22CSPC406	DESIGN AND ANALYSIS OF ALGORITHMS	L	T	P	C
		3	0	0	3

Unit-I INTRODUCTION

ALGORITHM:

A finite set of instruction and specified a sequence of operation is to be carried out in order to solve a specific problem or a class of problem

CHARACTERISTICS:

*Input
*Output
*Definiteness
*Finiteness
*Effectiveness
*Uniqueness
*Feasible
*Flexibility
*Efficient
*Independent
*Correctness
*Simplicity

ADVANTAGES:

*Effective communication
*Easy debugging
*Easy and efficient coding
*Independent of programming languages

DISADVANTAGES:

*Developing algorithm for complex problem could be time consuming and difficult to understand
*It is tedious task to understand the algorithm

NEED OF ALGORITHM:

*To understand the basic idea of the problem
*Find an approach to solve the problem
*Efficiency of executing techniques
*To understand the basic principal of designing algorithm.

Computer science is the systematic study of algorithm and data structure specified the common property the mechanical and linguistic realization and application.

ALGORITHMIC THINKING:

Algorithmic thinking is an analytics skill that is required for writing effective to solve the problem.

ALGORITHMICS:

Algorithmic is the art of designing, implementing, analyzing algorithm.

PROBLEMS:

*Computational
*No computational
COMPUTATIONAL PROBLEM:
*Structuring problems and search problems
*Construction problem
*Decision problem
*Optimization problems

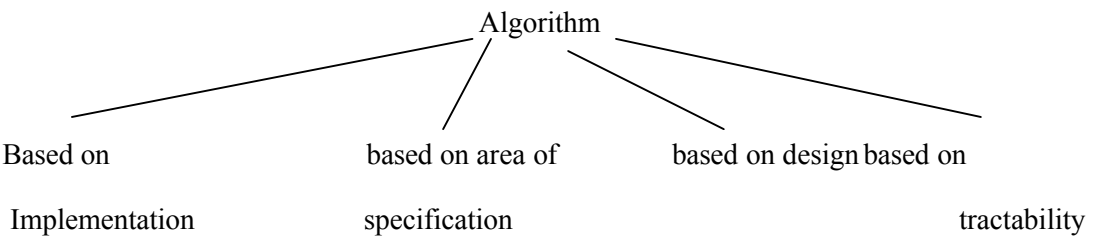
FACTORS:

*Relationship between input and output
*Legal input and correct output

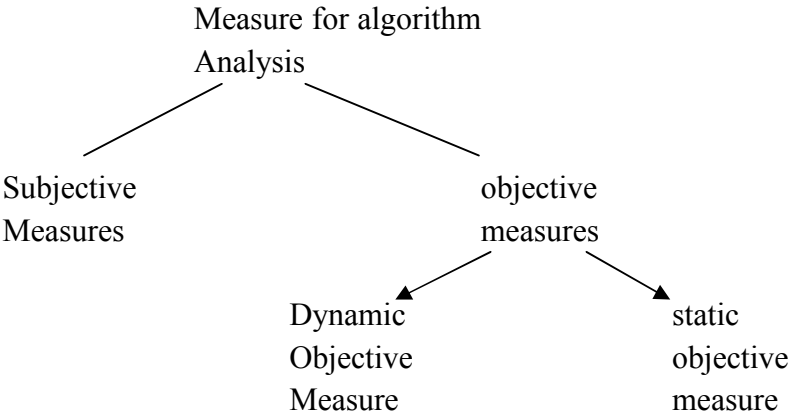
FUNDAMENTAL STAGES OF PROBLEM SOLVING:

1) Understanding the problem
2) Planning an algorithm
3) Designing an algorithm
*A skilled designer algorithmic is called algorists.
*The algorithm can be coded this stage is called algorithm specification
3 Stages**:**
*Natural language
*Pseudo code
*Programming language
4) Validity and verification of algorithm
5) Analysis an algorithm
6) Implementing an algorithm
7) Performing empirical analysis

CLASSIFICATION OF ALGORITHM:



BASICS OF ALGORITHM ANALYSIS:



Analysis of a program involves two factors
*Run time
*Compile time

TIME COMPLEXITY:

Time complexity is a measure of how much run time an algorithm required for executed when the input size is scaled.

CLASS OF FUNCTIONS:

Types of time functions

*Constant-0(1)
*logarithmic-0(log n)
*Linear -0(n)
*Quadratic-0(n2)
*Cubic-0(n3)
*Exponential-0(2^n)

MEASURING RUN TIME:

*STEP COUNT
*OPERATION COUNT
*ASYMPOTIC ANALYSIS
*RECURRENCE RELATIONS
*AMORTIZED ANALYSICS

STEP COUNT:

*Ullman proposed step count
*To determine upper and lower bound.

OPERATION COUNT:

* Knuth proposed operation count.
*To determine worst, best, average

STEP COUNT:

a) Declarative statement

*With no initialization have a statement count of zero. Incase initialization made step count is 1.

b) Comments and brackets

Step count is zero.

C) Expression

Step count is 1.

d) Assignment statement, function return statement and other statement

Step count is 1.

Algorithm test(A,B,C)

Begin

A=A+1;

C=A+2;

D=A+B;

End

Total count is obtained by multiplying the frequency and steps for execution

Step no.	Program	Steps for execution	Frequency	Count
1	Algorithm	0	-	-
2	Begin	0	-	-
3	A=A+1	1	1	1
4	C=A+2	1	1	1
5	D=A+B	1	1	1
6	End	0	-	-
			Total=	3

OPERATION COUNT:

It is no of operation is counted instead step for algorithm analysis the operation can be divide into2 categories.

*Elementary or basic

*Non elementary or non-basic

Some of commonly used basic operation are

*Assignment operation

*Arithmetic operation

*Logical operation

*Compression operation

Non elementary operation involves many elementary operations sorting, finding or maximum or minimum of array etc….

Step required for performing operation:

*Count the no of basic operation of the program and express as a formula

*Simplify the formula

*Represent the time complexity as a function of operation count

RULES FOR FINDING THE OPERATION COUNT FOR AN ALGORITHM:

SEQUENCE:

Begin

S1 required m operation

S2 required n operation

End

SELECTION:

Statement p requires m operation

Statement q requires n operation

The maximum no of operation of either if part or he else part is considered as the operation count

REPETATION:

If a loop execute a task n times and if a task involves m operation then task is m x n this is called multiplication principle.

1)

Algorithm

Step no,	Algorithm segment	Elementary operation accounted for	Operation cost	Repetation
1	Algorithm swap(a,b)	-	-	0
2	Begin	-	-	0
3	Temp=a;	Assignment operation	C1	1
4	A=b;	Assignment operation	C2	1
5	B=temp;	Assignment Operation	C3	1
6	end	-	-	0
		T(n)=	C1+C2+C3	

This is Constant time algorithm.

For loop:

1) For(i=0;i<n;i++){

Statements; ----→time complexity=O(n)

}

2) For(i=n;i>n;i--){
   Statements; ----→time complexity=O(n)

}

3) For(i=0;i<n;i=i+2){

   Statements; ----→time complexity=O(n/2)

   }

4) For(i=0;i<n;i++){ -----n+1
   For(j=0;j<n;j++){ -----n(n+1)
   Statements; -----n*n
   }}
   Time complexity-T(n)=O(n^2)
5) P=0;

   For(i=1;p<=n;i++){

   P=P+i;

   }

i	p
1	0+1=1
2	1+2=3
3	1+2+3=6
.	.
.	.
.	.
k	.

1+2+3+…..+k=(k(k+1))/2

p =(k^2+k)/2

Assume,

p>n

=(k^2+k)/2 >n

=k^2>n

K=√n

O(n)= √n

While:

1) I=0 -----1

While(i<n){ -----n+1

Statement; -----n

I++; -----n

} ----------

3n+2

Time complexity=O(n)

2)a=1; ----1

While(a>b){ ----n+1

Statement; ----n

A=a*2; ----O($\log_2 n$)

}

Time complexity=O($\log_2 b$)

3)i=1;
While(i>1){
Statement;
i=i/2;
}
Time complexity=O(log n)

If:

1)While(m!=n){

If(m>n)

M=m-n;pl

Else

M=n-m

}

Time complexity=O(n)

2)algorithm test(n){

If(n<5){

Printf(n);

}

Else{

For(i=0;i<n;i++){

Printf(n);

}}}

Best case time complexity=O(1)

Worst case time complexity=O(n)

BEST, WORST, AVERAGE CASE COMPLEXITY:

Worst, best, average case efficient of algorithm can be estimated by considering the different distribution of input data.

WORST CASE COMPLEXITY AND UPPER BOUND:

It is defined t(n) has the complexity function w(n) of the worst case input for which the algorithm takes maximum time and also cause a computer to run lower.

W (n) =max

Example: w (n) =0(n) linear search

BEST CASE COMPLEXITY AND LOWER BOUND:

The best case of analysis give the minimal computation of algorithm for all the validity input of the algorithm

B (n) =min

Example: b (n) =1 linear search

AVERAGE CASE COMPLEXITY:

Average case analysis assume that input is a random and provide predation about the running time of the algorithm for random input

BINARY SEARCH TREE

*Best case-searching root elements

*Worst case-searching leaf elements

ASYMPTOTIC ANALYSIS:

*It is analysis of a given algorithm with larger value of input data. it is theory of approximation.

*It can be very effective for algorithm analysis for finding exact time complexity is difficult for process of counting.

ASYMPOTIC NOTATIONS:

This are helpful in classify algorithm and also specify upper and lower bound of an algorithm. All time functions are of the form t:n->r the functions t evaluates only for integral natural number and r is the positive real number.

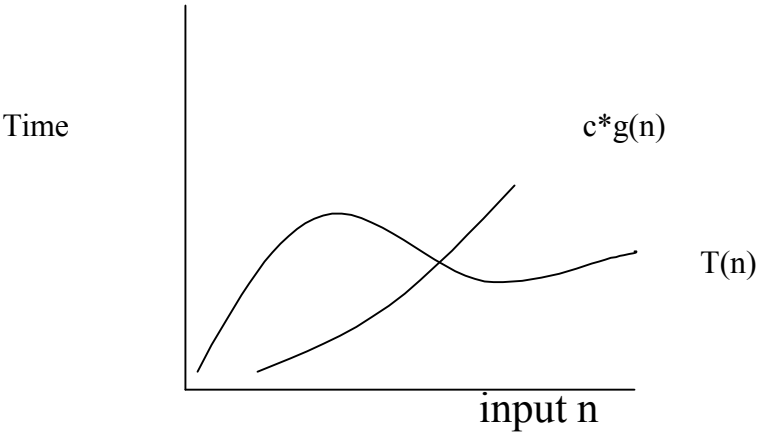
BIG OH NOTATION:

It can be used in following incident for expressing upper bound or worst case algorithm while expressing time complexity is almost condition.

Let g and h be a set of two functions that map a set of two natural numbers through a set of positive numbers t=n;R>=0 let o(g) is a similar rate of growth with relation t(n)=O(g(n)) holds true if there exits two positive constants c and n_0 such that t(n)<=c* g(n)

The function t(n) is said to be O(big oh) of g(n). this is denoted as t(n) ε O(g(n)) or t(n)=O(g(n))

Then implies that t(n) is said to be O(big oh) of g(n) .then approximately g(n) function (i.e) a function with a growth rate less than or equal to that of g(n) this implies that t grows at a slower rate than a constant time g(n) for the all values of larger input of size N.



example:

t(n)=2n+3

2n+3 <= 10 n n=1

T(n)<=O(g(n))

2n+3<=2n+3n

2n+3<=5n

T(n)εO(g(n))

$2n+3<=2n^2+3n^2$

$T(n)=O(n^2)$

BIG OMEGA NOTATION:

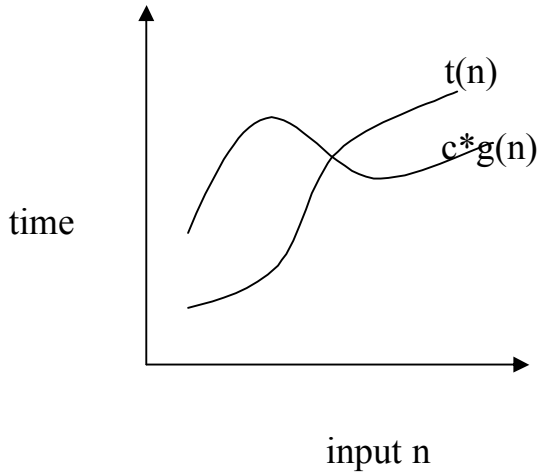
It can be used in following incident for expressing lower bound or best case algorithm while expressing time complexity is almost condition.

