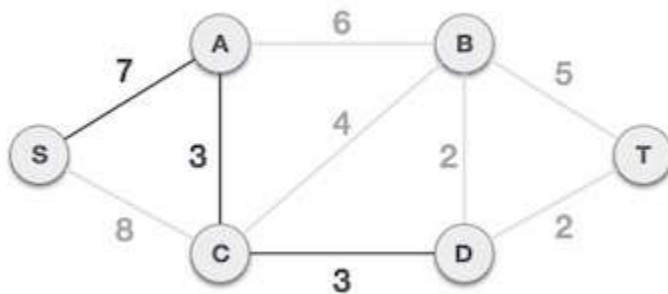
After choosing the root node **S**, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



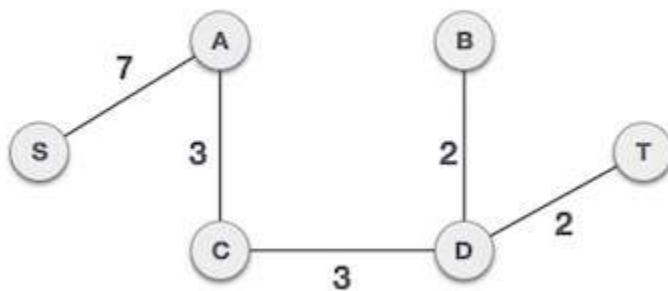
Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



After adding node **D** to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.



UNIT-IV

P and NP Class

In Computer Science, many problems are solved where the objective is to maximize or minimize some values, whereas in other problems we try to find whether there is a solution or not. Hence, the problems can be categorized as follows –

Optimization Problem

Optimization problems are those for which the objective is to maximize or minimize some values. For example,

- Finding the minimum number of colors needed to color a given graph.
- Finding the shortest path between two vertices in a graph.

Decision Problem

There are many problems for which the answer is a Yes or a No. These types of problems are known as **decision problems**. For example,

- Whether a given graph can be colored by only 4-colors.
- Finding Hamiltonian cycle in a graph is not a decision problem, whereas checking a graph is Hamiltonian or not is a decision problem.

What is Language?

Every decision problem can have only two answers, yes or no. Hence, a decision problem may belong to a language if it provides an answer ‘yes’ for a specific input. A language is the totality of inputs for which the answer is Yes. Most of the algorithms discussed in the previous chapters are **polynomial time algorithms**.

For input size n , if worst-case time complexity of an algorithm is $O(n^k)$, where k is a constant, the algorithm is a polynomial time algorithm.

Algorithms such as Matrix Chain Multiplication, Single Source Shortest Path, All Pair Shortest Path, Minimum Spanning Tree, etc. run in polynomial time. However there are many problems, such as traveling salesperson, optimal graph coloring, Hamiltonian cycles, finding the longest path in a graph, and satisfying a Boolean formula, for which no polynomial time algorithms is known. These problems belong to an interesting class of problems, called the **NP-Complete** problems, whose status is unknown.

In this context, we can categorize the problems as follows –

P-Class

The class P consists of those problems that are solvable in polynomial time, i.e. these problems can be solved in time $O(n^k)$ in worst-case, where k is constant.

These problems are called **tractable**, while others are called **intractable or superpolynomial**.

Formally, an algorithm is polynomial time algorithm, if there exists a polynomial $p(n)$ such that the algorithm can solve any instance of size n in a time $O(p(n))$.

Problem requiring $\Omega(n^{50})$ time to solve are essentially intractable for large n . Most known polynomial time algorithm run in time $O(n^k)$ for fairly low value of k .

The advantages in considering the class of polynomial-time algorithms is that all reasonable **deterministic single processor model of computation** can be simulated on each other with at most a polynomial slow-d

NP-Class

The class NP consists of those problems that are verifiable in polynomial time. NP is the class of decision problems for which it is easy to check the correctness of a claimed answer, with the aid of a little extra information. Hence, we aren't asking for a way to find a solution, but only to verify that an alleged solution really is correct.

Every problem in this class can be solved in exponential time using exhaustive search.

P versus NP

Every decision problem that is solvable by a deterministic polynomial time algorithm is also solvable by a polynomial time non-deterministic algorithm.