Definition:

Let t and g be two function that map a of set of natural number through a set of position realnumbers t=N->R<=0. Let P(g(n)) holds true if there exists two positive constants c and n_0 such that t(n)>=c*g(n).

The function t(n) is said to be Ωof g(n).this is denoted as t(n)εΩ(g(n)) or t(n) =Ω(g(n))

This implies that t(n) never takes more than approximately g(n) function (i.e) a function with growth rate constant time that of g(n).



example:

t(n)=2n+3

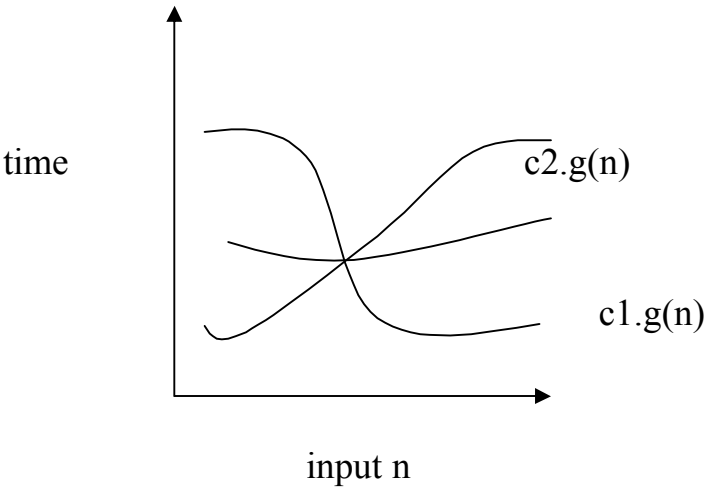
2n+3>=1*n

2n+3>=1*log n

Big theta notation:

This notation gives the both upper bound and lower bound of algorithm . let t and g two function that map a set of natural numbers to a set of positive real numbers .let the relationship c1.g(n)<=f(n)<=c2.g(n) for all n>=n_0 and for constants c1 and c2 then it can be represented as t(n)= Θ(g(n))

That is t(n) grows at a same rate of constants time g(n) for sufficiently large value of n.



example:

$t(n)=n^2\log n+n$

$1*n^2\log n<=n^2\log n*n=10n^2\log n$

$\Theta(n^2\log n)$

Little oh notation

It can be used instead of big oh notation as a little oh notation represents a lower bound the relation $t(n)=o(g(n))$ holds good if the two positive constants c and n_0 such that $t(n)<c*g(n)$

Little omega notation:

The relation $t(n)=\omega(g(n))$ holds good if the two positive constant c and n_0 such that $t_n>c_x\ g(n)$

Tilde:

The notation helpful for when function t(n) and g(n) grows at the same rate.

Name of the notation	What it means	In terms of limit	How it is represented	Equivalent to
O(big oh)	Growth of t(n) is <=the growth of g(n)	$\lim_{n\to\infty} t(n)/g(n)=c$ $c>=0$	T(n)=O(g(n))	$\leq$
Ω(big omega)	Growth of t(n)	Lim n to ∞	T(n)= Ω(g(n))	$\geq$

	is>=the growth of g(n)	$t(n)/g(n) \neq 0$		
Θ(theta)	Growth of t(n) is >= the growth of g(n)	Lim n to ∞ $t(n)/g(n)=c_1$ c>0	T(n)= Θ(g(n))	≃
o(little oh)	Growth of t(n) is << the growth of g(n)	Lim n to ∞ t(n)/g(n)= 0	T(n)=o(g(n))	<
ω(little omega)	Growth of t(n) is >> the growth of g(n)	Lim n to ∞ t(n)/g(n)=∞	T(n)=ω(g(n))	>
~(tilde)	Growth of t(n) is = to the growth of g(n)	Lim n to ∞ t(n)/g(n)=1	T(n)~g(n)	=

ASYMPOTIC RULES:

*Reflectivity rule
*Transitive rule
*Law of composition
*Multiplication rule
*Law of addition

Reflexivity rule:

For any complexity function g(n) the reflexivity property is given as t(n)=O(g(n)), t(n)= Ω (g(n)),t(n)= Θ(g(n))

Transitive rule:

if t(n)=O(g(n)) and g(n)=O(h(n)) then transitive rule defines as t(n)=O(h(n))

Law of composition:

O(O(t(n)))=O(t(n))

Law of addition:

Assume that the algorithm a is written in such a way that some portion of have complexity n,n^2,log n and some have n+log n+n^2. The law of addition states the following.

T(n)+g(n)=O(max(t(n),g(n)))

T(n)+g(n)= Ω (max(t(n),g(n)))

T(n)+g(n)= Θ (max(t(n),g(n)))

Multiplication rule:

For i=1 to n do ←-------execute n+1 times

Perform operation O(1)←----$^{i}_{\text{execute n time}}$

End for

O(1)=)(n)

If there are two loops the inner loop could be executed n+1 times this is called the multiplication rule.

SPACE COMPLEXITY ANALYSIS:

Space analysis fixed components and variable port fixed components is defined as portion of memory that are independent of input output.

ANALYSIS OF RECURSIVE ALGORITHM THROUGH RECURRENCE RELATIONS:

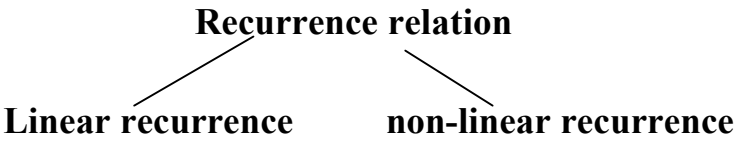
*To analysis the recursive algorithm.

*Recurrence equations defines a sequence using the elements of the sequence .a sequence is a finite or infinite list of no's.

*The recurrence relation is basically definition of a function I in terms of itself

*The recurrence equation are difference equation is a descript equivalent of a differential equation that express a terms of sequence as a function of residing terms.

CLASSSIFICATION:



LINEAR RECURRENCE:

- A linear recurrence equation for a sequence {t0,t1,t2,…..tn} express the final terms tn as a linear combination of its terms in a polynomial form.
- The recurrence equation of Fibonacci seriescan be represented as $tn=t_{n-1}+t_{n-2}$
- In general the recurrence equation is $a_{0=}\ t_n+t_{n+1}+\ldots..a_kt_{n-k}=t(n)$.where k and ai terms are constant k be the order of recurrence equation.

Types:

*based on order

*based on co efficient

*based on homogeneity

Order of recurrence equation:

*The number of residing terms used for computing the present terms of a sequence is called the order of recurrence equation.

*the order is difference between the highest be lowest subscript of dependent variable in recurrence equation.

Example:

$T_n-T_{n-1}-T_{n-2}=0$

The order is n-(n-2)=2

First order-factorial number

$T_n=t_{n-1}+1$

Second order-general order

$F(x)=a_0t_{n-1}+a_1t_{n-1}+a_2t_{n-2}$

Homogenous vs non homogenous:

Consider this function $f(n)=a_0t_{n-1}+\ldots\ldots a_nt_{n-k}$

If f(n)=0 then it is called as homogenous equation.

Example:

$T_n=t_{n-1}+t_{n-2}$

If f(n) !=0 then it is called as non homogenous equation.

constant vs variable coefficient:

$f(n)=a_0t_0+a_1t_{n-1}+\ldots a_nt_{n-k}$

in above equation the a_i may be constant on variable.

NON LINEAR RECURRENCE:

It depends mainly on divided & conquer method.

MEHODS FOR SOLVING RECURRENCE EQUATION:

*Guess and verify method
*Substitution method
*Recurrence tree method
*Difference method
*Polynomial reduction
*Generating function
*Table loop up method/master theorem method.

MASTER THEOREM:

Let the time complexity function be the positive and eventually a non- decreasing function of following form

T(n)

$T(n)=at(n/b)+cn^k$

T(1)=d

Where d, a, k and b are all constants and $b \geq 2$, $k \geq 0$, $a > 0$, $c > 0$ and $d \geq 0$.The solution for the recurrence equation is given as follows:

Case 1: $t(n) \in \Theta(n^k)$ if $a < b^k$

Case 2: $t(n) \in \Theta(n^k \log n)$ if $a = b^k$

Case 3: $t(n) \in \Theta(n \log_b a)$ if $a > b^k$

Example;

$T(n)=8t(n/2)+n^2$

$At(n/2)+cn^k$

A=8,b2,c=1 and k=2

There conditions are

1. $A=b^k$
2. $A>b^k$
3. $A<b^2$

RECURSION TREE[SUBSTITUTION METHOD AND TREE METHOD]:

PROBLEMS

Substitution method

a) $T(n)=T(n-1)+\log n$

Subs T(n-1)

$T(n-1)=T(n-2)+\log(n-10$

$T(n)=[T(n-3)+\log (n-2)+\log(n-1)]+\log n$

.

.

.

$T(n-k)+\log(n-(k-1))+\log(n=-(k-2))+\log(n-(k-3))+\log n$

Let n-k=1

K=n

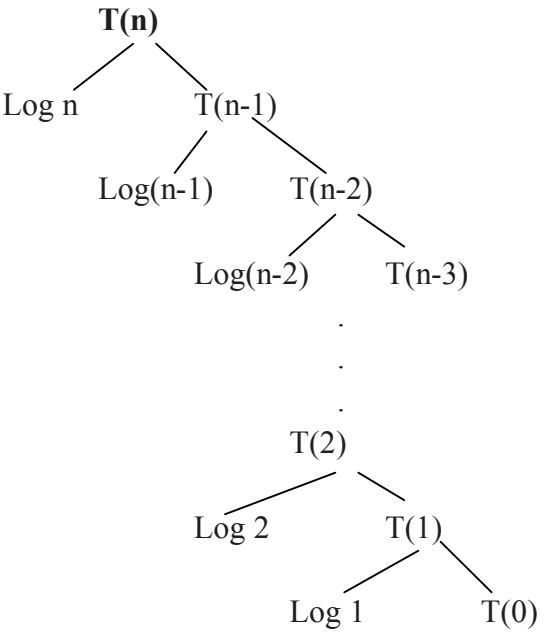
$1+\log 1+\log 2+\log 3+\ldots+\log n$

$1+\log(n!)$

$1+\log n^n$

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

Tree method



Log n+log (n-1)+…..log2+log 1
Log[n(n+1)….2*1]
Log n!
O(n log n)=T(n)

2)**T(n)={1 ,n=1;**
2T(n/2) ,n>1

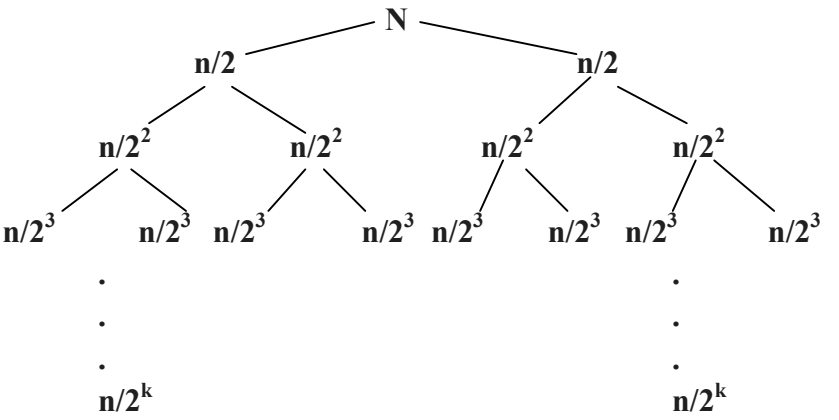
Solution:

Substitution method

Subs T(n/2)

$T(n)=2(2T(n/2^2)+n/2)+n$
$T(n)=2^2T(n/2^2)+n+n$
Subs $T(n/2^2)$
$T(n)=2^2[2T(n/2^2)+n/2^2]+2n$
$2^3T(n/2^3)+3n$
.
.
$T(n)=2^kT(n/2^k)+kn$
Assume $T(n/2^{k)}=T(1)$
$n/2^k=1$
$n=2^k$

k=log n
$T(n)=2^kT(1)+kn$
n*1+n log n
T(n)=O(n log n)
Tree method



assume $n/2^k=1$
$n-2^k=k$
k=log n

UNIT-2

FUNDAMENTAL ALGORITHMIC STRATEGIES

Brute Force Approach

This is approach is a direct and straight forward techniuqe of a problem solving in which all the possible solutio of a given problem are enumerated.We solve a many problems in life using Brute force Approach.

EXAMPLES:

1.Exploring all the park to a nearby market to find the shortest path.

2.Arranging book in a reach using all possibilities to optimize the book rack

3.Suitcase password checking.

ADVANTAGE:

- This is the directed way to find the correct solution by listing all the possible solution of a problem.
- Brute force Approach is ideal for solving smaller and similar problem.It can serve as a comparison bench mark.
- It is a generic method because it is not linked to any specific domain of problem.

DISADVANTAGE:

- Brute force algorithm are slow.
- Brute force method is inefficient.
- This method depend on the computing power of computer system for solving problem on a good algorithm design.
- Brute force algorithm are not creative compared to algorithm that are constructed using some other design methods.

Greedy Approach

The aim of optimization problem is to find a but solution from all feasible solution.Optimization method are used in stages.At every stage a decision are choice in made this decision are locally optimal solution.Finally,the global optimal solution by combining locally optimal solution (i.e it will start with empty set)

COMPONENTS OF GREEDY ALGORITHM:

- Objective funtion
- Generating multiple candidate solution
- Selection procedure
- Feusibility check
- Solution check

1.OBJECTIVE FUNCTION:

- **It should be either maximized or minimized based on the given problem.**

2.GENERATING MULTIPLE CANDIDATE SOLUTION:

- **A Greedy problem may have n inputs or candidate solution.All possible solution may not be optimal solution.Hence it has to be check whether the candidate solution fulfill the constraints if so they are selected as a possible solution.**

3.SELECTION PROCEDURE:

- A solution procedure must exists for a greedy algorihm to choose the next algorithm.Selection must be done based on some greedy criteria.

4.FEUSIBLITY CHECK:

- **It determine the selected item is feasible as per the constraints.**

5.SOLUTION CHECK:

- **This check whether the partial solution together construct a global solution for the given problem and if so solution is return.**

ALGORITHM:

%%Input:ArrayA[…….n]

%%Output:Solution of a problem

Solution set=NULL

While(solution is not complete)

do

Select a best candidate solution X % Selection procedure

If X is a feasible solution then

%%Feasible if

%%Constraints are satisfied

Add the solution ‘X’ to the solution set

End if

%%Check if the solution of the given problem is obtained if(solution obtained)then %% Solution check return solution set

End if

End white

End

DYNAMIC PROGRAMMING

Dynamic proramming is useful for solving the multistage optimization.A problem is divide and problem into sub problem and establish the accursing problem.A sub problem trait represent the all part of the original problem which is solved for obtaining the optimal solution.This process of enlargement repeated till the age of the sub problem numbers the whole original problem that is solution for the whole problem is obtained by combining of the optimal solution of the subproblem.

COMPONENTS OF DYNAMIC PROGRAMMING:

- Stages
- State
- Decision
- Policy

DECISION:

In every stage there can be multiple decision out of which the best solution will be taken.That is decision taken at every stage should be called stage variable.

STATE:

A state indicate the sub problem which the decision needs to be taken.The variable that are used to taken on decision at every stage.That are called state variable.

POLICY:

A policy is a rule that determined the decision at each stage.A policy is called optimal policy if it is globally optimal.This is called Bellman's principal of optimality.

Branch and bound technique

Branch and Bound technique uses the state space tree fo solving the problem.It is used for solving the optimisationproblem.So in this technique we have two steps used to solve the problem.

1.Branching

2.Bounding

It is the first step in which involves division of a given problem into two or more subproblem.The subproblems are similar to the original problem but smaller in sizethe operation that are applied for the original problem are applied to the sub problemAssume that f(x) is the function sub problem and S is the state space tree that has all solving.Hence it is set S is called teasibleregion.Hence set can be divide into k region such that the union of all Si use back Si

The division of staete space based on the constraints associated with the given problem.Thesecod step is called the boundig step which helps in limiting the growth of the state space.

BACKTRACKING

Backtracking is the systematic method for seraching one or more solution for a given problem.Its is a refind brute force approach used for solving problem.It can reaffctively solve multidecision problem where the final solution is visualized as set of divisions/choices.The execution of decision/choices leads another set of decision.This decision can be followed till one encounters a successful solution.It solves three kinds problem

1.Enumeration problem

2.Decision problem

3.Optimisation problem

In back tracking the constraints of the given problem is given by the bounding function.Back tracking process defines a solution vector as n tuple vector for the given problem where n is the number of components of solution vector and each excite represents a partial solution.This partial solution components are generated based on the concepts of constraints.In backtracking two types of constraints they are:

1.Implicit

2.Explicit

1.IMPLICIT:

- Implicit constarints are rules that limit the processing of solution vector that maximize or satisy the craterianfunction.Thecarterian function is called a Bounding/Validity promising function.

2.EXPLICIT:

- Explicit constraints are rules that resctrict the components of the solution vector Xi form chosing specific value from a set S.

Knapsack problem

The kanpsack problem has knapsack of capacity K there are n different items each of which is associated with a weight Wi and profit Pi.

The objective of this problem is to load a knapack with as many as item as possible subjected to the capacity of the knapsack to get the maximum profit.

TYPES OF KNAPSACK PROBLEM:

1.Fractional knapsack problem

2.Integer knapsack problem

ALGORITHM:

i%%Input i times with profit

%%Output:Optimal packing order of itmes stored in solution vector

Begin

%%Intialize the solution vector

For i<=1 to n do

x(i):=0.0;

End for

Load=0 %%Intialise weight of knapsack i

i=1 %%start with the first size

```
while((load<w)and (i<=n))do
if((wi+load)<=w)then
   load=load+wi;      %%load item fully
   x(i)=1;              %%mark in the solution vector that the item is loaded fully.
Else
   r=w=load            %%compute the space left out
load=load+r/wi         %%fit knapsack with fraction of items
x(i)=r/w(i)            %%record the amountof items in solution vector
end if
end while
return(x)              %%Return vector solution
End
```

KNAPSACK PROBLEM USING GREEDY APPROACH

1)

Items	1	2	3
weight	14	18	10
profit	24	20	16

M=0

Solution:

p/w	1.7	1.1	1.6

N=3,m=20
X=(x1,x2,x3)
X1=20-14=6
X3=6/10
X=(1,0,6/10)
Total weight=$\sum_{i=1}^{3} x_i w_i$
=$x_1w_1+x_2w_2+x_3w_3$
=14*1 + 0*18 +6/10*10

=14+0+6

Total weight=20

Total profit=$\sum^{n}_{i=1} x_i p_i$

=$x_1p_1+x_2p_2+x_3p_3$

=1*24 + 0*20 + 6/10*16

=24+9.6

Total profit=33.6

Knapsack problem using dynamic programming:

items	1	2	3
weight	1	2	4
value	1	6	4

M=3

Solution:

p	w		0	1	2	3
-	-	0	0	0	0	0
1	1	1	0	1	1	1
6	2	2	0	1	6	7
4	4	3	0	1	6	-

$\mathbf{V}_{(I,w)}=\max\{v_{[i-1,w]},v_{[i-1,w-w(i)]}+p_i\}$

$V_{[1,1]}=\max\{v_{[0,1]},v_{[0,0]}+1\}$

=max{0,1}

$V_{[1,1]}=1$

$V_{[1,2]}=\max\{v_{[0,2]},v_{[0,1]}+1\}$

=max{0,1}

$V_{[1,2]}=1$

$V_{[1,3]}=\max\{v_{[0,3]},v_{[0,2]}+1\}$

=max{0,1}

$V_{[1,3]}=1$

$V_{[2,1]}=\max\{v_{[1,1]},v_{[1,-1]}+6\}$

=max{1,doesnot exists}

$V_{[2,1]}=1$

$V_{[2,2]}=\max\{v_{[1,2]},v_{[1,0]}+6\}$

=max{1,0+6}

$V_{[2,2]}=6$

$V_{[2,3]}=\max\{v_{[1,3]},v_{[1,1]}+6\}$

=max{1,1+6}

$V_{[2,3]}=7$

$V_{[3,1]}=\max\{v_{[2,1]},v_{[2,-3]}+4\}$

=max{1,doesnot exists}

$V_{[3,1]}=1$

$V_{[3,2]}=\max\{v_{[2,2]},v_{[2,-2]}+4\}$
$=\max\{6,\text{doesnot exists}\}$
$V_{[3,2]}=6$
$V_{[3,3]}=\max\{v_{[2,3]},v_{[2,-1]}+4\}$
$=\max\{7,\text{doesnot exists}\}$
$V_{[3,3]}=7$
When i=3,j=3
K[3][3]==k[2][3]
7==7 true (0)
When i=2,j=3
K[2][3]=k[1][3]
7==1 false
When i=1,j=1
K[1][1]=k[0][1]
1==0 false
X={1,1,0}

TRAVELLING SALESPERSON PROBLEM:

The travelling salesman problem is a graph computational problem where the salesman needs to visit all cities (represented using nodes in a graph) in a list just once and the distances (represented using edges in the graph) between all these cities are known. The solution that is needed to be found for this problem is the shortest possible route in which the salesman visits all the cities and returns to the origin city.

ALGORITHM:

- Travelling salesman problem takes a graph G {V, E} as an input and declare another graph as the output (say G') which will record the path the salesman is going to take from one node to another.
- The algorithm begins by sorting all the edges in the input graph G from the least distance to the largest distance.
- The first edge selected is the edge with least distance, and one of the two vertices (say A and B) being the origin node (say A).
- Then among the adjacent edges of the node other than the origin node (B), find the least cost edge and add it onto the output graph.
- Continue the process with further nodes making sure there are no cycles in the output graph and the path reaches back to the origin node A.
- However, if the origin is mentioned in the given problem, then the solution must always start from that node only. Let us look at some example problems to understand this better.

TRAVELLING SALES PERSON USING DYNAIMMIC PROGRAMMING APPROACH:

a)

	1	2	3	4
1	0	5	3	10
2	2	0	5	7
3	4	3	0	8
4	6	5	9	0

Solution:
When |s|=Φ
Cost(2, Φ)=d[2,1]=2
Cost(3, Φ)=d[3,1]=4
Cost(4, Φ)=d[4,1]=6
When |s|=1
Cost(2,{3})=d[2,3]+cost(3, Φ)=5+4=9
Cost(2,{4})=d[2,4]+cost(4, Φ)=7+6=13
Cost(3,{2})=d[3,2]+cost(2, Φ)=3+2=5
Cost(3,{4})=d[3,4]+cost(4, Φ)=8+6=14
Cost(4,{2})=d[4,2]+cost(2, Φ)=5+2=7
Cost(4,{3})=d[4,3]+cost(3, Φ)=9+4=13
When |s|=2
Cost(2,{3,4})=min{d[2,3]+cost(3,{4}),d[2,4]cost(4,{3})}
=min{5+14,7+13}
=min{19,20}
=19
Cost(3,{2,4})=min{d[3,2]+cost(2,{4}),d[3,4]+cost(4,{2})}
=min{3+13,8+7}
=min{16,15}=15
Cost(4,{2,3})=min{d[4,2]+cost(2,{3}),d[4,3]+cost(3,{2})}
=min{5+9,9+5}
=min{14,14}
=14
When |s|=3
Cost(1,{2,3,4})=min{d[1,2]+cost[2,{3,4}],d[1,3]+cost[3,{2,4}],d[1,4]+cost[4,{2,3}]}
=min{5+19,3+15,10+14}
=min{24,18,24}
=18
1—3—4—2—1

TRAVELLING SALES PERSON PROBLEM USING BRANCH AND BOUND TECHNIQUE:

	1	**2**	**3**
1	∞	**4**	**2**
2	**3**	∞	**4**
3	**1**	**8**	∞

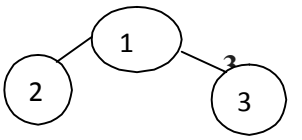
SOLUTION:

Row reduction

∞	**4**	**2**	2
3	∞	**4**	**3**
1	**8**	∞	**1**

6

∞	**2**	**0**	
0	∞	**1**	
0	**7**	∞	
0	**2**	**9**	**=2**



∞	**0**	**0**
0	∞	**1**
0	**5**	∞

total cost D_F reduction

6+2=8

Find cost from(1 to 2)

Make 1st row and 2nd column as infinity and (2,1) as infinity

	1	2	3
1	∞	∞	∞
2	∞	∞	**1**
3	**0**	∞	∞

1

C(1,2)+r+r^

=0+8+1

=9

Find cost from (1 to 3)

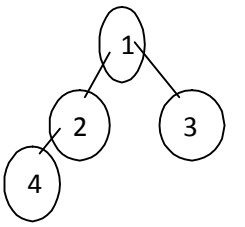
Make 1st row and 3rd column as infinity and (3,1) as infinity.