All problems in P can be solved with polynomial time algorithms, whereas all problems in NP - P are intractable.

It is not known whether $P = NP$. However, many problems are known in NP with the property that if they belong to P, then it can be proved that $P = NP$.

If $P \neq NP$, there are problems in NP that are neither in P nor in NP-Complete.

The problem belongs to class P if it's easy to find a solution for the problem. The problem belongs to NP, if it's easy to check a solution that may have been very tedious to find.

Cook's Theorem

Stephen Cook presented four theorems in his paper "The Complexity of Theorem Proving Procedures". These theorems are stated below. We do understand that many unknown terms are being used in this chapter, but we don't have any scope to discuss everything in detail.

Following are the four theorems by Stephen Cook –

Theorem-1

If a set S of strings is accepted by some non-deterministic Turing machine within polynomial time, then S is P-reducible to {DNF tautologies}.

Theorem-2

The following sets are P-reducible to each other in pairs (and hence each has the same polynomial degree of difficulty): {tautologies}, {DNF tautologies}, D3, {sub-graph pairs}.

Theorem-3

- For any $T_Q(k)$ of type Q , $T_Q(k)k^{1/(\log k)^2}T_Q(k)k(\log k)^2$ is unbounded
- There is a $T_Q(k)$ of type Q such that $T_Q(k) \leq 2k(\log k)^2 T_Q(k) \leq 2k(\log k)^2$

Theorem-4

If the set S of strings is accepted by a non-deterministic machine within time $T(n) = 2^n$, and if $T_Q(k)$ is an honest (i.e. real-time countable) function of type Q , then there is a constant K , so S can be recognized by a deterministic machine within time $T_Q(K8^n)$.

- First, he emphasized the significance of polynomial time reducibility. It means that if we have a polynomial time reduction from one problem to another, this ensures that any polynomial time algorithm from the second problem can be converted into a corresponding polynomial time algorithm for the first problem.
- Second, he focused attention on the class NP of decision problems that can be solved in polynomial time by a non-deterministic computer. Most of the intractable problems belong to this class, NP.
- Third, he proved that one particular problem in NP has the property that every other problem in NP can be polynomially reduced to it. If the satisfiability problem can be solved with a polynomial time algorithm, then every problem in NP can also be solved in polynomial time. If any problem in NP is intractable, then satisfiability problem must be intractable. Thus, satisfiability problem is the hardest problem in NP.
- Fourth, Cook suggested that other problems in NP might share with the satisfiability problem this property of being the hardest member of NP.

UNIT-V

Approximate Algorithms

Introduction:

An Approximate Algorithm is a way of approach **NP-COMPLETENESS** for the optimization problem. This technique does not guarantee the best solution. The goal of an approximation algorithm is to come as close as possible to the optimum value in a reasonable amount of time which is at the most polynomial time. Such algorithms are called approximation algorithm or heuristic algorithm.

- For the traveling salesperson problem, the optimization problem is to find the shortest cycle, and the approximation problem is to find a short cycle.
- For the vertex cover problem, the optimization problem is to find the vertex cover with fewest vertices, and the approximation problem is to find the vertex cover with few vertices.

Performance Ratios

Suppose we work on an optimization problem where every solution carries a cost. An Approximate Algorithm returns a legal solution, but the cost of that legal solution may not be optimal.

For Example, suppose we are considering for a **minimum size vertex-cover (VC)**. An approximate algorithm returns a VC for us, but the size (cost) may not be minimized.

Another Example is we are considering for a **maximum size Independent set (IS)**. An approximate Algorithm returns an IS for us, but the size (cost) may not be maximum. Let C be the cost of the solution returned by an approximate algorithm, and C^* is the cost of the optimal solution.