	1	2	3
1	∞	∞	∞
2	0	∞	∞
3	∞	**5**	∞

----

5 **9** **13**

=c(1,3)+r+r^
=0+8+5=13

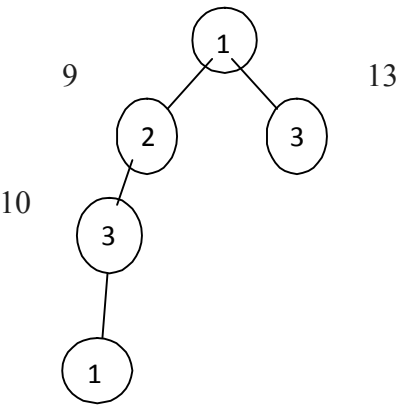


Find the cost from (2 to 3)
Make 2nd row and 3rd column as infinity and (3,2) as infinity.

	1	2	3
1	∞	**0**	∞
2	∞	∞	∞
3	0	∞	∞

=c(2,3)+r+r^
=1+9+0=10

1—2—3—1



N Queen Problem

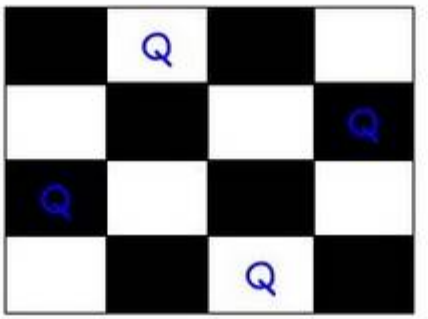
Given an integer **n**, the task is to find the solution to the **n-queens problem**, where **n** queens are placed on an **n*n** chessboard such that no two queens can attack each other.

What is N Queen Problem?

In **N-Queen problem**, we are given an **NxN** chessboard and we have to place **N** number of queens on the board in such a way that no two queens attack each other. A queen will attack another queen if it is placed in horizontal, vertical or diagonal points in its way. The most popular approach for solving the N Queen puzzle is Backtracking.

Input Output Scenario

Suppose the given chessboard is of size 4x4 and we have to arrange exactly 4 queens in it. The solution arrangement is shown in the figure below −



The final solution matrix will be −

0 0 1 0

1 0 0 0

0 0 0 1

0 1 0 0

Backtracking Approach to solve N Queens Problem

In the naive method to solve n queen problem, the algorithm generates all possible solutions. Then, it explores all of the solutions one by one. If a generated solution satisfies the constraint of the problem, it prints that solution.

Follow the below steps to solve n queen problem using the backtracking approach −

- Place the first queen in the top-left cell of the chessboard.
- After placing a queen in the first cell, mark the position as a part of the solution and then recursively check if this will lead to a solution.
- Now, if placing the queen doesn't lead to a solution. Then go to the first step and place queens in other cells. Repeat until all cells are tried.
- If placing queen returns a lead to solution return TRUE.
- If all queens are placed return TRUE.
- If all rows are tried and no solution is found, return FALSE.

UNIT III

GRAPH AND TREE ALGORITHMS

Traversal Algorithms :

Traversal algorithms are methods used to visit and explore all the nodes (vertices) in a data structure such as a graph or a tree. These algorithms determine the order in which the nodes are visited and processed. The two most common traversal algorithms are:

- Depth First Search [DFS]
- Breadth First Search [BFS]

Traversal algorithms are fundamental in graph theory and are essential for various applications such as searching, pathfinding, and data analysis in computer science and other fields. The choice of traversal algorithm depends on the specific problem and the characteristics of the data structure being traversed.

Depth First Search [DFS] :

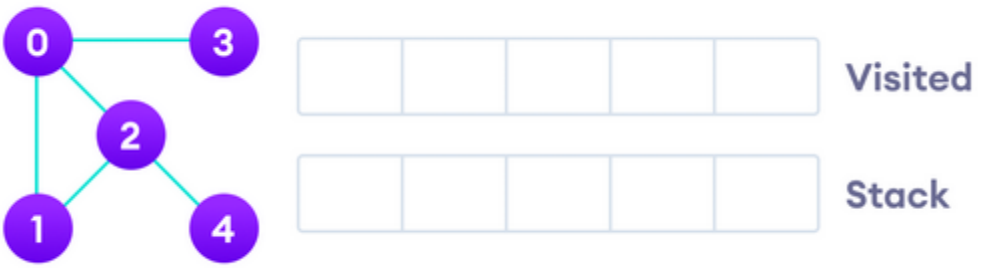
Depth-First Search (DFS) is a graph traversal algorithm that explores a graph or a tree data structure by going as deep as possible along each branch before backtracking. It starts at the root node and keeps exploring as deep as it can along a particular path until it reaches a leaf node or a node with no unvisited neighbors.

When DFS reaches a dead-end, it backtracks to the most recent node with unexplored branches and continues the exploration from there. This process continues until all nodes have been visited.

DFS can be implemented using recursion or an explicit stack data structure. It is useful for tasks like searching for a path between two nodes, exploring all possible paths in a graph, and finding connected components in a graph

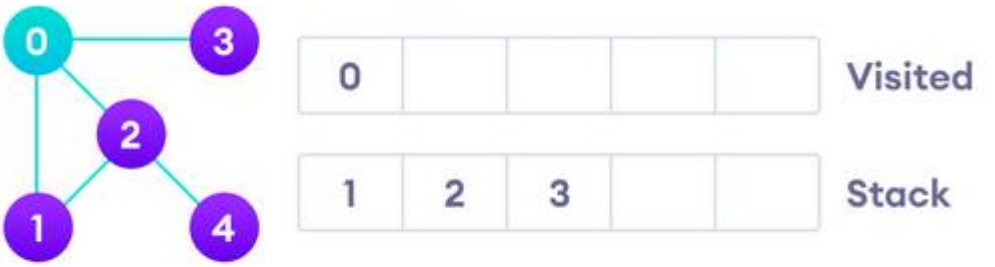
Example :

We use an undirected graph with 5 vertices



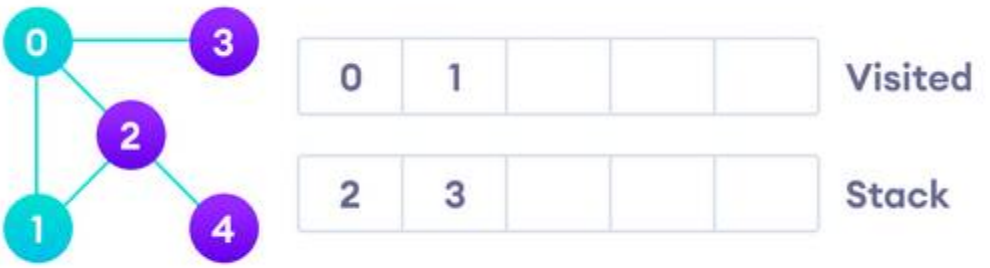
Undirected graph with 5 vertices

We start from vertex 0, the DFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



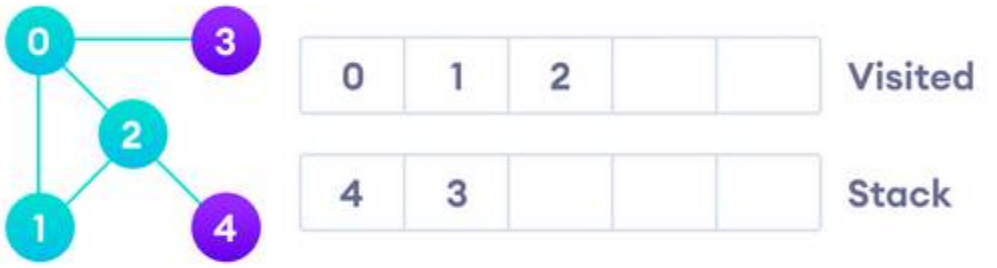
Visit the element and put it in the visited list

Next, we visit the element at the top of stack i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.

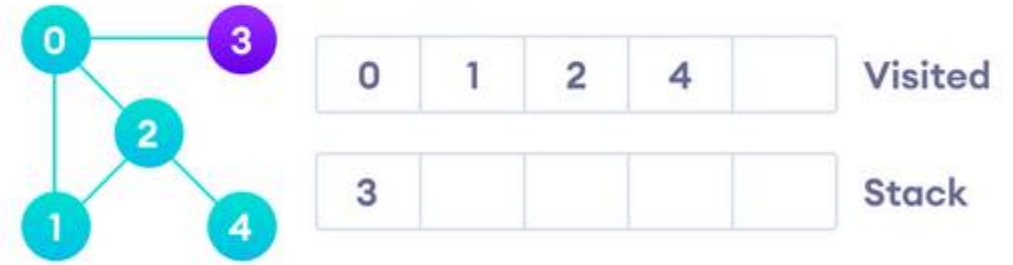


Visit the element at the top of stack

Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.

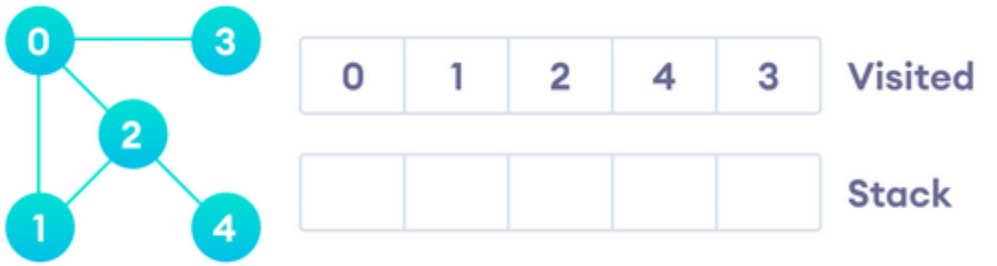


Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.



Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.

After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.



After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.

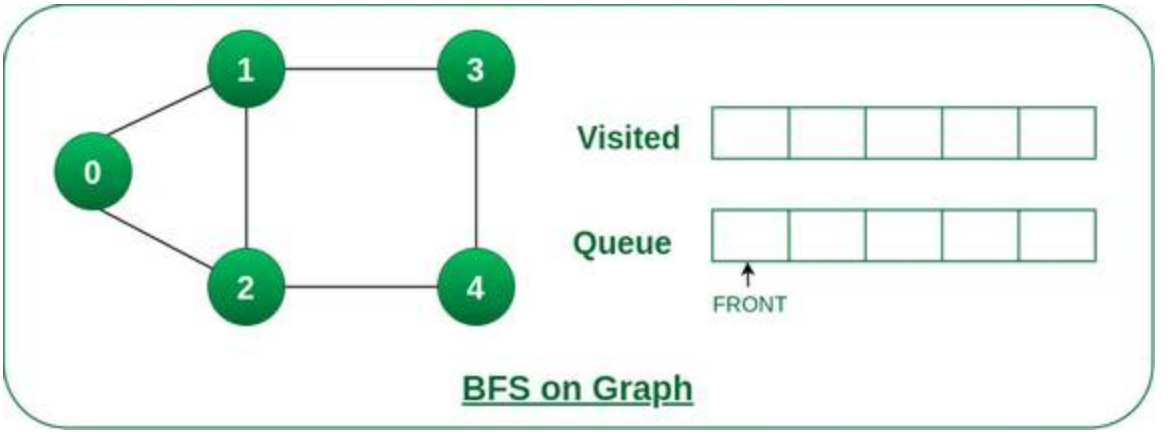
Breadth First Search [BFS] :

Breadth-First Search (BFS) is a way to explore a place step by step, starting from the center and moving outwards level by level. It's like searching for something by looking at nearby things first before checking farther ones.

BFS is often used to find the shortest path from one point to another in a map or a graph. It guarantees that you will find the shortest route by exploring the closest options before checking the ones that are farther away. It helps you search efficiently by exploring nearby areas before moving on to more distant ones.

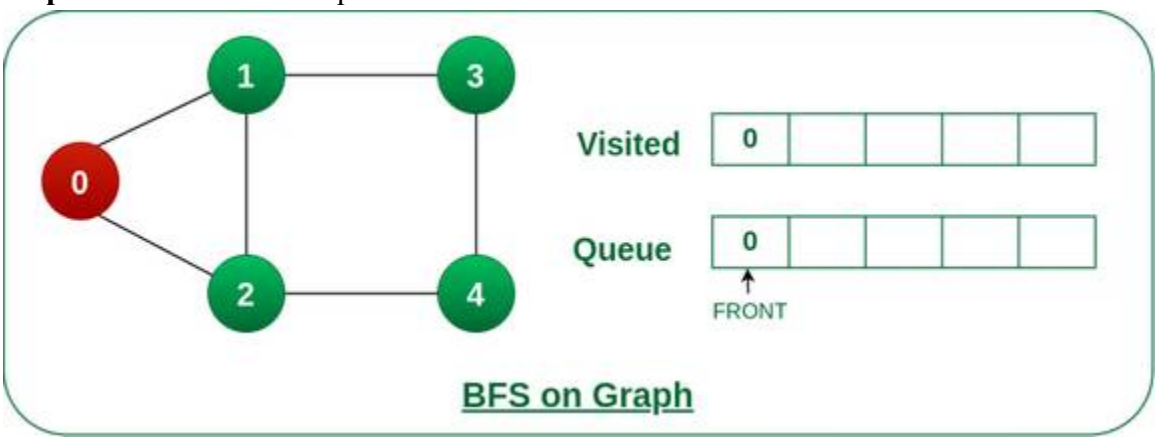
Example:

Step 1: Initially queue and visited arrays are empty.



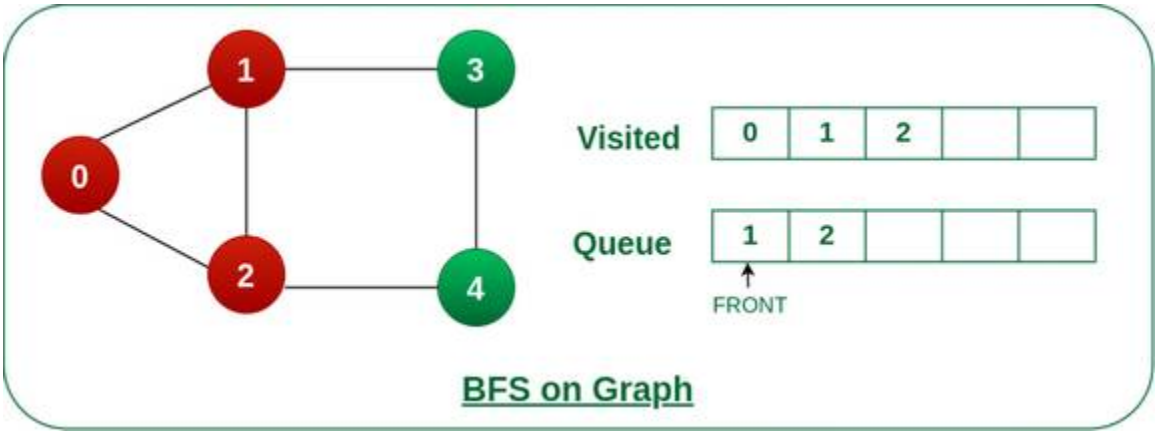
Queue and visited arrays are empty initially.

Step 2: Push node 0 into queue and mark it visited.



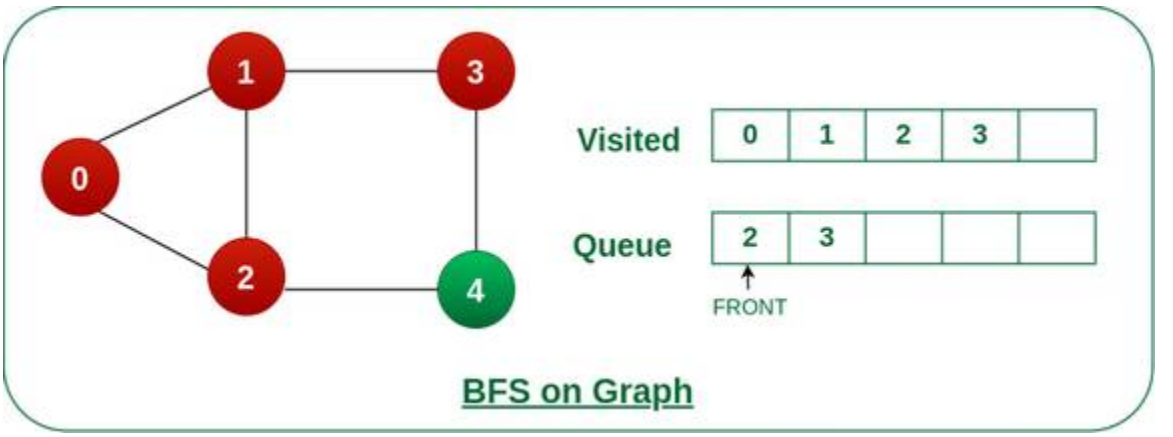
Push node 0 into queue and mark it visited.

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



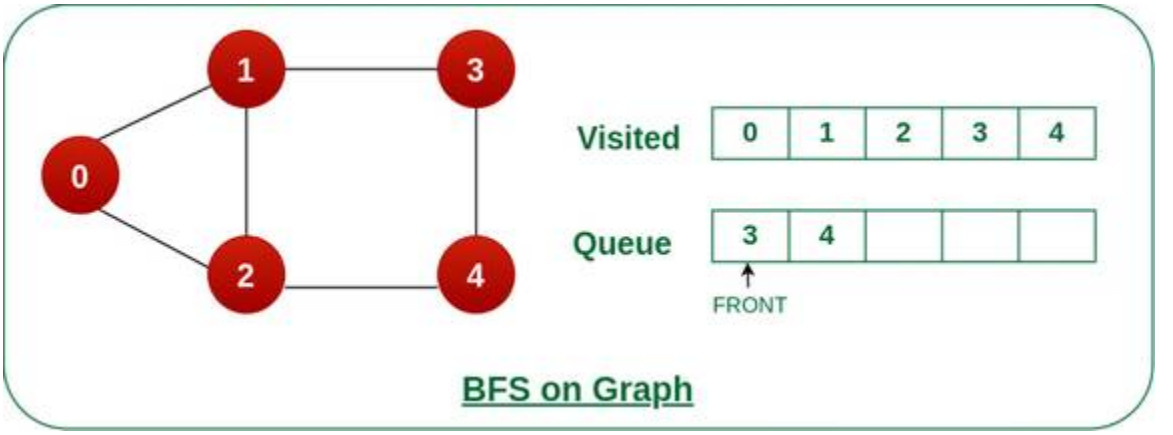
Remove node 0 from the front of queue and visited the unvisited neighbours and push into queue.

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



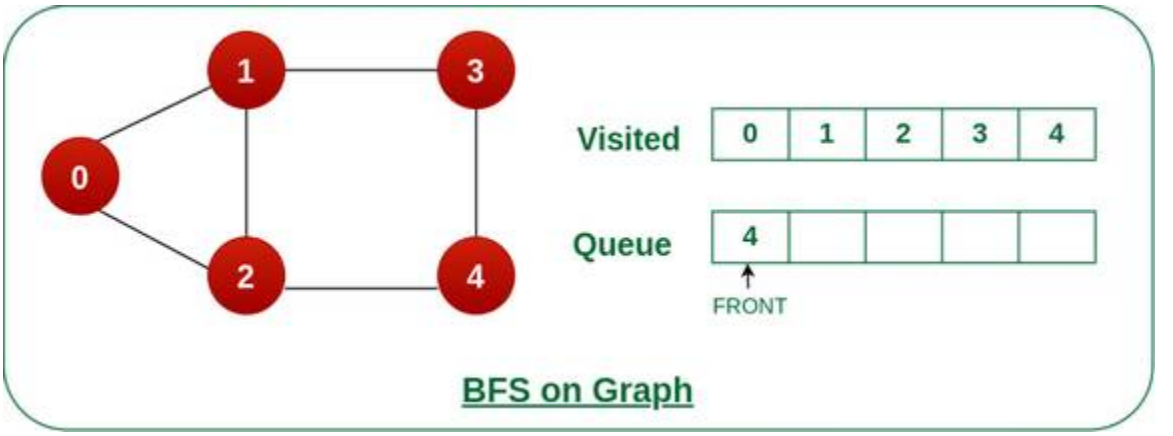
Remove node 1 from the front of queue and visited the unvisited neighbours and push

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



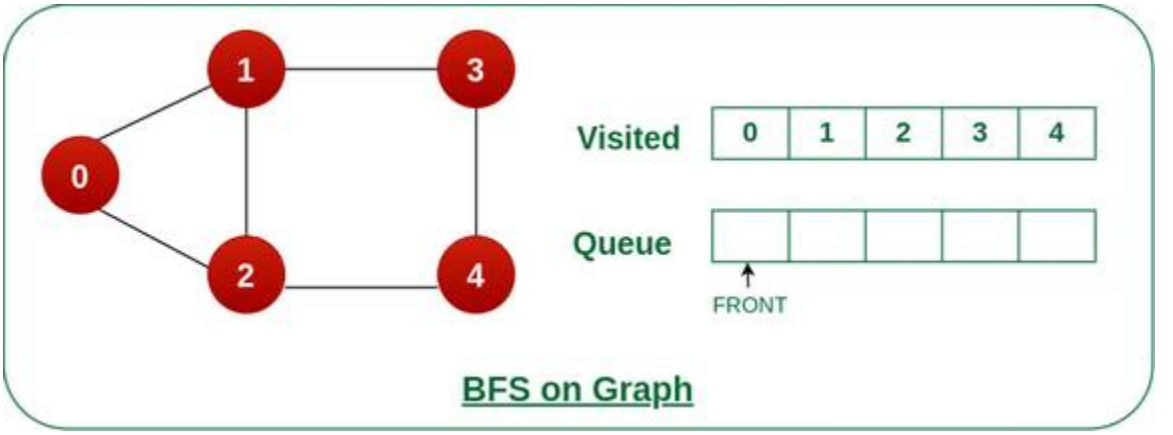
Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.

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.



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

Step 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.



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

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

Shortest Path Algorithm :

The shortest path algorithm is a set of computational methods used to find the shortest path between two nodes in a graph or a network. The "shortest path" refers to the path with the minimum sum of edge weights (or costs) between the source node (starting point) and the destination node (target).

The concept of the shortest path is often used in various real-world applications, such as finding the shortest route on a map, optimizing transportation systems, routing data packets in computer networks, and solving resource allocation problems. Well-known algorithms for finding the shortest path is Dijkstra's algorithm.

Dijkstra's Algorithm:

Dijkstra's algorithm is a greedy algorithm that works efficiently for finding the shortest path from a single source node to all other nodes in a graph with non-negative edge weights. It maintains a priority queue to keep track of the nodes to be visited in increasing order of their distance from the source node. The algorithm iteratively selects the node with the shortest distance and explores its neighbors, updating the distance to the neighbors if a shorter path is found. Dijkstra's algorithm guarantees that the shortest path to each node is found in a non-negative weighted graph.

Algorithm:

Step 1: Take a directed graph and form a distance matrix, as follows:

$$d_{ij} = \begin{cases} 0 & \text{distance between node } i \text{ and } i \\ \infty & \text{if there is no edge between } i \text{ and } j \\ w(i,j) & \text{Otherwise} \end{cases}$$

Step 2: Identify the source node s. This is the starting node.

Step 3: For all other vertices $j \neq s$; calculate the new distance. Initially, assign the source s a permanent label. This means that the node s has already been selected and its distance from the source is known.

Step 4: Choose the remaining vertices. Find the distance of only those vertices that are known to s from the source. This is what we call as a special path. This requirement may update the distance value. There are only two possibilities: The existing distance may itself be minimum. In that case, it is left alone. On the other hand, if the new distance is smaller than the old distance, then it is updated. Such a process of improving the result by path update is known as relaxation. The minimum vertex is then added to the set of vertices that are known to the source s.

Step 5: Continue Step 4 until all the vertices get a permanent label.