We say the approximate algorithm has an approximate ratio $P(n)$ for an input size n , where

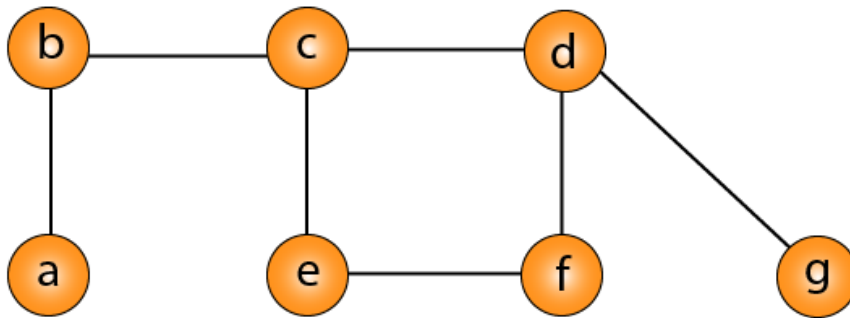
Intuitively, the approximation ratio measures how bad the approximate solution is distinguished with the optimal solution. A large (small) approximation ratio measures the solution is much worse than (more or less the same as) an optimal solution.

Observe that $P(n)$ is always ≥ 1 , if the ratio does not depend on n , we may write P . Therefore, a 1-approximation algorithm gives an optimal solution. Some problems have polynomial-time approximation algorithm with small constant approximate ratios, while others have best-known polynomial time approximation algorithms whose approximate ratios grow with n .

Vertex Cover

A Vertex Cover of a graph G is a set of vertices such that each edge in G is incident to at least one of these vertices.

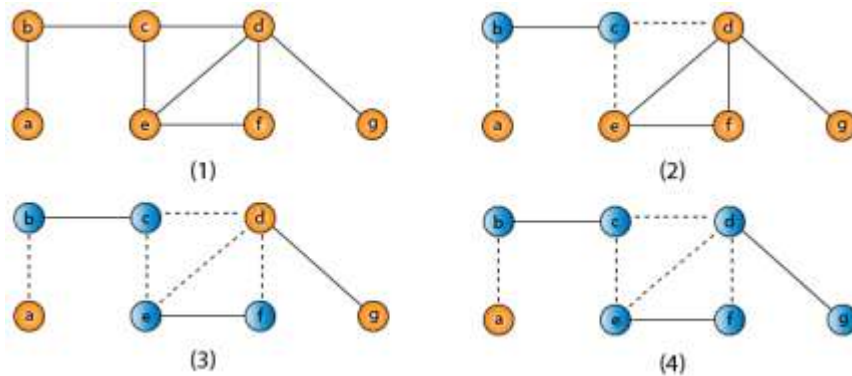
The decision vertex-cover problem was proven NPC. Now, we want to solve the optimal version of the vertex cover problem, i.e., we want to find a minimum size vertex cover of a given graph. We call such vertex cover an optimal vertex cover C^* .



An approximate algorithm for vertex cover:

1. Approx-Vertex-Cover ($G = (V, E)$)
2. {
3. $C = \text{empty-set};$
4. $E' = E;$
5. While E' is not empty **do**
6. {
7. Let (u, v) be any edge in E' : (*)
8. Add u and v to C ;
9. Remove from E' all edges incident to
10. u or v ;
11. }
12. Return C ;
13. }

The idea is to take an edge (u, v) one by one, put both vertices to C , and remove all the edges incident to u or v . We carry on until all edges have been removed. C is a VC. But how good is C ?



$VC = \{b, c, d, e, f, g\}$

What is a Randomized Algorithm?

An algorithm that uses random numbers to decide what to do next anywhere in its logic is called Randomized Algorithm.. For example, in Randomized Quick Sort, we use random number to pick the next pivot (or we randomly shuffle the array). And in Karger's algorithm, we randomly pick an edge.

How to analyse Randomized Algorithms?

Some randomized algorithms have deterministic time complexity. For example, this implementation of Karger's algorithm has time complexity as $O(E)$. Such algorithms are called Monte Carlo Algorithms and are easier to analyse for worstcase.

On the other hand, time complexity of other randomized algorithms (other than Las Vegas) is dependent on value of random variable. Such Randomized algorithms are called Las Vegas Algorithms. These algorithms are typically analysed for expected worst case. To compute expected time taken in worst case, all possible values of the used random variable needs to be considered in worst case and time taken by every possible value needs to be evaluated. Average of all evaluated times is the expected worst case time complexity. Below facts are generally helpful in analysis os such algorithms.

LinearityofExpectation

Expected Number of Trials until Success.

For example consider below a randomized version of QuickSort.