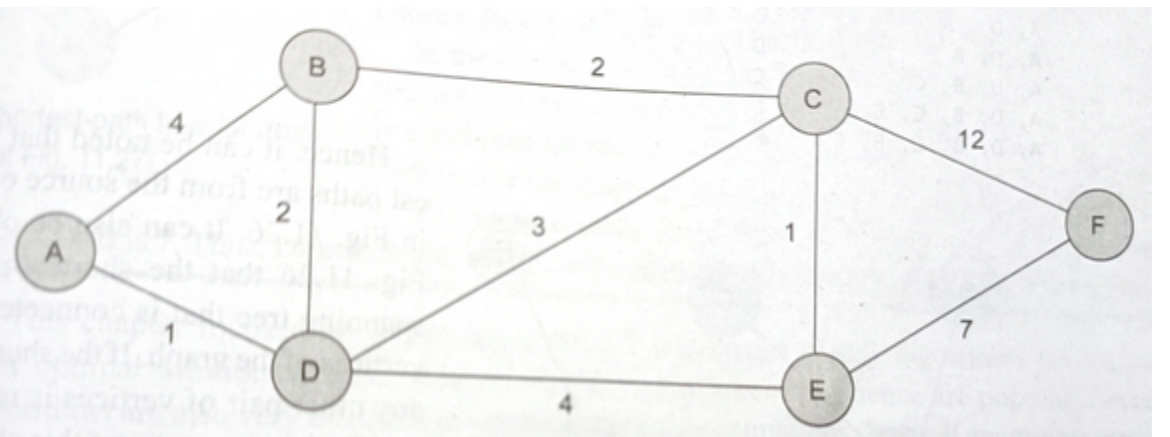
Step 6: Generate paths from the source node s.

The following is the formal algorithm for this method:

Algorithm SSSP(G) :

```
%% Input: weighted graph G = <V, E>
%% Output: Distance between source node s and a given node
Begin
      known = {s}
      d[s] = 0                                          %% Set the distance of s as
    %% Initialize the distance of other vertices as follows
      for each vertex v ∈ V such that v ≠ s do          %% Initialize
        if (s, v) ∈ E then
          d[v] = w[s, v]
        else
          d[v] = ∞
        End if
      End for
    while (known ≠ V) do
      pick a node v from V - s with smallest d[v]       %% Selection procedure
      known = known U{v}
      for all the nodes w ∈ V - s such that (u, v) ∈ E do
        d[w] = minimum{d[w], d[v, w]}//relax
      End for
    End while
    Return d
End
```

Example:



Answer:

P_v	d(v) v = A	v = B	v = C	v = D	v = E	v = F	Chosen vertex
Initialization {}	0	∞	∞	∞	∞	∞	A min.
{A}	–	4, A	∞	1,A	∞	∞	D
{A, D}	–	2, D	3, D	–	4, D	∞	B
{A, D, B}	–	–	3, D	–	4, D	∞	C
{A, D, B, C}	–	–	–	–	4, D	12, C	E
{A, B, D, C, E}	–	–	–	–	–	7, E	F
{A, B, D, C, E, F}	–	–	–	–	–	–	–

For this problem, the shortest paths from source A are as follows:
Start from source vertex A

A, D	D
A, D, B	B
A, D, B, C	C
A, D, B, C, E	E
A, D, B, C, E, F	F

Transitive Closure :

Given a directed graph G = (V, E), the transitive closure of G is a new graph G' = (V, E') where there is an edge (u, v) in E' if and only if there exists a directed path from vertex u to vertex v in G.

Transitive closure is a fundamental concept in graph theory and is used to determine the reachability between pairs of vertices in a directed graph. It helps identify all possible paths, including indirect paths, between vertices.

The informal algorithm for the Warshall algorithm is given as follows:

Step 1: Read weighted graph G = <V , E >.

Step 2: Initialize P [i ,j] with the adjacency matrix of G.

Step 3: Recursively compute for k = 1, 2, ..., n.

3a: $P_{ij}^{k} = P_{ij}^{(k-1)} \vee (P_{ik}^{(k-1)} \wedge P_{kj}^{(k-1)})$

Step 4: Return path matrix P.

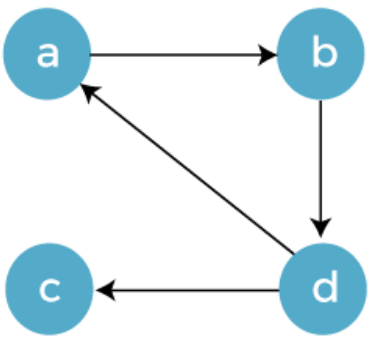
Step 5: End.

The formal algorithm for the Warshall algorithm can be written as follows:

Algorithm Warshall (G, A):

```
%% Input: Graph G and adjacency matrix A
%% Output: Path matrix P
Begin
     for i = 1 to n
         for j = 1 to n   %% Initialize
             P[i,j] = A[i,j]
         End for
     End for
     for k = 1 to n
         for i = 1 to n
             for j = 1 to n          %% Initialize
                 P[i,j,k] = P[i,j,k-1]∨(P[i,k,k-1]∧P[k,j,k-1])
                 %% P_ij^(k) = P_ij^(k-1) ∨ (P_ik^(k-1) ∧ P_kj^(k-1))
             End for
         End for
     End for
     return P
End
```

Example:



For this graph R(0) will be looked like this:

$R^{(0)} =$

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

Here R(0) shows the adjacency matrix. In R(0), we can see the existence of a path, which has no intermediate vertices. We will get R(1) with the help of a boxed row and column in R(0).

In R(1), we can see the existence of a path, which has intermediate vertices. The number of the vertices is not greater than 1, which means just a vertex. It contains a new path from d to b. We will get R(2) with the help of a boxed row and column in R(1).

$R^{(1)} =$

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	**1**	1	0

In R(2), we can see the existence of a path, which has intermediate vertices. The number of the vertices is not greater than 2, which means a and b. It contains two new paths. We will get R(3) with the help of a boxed row and column in R(2).

$R^{(2)} =$

	a	b	c	d
a	0	1	0	**1**
b	0	0	0	1
c	0	0	0	0
d	1	1	1	**1**

In R(3), we can see the existence of a path, which has intermediate vertices. The number of the vertices is not greater than 3, which means a, b and c. It does not contain any new paths. We will

get R(4) with the help of a boxed row and column in R(3).

$$R^{(3)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \left[\begin{array}{cccc} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{array}\right] \end{array}$$

In R(4), we can see the existence of a path, which has intermediate vertices. The number of the vertices is not greater than 4, which means a, b, c and d. It contains five new paths.

$$R^{(4)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \left[\begin{array}{cccc} \mathbf{1} & 1 & \mathbf{1} & 1 \\ \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{array}\right] \end{array}$$

Minimum Spanning Tree:

Minimum Spanning Tree (MST) is a crucial concept in graph theory and optimization. It involves finding the smallest tree that connects all vertices in a graph while minimizing the total edge weight.

Applications:

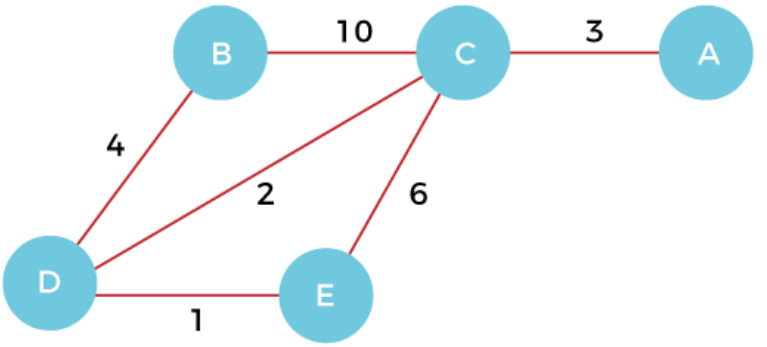
- Network design
- Transportation
- Communication, etc.

Prim's Algorithm:

Algorithm Overview:

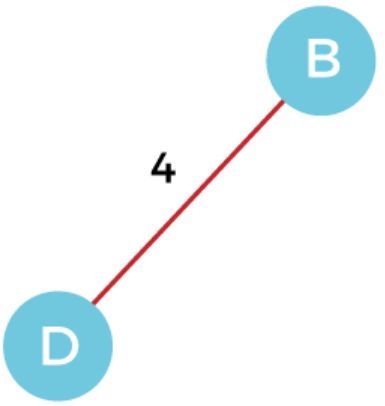
- Start with an arbitrary vertex and repeatedly add the edge with the lowest weight connecting a vertex in the MST to a vertex outside the MST.
- Maintain a set of vertices in the MST and a set of vertices outside the MST.
- Greedy approach: Always choose the edge with the minimum weight to expand the MST.

Example:

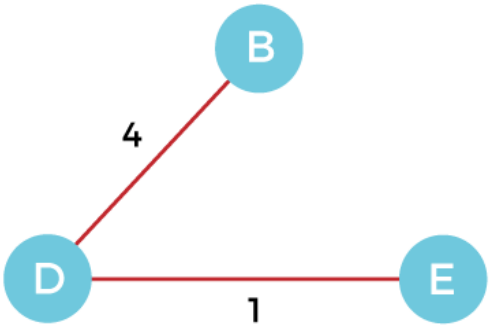


Step 1 - First, we have to choose a vertex from the above graph. Let's choose B.

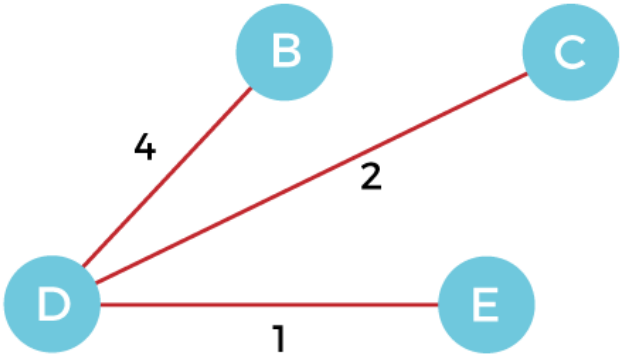
Step 2 - Now, we have to choose and add the shortest edge from vertex B. There are two edges from vertex B that are B to C with weight 10 and edge B to D with weight 4. Among the edges, the edge BD has the minimum weight. So, add it to the MST.



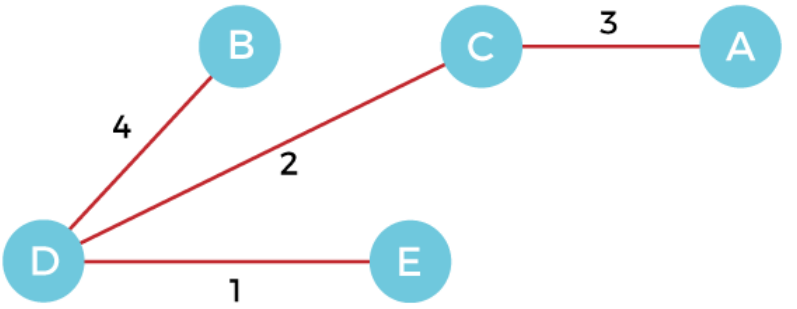
Step 3 - Now, again, choose the edge with the minimum weight among all the other edges. In this case, the edges DE and CD are such edges. Add them to MST and explore the adjacent of C, i.e., E and A. So, select the edge DE and add it to the MST.



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 -

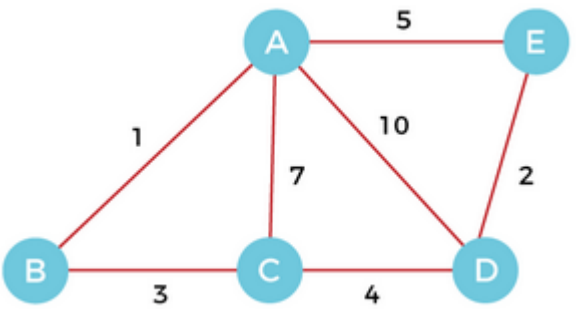
Cost of MST = 4 + 2 + 1 + 3 = 10

Kruskal's Algorithm:

Algorithm Overview:

- Sort all edges by weight in non-decreasing order.
- Start with an empty set of edges and repeatedly add the next edge in the sorted order if it does not create a cycle.
- Utilizes the disjoint-set (or union-find) data structure to keep track of connected components.
- Greedy approach: Always choose the smallest edge that doesn't form a cycle.

Example:



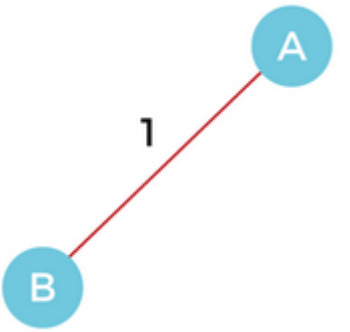
The weight of the edges of the above graph is given in the below table

Edge	AB	AC	AD	AE	BC	CD	DE
Weight	1	7	10	5	3	4	2

Now, sort the edges given above in the ascending order of their weights.