A **Central Pivot** is a pivot that divides the array in such a way that one side has at-least $1/4$ elements.

```
// Sorts an array arr[low..high]
```

```
randQuickSort(arr[], low, high)
```

```
1. If low >= high, then EXIT.
```

```
2. While pivot 'x' is not a Central Pivot.
```

- (i) Choose uniformly at random a number from [low..high].
Let the randomly picked number be **x**.
- (ii) Count elements in arr[low..high] that are smaller than arr[x]. Let this count be **sc**.
- (iii) Count elements in arr[low..high] that are greater than arr[x]. Let this count be **gc**.
- (iv) Let **n** = (high-low+1). If $sc \geq n/4$ and $gc \geq n/4$, then x is a central pivot.

3. Partition arr[low..high] around the pivot x.

4. // Recur for smaller elements
randQuickSort(arr, low, sc-1)

5. // Recur for greater elements
randQuickSort(arr, high-gc+1, high)

The important thing in our analysis is, time taken by step 2 is $O(n)$.

How many times while loop runs before finding a central pivot?

The probability that the randomly chosen element is central pivot is $1/2$.

Therefore, expected number of times the while loop runs is 2 (See this for details)

Thus, the expected time complexity of step 2 is $O(n)$.

What is overall Time Complexity in Worst Case?

In worst case, each partition divides array such that one side has $n/4$ elements and other side has $3n/4$ elements. The worst case height of recursion tree is $\log_{3/4} n$ which is $O(\log n)$.

$$T(n) < T(n/4) + T(3n/4) + O(n)$$

$$T(n) < 2T(3n/4) + O(n)$$

Solution of above recurrence is $O(n \log n)$

Note that the above randomized algorithm is not the best way to implement randomized Quick Sort. The idea here is to simplify the analysis as it is simple to analyse.

Typically, randomized Quick Sort is implemented by randomly picking a pivot (no loop). Or by shuffling array elements.

Possible Questions

Title of the paper : Design and Analysis of Algorithms

Part-A

Answer All Questions (20*1=20)
(Online Examination)

Part-B

Answer ALL Questions (5*16=80)

21. a i) Explain in detail Big oh, Big Omega and Big Theta Notations with necessary illustrations. (8)
ii) Discuss fundamentals of algorithm analysis framework in detail. (8)

(OR)

21. b i) Discuss the important problem types in algorithm solving (8)
ii) Write short notes on fundamentals of algorithmic solving. (8)

22. a i) Using iterative method, find the asymptotic value for (8)

$$T(n) = \begin{cases} 1 & \text{for } n=1 \\ 3T(n/4) + n & \text{for } n>1 \end{cases}$$

- ii) With the help of example explain how a recursive algorithm can be represented by recurrence relation. (8)

(OR)

22. b i) What is empirical analysis of algorithms? Discuss how empirical analysis will be done. (8)
ii) Write short notes on algorithm visualization. (8)

23. a Explain divide and conquer technique in detail. Name some problems which apply divide and conquer techniques for solving it. Discuss binary search algorithm and analyze it. (16)

(OR)

23. b Explain selection sort algorithm with an example and explain the relevance of brute force method in solving it. (16)

- 24 a) Explain Floyd's Algorithm for all pair shortest path algorithm with example and analyze its efficiency

(OR)

- b) Write the Huffman's Algorithm. Construct the Huffman's tree for the following data and obtain its Huffman's Code.

Character	A	B	C	D	E	-
probability	0.5	0.35	0.5	0.1	0.4	0.2

- 25 a) Explain the Assignment problem in Branch and bound with Example. (16)

(OR)

- b) Write backtracking algorithm for N queens problem and Hamiltonian problem. (16)

Department of Computer Science and Engineering

Questions	opt1	opt2	opt3	opt4	Answer
-----------	------	------	------	------	--------

Subject Code : 15BECS405

Name of the Course : II B.E CSE

Title of the paper : Design and Analysis of Algorithms

Semester : IV

Time : 3Hrs

Max Marks : 100 marks

Date :

Part-A

**Answer All Questions (20*1=20)
(Online Examination)**

Part-B

Answer ALL Questions (5*16=80)

21. a i) Explain in detail Big oh, Big Omega and Big Theta Notations with necessary illustrations.(8)
ii)Discuss fundamentals of algorithm analysis framework in detail. (8)

(OR)

21. b i)Discuss the important problem types in algorithm solving (8)
ii) Write short notes on fundamentals of algorithmic solving. (8)

22. a i) Using iterative method, find the asymptotic value for (8)

$$T(n) = \begin{cases} 1 & \text{for } n=1 \\ 3T(n/4) + n & \text{for } n>1 \end{cases}$$

- ii) With the help of example explain how a recursive algorithm can be represented by recurrence relation. (8)

(OR)

22. b i) What is empirical analysis of algorithms? Discuss how empirical analysis will be done. (8)
ii) Write short notes on algorithm visualization. (8)

23. a Explain divide and conquer technique in detail. Name some problems which apply divide and conquer techniques for solving it. Discuss binary search algorithm and analyze it. (16)

(OR)

23. b Explain selection sort algorithm with an example and explain the relevance of brute force method in solving it. (16)

- 24 a) Explain Floyd's Algorithm for all pair shortest path algorithm with example and analyze its efficiency

(OR)

- b) Write the Huffman's Algorithm. Construct the Huffman's tree for the following data and obtain its Huffman's Code.

Character	A	B	C	D	E	-
probability	0.5	0.35	0.5	0.1	0.4	0.2

- 25 a) Explain the Assignment problem in Branch and bound with Example. (16)

(OR)

- b) Write backtracking algorithm for N queens problem and Hamiltonian problem. (16)

.....
Online Questions

The average and worst case complexity of merge sort is	$O(n^2), O(n^2)$	$O(n^2), O(n \log n)$	$O(n \log n), O(n^2)$	$O(n \log n), O(n \log n)$	$O(n \log n), O(n \log n)$
The time taken by binary search to search for an element in sorted array is	$O(n)$	$O(\log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$
Let there be an array of length 'N', and the selection sort algorithm is used to sort it, how many times a swap function is called to complete the execution?	N log N times	log N times	N^2 times	N-1 times	N-1 times
The Sorting method which is used for external sort is	Bubble sort	Quick sort	Merge sort	Radix sort	Merge sort
Which of the following sorting procedure is the slowest?	Quick sort	Heap sort	Shell sort	Bubble sort	Bubble sort
the time required to search an element in a binary search tree having n elements is	$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(\log n)$
the number of comparisons required by binary search of 100000 elements is	15	20	25	30	20
Best case running time of quick sort is	$O(n)$	$O(\log n)$	$O(n \log n)$	$O(n^2)$	$O(n \log n)$
which of the sorting algorithm does not have a worst case running time of $O(n^2)$	Selection sort	insertion sort	merge sort	quick sort	merge sort
A sort which compares adjacent elements in a list and switches where necessary is a	insertion sort	heap sort	quick sort	bubble sort	bubble sort
A characteristic of the data that	Order of the list	length of the list	maximum value	mean of data values	Order of the

binary search tree but the linear search ignores, is the			in the list		list
A sort which uses the binary tree concept such that any number is larger than all the numbers in the subtree below it is called	Selection sort	insertion sort	quick sort	heap sort	heap sort
Worst case complexity of the insertion sort algorithm is	O(n²)	O(n)	O(n-1)	O(n+1)	O(n²)
Average case complexity of the insertion sort algorithm is	O(n²)	O(n)	O(n-1)	O(n+1)	O(n²)
Best case complexity of the insertion sort algorithm is	O(n ²)	O(n)	O(n-1)	O(n+1)	O(n)
Worst case complexity of the bubble sort algorithm is	O(n ³)	O(n ⁴)	O(n²)	O(n)	O(n²)
Best case complexity of the bubble sort algorithm is	O(n ³)	O(n ⁴)	O(n²)	O(n)	O(n²)
Average case complexity of the bubble sort algorithm is	O(n ³)	O(n ⁴)	O(n²)	O(n)	O(n²)
Worst case complexity of the selection sort algorithm is	O(n ³)	O(n ⁴)	O(n²)	O(n)	O(n²)
Average case complexity of the selection sort algorithm is	O(n ³)	O(n ⁴)	O(n²)	O(n)	O(n²)