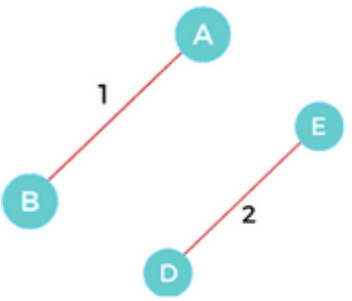
Edge	AB	DE	BC	CD	AE	AC	AD
Weight	1	2	3	4	5	7	10

Now, let's start constructing the minimum spanning tree.

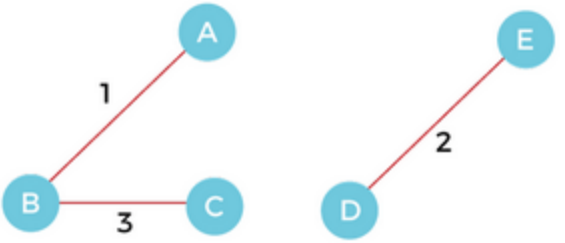


Step 1 - First, add the edge **AB** with weight **1** to the MST.

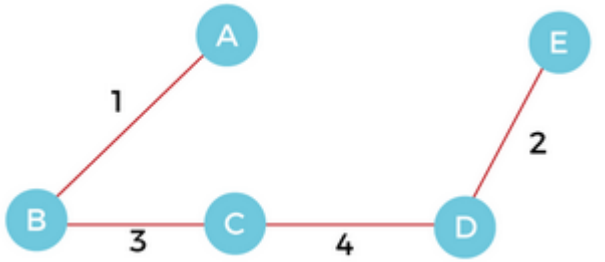
Step 2 - Add the edge **DE** with weight **2** to the MST as it is not creating the cycle.



Step 3 - Add the edge **BC** with weight **3** to the MST, as it is not creating any cycle or loop.



Step 4 - Now, pick the edge **CD** with weight **4** to the MST, as it is not forming the cycle

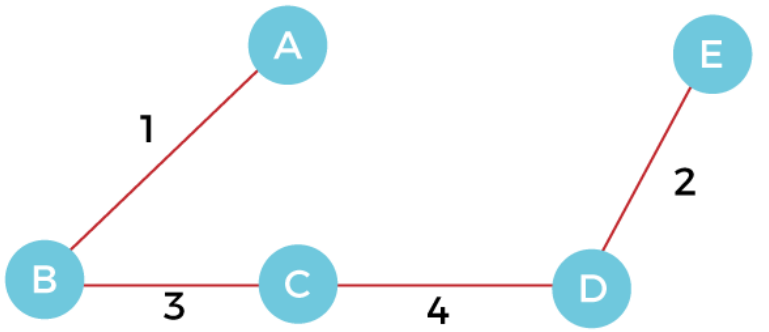


Step 5 - After that, pick the edge **AE** with weight **5.** Including this edge will create the cycle, so discard it.

Step 6 - Pick the edge **AC** with weight **7.** Including this edge will create the cycle, so discard it.

Step 7 - Pick the edge **AD** with weight **10.** Including this edge will also create the cycle, so discard it.

So, the final minimum spanning tree obtained from the given weighted graph by using Kruskal's algorithm is



The cost of the MST is = AB + DE + BC + CD = 1 + 2 + 3 + 4 = 10

Topological Sorting:

Given a directed graph G = (V, E), a topological ordering is a linear ordering of the vertices in V such that if there is a directed edge (u, v) in E, then vertex u comes before vertex v in the ordering.

Topological sorting is a crucial concept in graph theory and algorithms, especially in directed acyclic graphs (DAGs). It provides a linear ordering of the vertices in such a way that

for every directed edge (u, v), vertex u appears before vertex v in the ordering. Topological sorting has applications in scheduling, task sequencing, and dependency resolution.

Algorithm:

Step 1: Initialize a queue Q by traversing the graph and identify the next vertex u with no incoming edges. If there are many such vertices, then any node or vertex can be taken for further consideration. If there is none, then stop as topological sort cannot be performed.

Step 2: Pick the vertex u.

Step 3: Delete the vertex u that has no incoming edges along with all the edges.

Step 4: Repeat Steps 1-3 till all the vertices of the given graph are processed.

C

Example:

B

D

E

F

A

1.The adjacency matrix of the graph shown in the above figure is shown in the table.

Adjacency matrix of the graph shown in Figure

	A	**B**	**C**	**D**	**E**	**F**
A	0	1	0	0	0	0
B	0	0	1	1	0	0
C	0	0	0	1	0	0
D	0	0	0	0	1	0
E	0	0	0	0	0	1
F	0	0	0	0	0	0

Let us assume that m= 1. It can be observed that all the elements in the first column are zero. Therefore, A is assigned a number 1. Then delete the first column and first row, this yields a new adjacency matrix as shown in Table

Adjacency matrix after removing A node

	B	**C**	**D**	**E**	**F**
B	0	1	1	0	0
C	0	0	1	0	0
D	0	0	0	1	0
E	0	0	0	0	1
F	0	0	0	0	0

2. Delete B. Assign it the number 2 (i.e., m = m + 1). Delete the second row along with column. This yields a new adjacency matrix as shown in Table.

Adjacency matrix after removing B node

	C	**D**	**E**	**F**
C	0	1	0	0
D	0	0	1	0
E	0	0	0	1
F	0	0	0	0

3. Delete C. Assign it the number 3 (i.e., m = m + 1). Delete its column along with its row. This yields a new adjacency matrix as shown in Table.

Adjacency matrix after removing C node

	D	**E**	**F**
D	0	1	0
E	0	0	1
F	0	0	0

4. Delete D. Assign it the number 4 (i.e., m = m + 1). Delete its column along with its row. This yields a new adjacency matrix as shown in Table.

Adjacency matrix after removing D node

	E	**F**
E	0	1
F	0	0

5. Delete E. Assign it the number 5 (i.e., m=m+1). Deleting the column with its row finally yields the node F. The resulting linear ordering of the vertices is shown in Figure. The number of vertices assigned by the topological sort is given as a subscript.

F

E

D

C

B

A

Final linear ordering

UNIT-IV

BASICS OF COMPUTATIONAL COMPLEXITY

COMPUTATIONALCOMPLEXITY:

- Thealgorithmiccomplexitytheorydealswiththeanalysisofalgorithmsintermsofcomput ational resources such astimeorspace.
- Theobjectiveofcomplexitytheoryisto decide whetheragivenalgorithm is efficientornot.
- Computationalcomplexitydealswiththesolvabilityofagivenproblem.

CATEGORIZATIONOFALGORITHMS:

- The rearetwocategoriesofalgorithms,
    - PolynomialAlgorithm.
    - ExponentialAlgorithm

- POLYNOMIALALGORITHM:
    - Lettherebeapolynomialp(n)ofdegreen.

- Analgorithmiscalledapolynomialalgorithmiftheupperboundofthealgorithm is O(p(n)).
- Examplesofthispolynomialalgorithmaresearching,sortingalgorithms,etc….

- EXPONENTIALALGORITHMS:
  - Analgorithmiscalledexponentialif itscomplexityisnotpolynomial.
  - In other words, algorithms whose complexity is O(n!) or $O(2^n)$ are allexponentialalgorithms.
  - Thesealgorithmsareinfeasiblealgorithms,thatisthenproblemscannotbesolved withinareas onableamountoftime.
  - Example forthisalgorithmisTowerofHanoi.

DECISIONPROBLEMSANDTURINGMACHINE:

- The computational complexity theory deals with decision problemsratherthanoptimizationproblem.
- Inadecisionproblem(orarecognitionproblem)isa problem inwhichtheoutputofthis problem isrestrictedto 'yes'or'no'.
- Anoptimizationproblemhasobjectivefunctionsandconstraints.

- Anysolutionthatsatisfiestheconstraints iscalledafeasiblesolution.
- Anysolutionthatsatisfiestheconstraintsandmaximizesorminimizestheobjectivefunction iscalledanoptimal solution.
- Problems such as the TSP, Knapsack problem, and M-colouring problemareallexamples of optimizationproblem.
- Conversion of an optimization problem to a decision problem isimportant,asadecisionproblemsimplifiestheissuesofcomputabilitywithoutalteringtheessenceiftheproblem.
- Toprocessadecisionproblem,onerequiresacomputingmodel,calleda**TuringMachine.**
- ThiswasdesignedbyAllenTuringandwasproposedin1936.

ThecomponentsofaTuringMachine are:

- **Memory**:
  - A Turing Machine has infinite memory in the form of an infinitetape. Atapeis dividedinto asetofcells.
  - Everycellhasasymbol,called atapesymbol.
  - Tapesymbolsarederivedfromthegivenalphabet{0,1}fordecision problems.
- **Read/Writehead:**
  - Theread/write headisalsoknownasatapehead.

- Thisisapointerthatpoints tothebeginningofatape.

- **States:**
  - ATuringmachinehasmanystates,and,atanypointoftime,itcanbinanyoneofth estates.
  - Ofallthestatesthe following threeareimportant.
- **Startstate:**
  - The initialstateiscalledq_0.
  - Atthetime ofinitialization,themachineis inthe startstate.
- **q_{accept}state:**
  - Thisiscalledan‘acceptingstate’.
  - ATuringmachineisinthisstateforadecisionproblemwhoseansweris‘yes’.
- **q_{reject} state:**
  - Thisiscalledan‘rejectingstate’.
  - ATuringmachineis inthestatefora decisionproblemwhoseansweris‘no’.
- **Program**
  - ThisisanimportantcomponentofaTuringmachine.ATuringmachine reads an input character under the read/write head,themachine movestoanewstate.Thisis called**transition**.
  - Transitionsbasedonthecurrentstateandthe newcharactersare well defined for a Turing machine and are listed in a tableiscalleda **‘Transitiontable’**.
  - Asaresultofatransition,thefollowingactionsarepossible:
    - The read/write head moves by one square(or cell) ineithertherightortheleftdirection.
    - Theread/writeheadcanwriteacharacterwherethetapeheadrests.
    - The machine can move to a new state that can be aspecialstatesuchasanacceptingstateorarejectingstate.
  - ThesareallimportantcomponentsofaTuringmachine.

- **EncodingandLanguages:**
  - ATuringmachinetakesaninputdecisionproblemasanencodedstring of1s and0s.

COMPLEXITYCLASS:

ComplexityclassescanbedividedintoPclassandNP.

ClassP

All the problems(decision or optimization problems) for which polynomial timealgorithms exist are said to belong to the complexity class P. P stands forpolynomialtime. Examples:searching,sortingandselectionproblems.

ClassNP

>> These problems are said to be intractable,as they do not any efficientsolutions.
>> NP stands for non-determinstic polynomial and refers to problems that aresolvablein polynomialtime.

NPproblemsareoftwtypes-NP-hardandNP-complete.

NP-hardproblems

>>Itis notpossibleto solveallthe problems.

>> If all problems in NP are polynomial time reducible to it,even though it maynotbein NPitself.

NP-completeproblems

>> If a polynomial time algorithm exists for any of these problems,all problemsinNPwouldbepolynomialtimesolvable.

DETERMINSTICALGORITHM:

>> A deterministic algorithm is one whose behavior is completely determinedbyits inputsand thesequenceofits instructions.

>>Foraparticularinput,thecomputerwillgivealwaysthesameoutput.

>>Cansolvetheprobleminpolynomialtime.

>>Operationareuniquelydefined.

>>Likelinearsearchandbinarysearch.

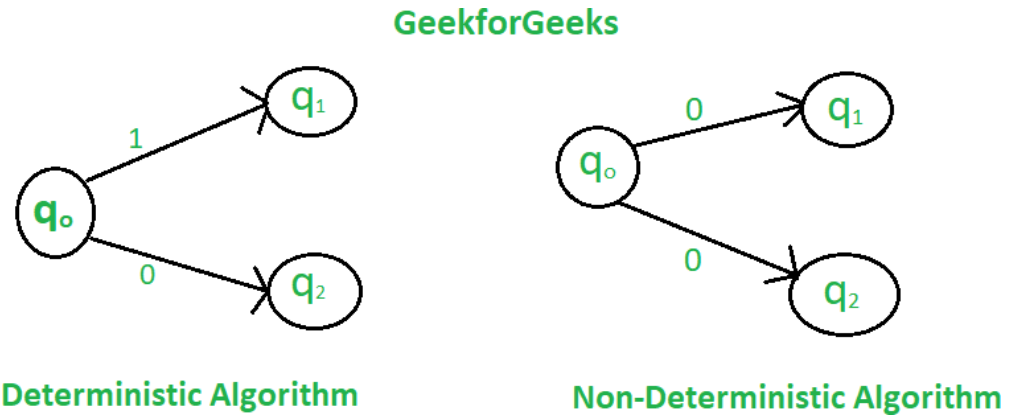
>>Deterministicalgorithmsusuallyhaveawell-definedworst-casetimecomplexity.

>>Deterministicalgorithmsareentirelypredictableandalwaysproducethesameoutputforthe sameinput.

>>Deterministicalgorithmsusuallyprovideprecisesolutionstoproblems.

>>Deterministicalgorithmsarecommonlyusedinapplicationswhereprecisioniscritical,s uchasincryptography,numericalanalysis,andcomputergraphics.

>>Examplesofdeterministicalgorithmsincludesortingalgorithmslikebubblesort,insertionsort, andselectionsort,aswellasmanynumericalalgorithms.



NON-DETERMINSTICALGORITHM:

>>Anon-deterministicalgorithmisoneinwhichtheoutcomecannotbepredictedwithcertainty,eveniftheinputs areknown.

>>Foraparticularinputthecomputerwillgivedifferentoutputsondifferentexecution.

>>Can’tsolvetheproblem polynomialtime.

>>Cannotdeterminethenextstepofexecutionduetomorethanonepaththealgorithmcantake.

>>Operationarenotuniquelydefined.

>>like0/1knapsackproblem.

>>Timecomplexityofnon-deterministicalgorithmsisoftendescribedintermsofexpectedrunningtime.

>>Non-deterministicalgorithmsmayproducedifferentoutputsforthesameinput duetorandomeventsor otherfactors.

>>Non-deterministicalgorithmsareoftenusedinapplicationswherefindinganexactsolutionisdifficultori mpractical,suchasinartificialintelligence,machinelearning,andoptimizationproblems.

>>Examplesofnon-deterministicalgorithmsincludeprobabilisticalgorithmslikeMonteCarlomethods,geneticalgorithms,andsimulatedannealing.

NON-DETERMINSTICSTAGES:

Therearetwostages:

- Guessingstage:Itgeneratesanarbitrarystringthatcanbethoughtasacandidatesolution.
- Verificationstage:
  >>Inthisstagewetakecandidatesolutionandinstanceoftheproblemastheinput.
  >>Itreturnsyesifthecandidaterepresenttheactualsolution(iftheguessingiscorrect).
  >>Iftheanswerisnoourguessingiswrong.

REDUCTIONTECHNIQUES:

>>Reductionbytransformandconquermethod.

>>Inthismethod,wesolveadifficultproblembytransformingitintoaknownproblemforwhichwehaveanoptimalsolution.

>>Basically,thegoalistofindareducingalgorithmwhosecomplexityisnotdominatedbytheresultingreducedalorithms.

UNIT-V
Advanced Topics

APPROXIMATION ALGORITHMS:

- **DEFINITION:**

An algorithm for problems (mostly NP-Hard or NP-Complete) is an approximate algorithm if it can give optimal solutions within a certain bound and also if it is possible to establish the solution guarantee analytically in worst case or an average in general.

The quality of an approximation algorithm is decided by

- Time Complexity analysis
- Comparing the generated feasible solutions with the optimal solutions ('Goodness Factor').

Goodness Factor or Goodness of an approximation algorithm relates the solution produced by the approximation algorithm and the actual optimal solution of the problem. It can be estimated through metrics. It is preferred to have an absolute performance measures, that is, the difference between the optimal and approximate solution is bounded by a constant, say k. But it is difficult to find such an approximation algorithm. Hence, one normally takes the relative performance.

In relative performance, a ratio of the optimal and approximation solution is obtained. This is called **Accuracy ratio (k)**.The range of Accuracy ratio is 0-1.

Let us assume that,

A - Approximation Algorithm for given problem Q,
I - Instance of the problem Q,
OPT (I) - Optimal solution of the problem Q,
A (I) - Feasible near optimal solution.

If Q is a minimization problem, then the ratio k=

If Q is a maximization problem, then the accuracy ratio is given as the reciprocal of k so that the accuracy ratio is the range 0-1.

An approximation algorithm that gives a near optimal solution in polynomial time, which is at most r times the optimal solution for any instance of the given problem is called **r-approximation algorithm**. Here r is called the **worst-case ratio** or **approximation ratio** and is given as follows:

The best value of r for which the inequality holds for all instances of the problem, that is, r=max I , is called a **Performance ratio**.

The value of performance ratio 'r' is 1 if the problem is of minimization problem and for maximization problem r1.

- **ADVANTAGES :**
  - Approximation algorithms optimize computer resources such as space and time. Hence, these algorithms are applicable not only to NP-Hard problems but can also replace algorithms that utilize more computer resources.
  - Approximation algorithms help categorize problems based on their difficulty levels.

- Approximation algorithms are valuable tools for developing and evaluating different types of heuristics for a given problem.

- **DISADVANTAGES :**
  - Approximation algorithms focus only on the worst-case measures and ignore heuristics that often work well in practical applications.
  - Approximation algorithms are limited to only a certain set of problems and not applicable to decision problems.

- **TYPES :**

An optimization problem is characterized by four parameters in approximation problems. The complexity classes of decision problems are P, NP-complete and NP-hard. These complexity classes can be extended for optimization problems also. These classes are called PO, NPO and NP-hard. NPO is a complexity class of optimization problems also.

There are three ways of solving PO, NPO and NP-hard categories of problems:

Exact Algorithms are exponential algorithms that provide exact solutions.
For the given problem. It cannot be practically implemented due to limited computer resources.

Approximate Algorithms are suitable for solving PO, NPO and NP-hard problems. These algorithms give approximate solutions and it is possible analytically to establish the solutions.

Heuristic Algorithms use trial and error methods to provide approximate solutions for computationally hard problems. The performances of the heuristic algorithms are often verified by computer experiments rather than by analytical methods.

- **CLASSIFICATION OF APPROXIMATION ALGORITHMS BASED ON APPROXIMATION RATIO :**

Based on approximation ratio r, the approximation algorithm can be classified as follows:

Approximate Algorithms An algorithm A is called an absolute or constant ratio approximate algorithm if approximate ratio r is a fixed constant. This implies that the difference between an absolute optimal solution and the solution generated by the approximation algorithm is always a constant. It also means that approximation algorithm is independent of the input instance of the problem.

Logarithmic Approximation If r is O (log (I)), where I is the instance of the problem, then the algorithm is called a logarithmic ratio or logarithmic approximation.

f (n)-approximation An algorithm A is called f (n)-approximation if and only if, for all instances of size, the approximation is O(). This assumes that the approximate solution A (I) is greater than Zero.

An algorithm A is called approxiamtion if and

Only if, for all instances of size, the following condition holds:

|A (I)-OPT (I) |≤for all instances I and for >0.

Approximation schemes An approximation algorithm A () is one that accepts two inputs-input instances and approximation ratio

(Where >0). It approximates the optimal solution within a bound of (1+) called a Scheme. There are two approximation schemes available

- Polynomial time approximation scheme (PTAS).
- Full time approximation scheme (FPTAS).

VERTEX COVER PROBLEM:

The input for a vertex cover problem is a graph G= (V, E). The aim of the vertex cover problem is to find a vertex cover such that every edge of G is incident on at least one vertex in vc.

A vertex is said to be a cover or node cover if the edges of the graph G are incident on it. A cover is thus a minimum set of vertices which ensures that all the edges of the given graph are incident on at least one vertex of the set vc. It is to choose a se S of vertices such that all the edges are covered. In other words, all edges are incident at least one vertex of the vertex cover. Finding a vertex cover for simple graphs like the preceding example is very easy. However, it is difficult in larger graphs that involve many vertices. Thus, vertex cover is an NP-hard problem.

The simplest greedy algorithm for finding a vertex cover is to pick an edge e, e {u, v} arbitrarily, and add one of its end points, say v, to a solution cover c. Then the edge e can be deleted along with all the edges that are incident to v. The procedure for a greedy vertex cover can informally be written as follows:

Step 1: Initialize the vertex cover vc as null, that is, vc = {}.

Step 2: While the edge list E is not empty do the following:

2a: Pick an edge e=<u, v> arbitrarily

2b:vc = vc {v}.

2c: Remove e along with all edges u.

2d: End while.

Step 3: Return vc and exit.

The procedure for a greedy vertex cover can formally be written as follows:

Algorithm greedy _ vertex-cover (G)

```
%% Input: G=<V, E>, E is the edge list and V the vertex list
%% Output: Cover for the given graph
Begin
Vc=   %% Initialize vertex cover as null
 E'=E
 while (E') do
        pick an edge e=(u, v)E' arbitrarily
        %% for weighted set cover this should be minimum
vc=vc {v}
        %% Update all edge list E'
        E'=E'-{all edges that are adjacent to u or v}
 End while
 Return vc
End
```

It can be observed that the algorithm keeps picking edges (u, v) and add one end of the edge, that is, v to the vertex cover. Then, the edge list is updated. This process is continued till the algorithm returns the vertex cover.

A better algorithm can be obtained as follows: Instead of outputting only one vertex u, both the end points of the edge can be added onto the vertex. The modified vertex cover approximation algorithm procedure can be written as follows:

Step 1: Initialize the vertex cover vc as null, that is, vc = {}.

Step 2: While the edge list E is not empty do the following:

2a: Pick an edge e= {u, v} arbitrarily.

2b:vc = vc {u, v}.

2c: Delete e and all edges that has u and v end points.

Step 3: Return vertex cover vc.

The formal algorithm for vertex cover is given as follows:

Algorithm mod_ greedy _ vertex-cover (G)

```
%% Input: G=<V, E>, E is the edge list and V the vertex list
%% Output: Vertex Cover
Begin
vc=   %% Initialize vertex cover as null
 E'=E
 while (E') do
        pick an edge e=(u, v)E' arbitrarily
        %% for weighted set cover this should be minimum
```

```
vc=vc {u, v}
        %% Update all edges
        E'=E'-{all edges that are adjacent to either u or v}
 End while
 Return vc
End
```

Complexity Analysis of vertex cover:

A vertex cover covers every edge in matching M and includes at least one end point. In matching, two edges share the same point. Therefore, it can be proved that OPT≥|M|. It has been proved that the accuracy ratio is 2, as the arbitrary picking of an edge amounts to maximal matching M. The minimal covering is |M|. However, the vertex cover algorithm returns a vertex cover with at least vertices of size 2*|M|. Therefore, one can conclude that the returned vertex cover is ≥ 2 x OPT. Therefore, it can be observed that the accuracy ratio of vertex cover is 2. Thus, the complexity analysis of vertex cover problem is 2 OPT.

RANDOMIZED ALGORITHMS:

- **DEFINITION:**

The input for a randomized algorithm are similar to those of deterministic algorithms, along with the sequence of random bits that can be used by the algorithm for making random choices. In other words, a randomized algorithm is one whose behavior depends on the inputs, similar to a deterministic algorithm, and the random choices are made as part of its logic. As a result, the algorithm gives different outputs even for the same input. In other words, the algorithm exhibits randomness; hence its run-time is often explained in terms of a random variable.

- **ADVANTAGES :**
    - Randomized algorithm are known for their simplicity. Any deterministic algorithm can easily be converted to a randomized algorithm. These algorithms are very simple to understand and implement.
    - Randomized algorithms are very efficient. They utilize little execution time and space compared to any deterministic algorithms.
    - Randomized algorithms exhibit superior asymptotic bounds compared to deterministic algorithms. In other words, the algorithm complexity of randomized algorithms is better than that of most of the deterministic algorithms.
- **DISADVANTAGES :**
    - Reliability is an important issue in many critical applications, as not all randomized algorithms give correct answers always. In addition, many randomized

algorithms may not terminate. Hence, reliability is an important concern that needs to be dealt with.

- The quality of randomized algorithms is dependent on the quality of the random number generator used as part of the algorithm.

A randomized algorithm does not use a single design principle. Instead one should view randomized algorithms as those designed using a set of principles. Some of the design principles are listed below.

Concept of witness this principle involves the question of checking whether a given input possesses a property X or not. It is established by finding a certain object called a witness or a certificate. The witness is identified to prove the fact that the input indeed has the desired property X. By conducting a fewer trials it can be found out whether the property was really present. The presence of a witness is a strong proof of the property X. Otherwise one should conclude that the input does not have such a property X based on the absence of witnesses.

Fingerprinting Fingerprint is a shorter message that is representative of a larger object. Fingerprinting is a technique wherein we get a comparison of two large objects A and B only by comparing their respective short fingerprints. If two fingerprints does not match, then the objects A and B are different. However, if the fingerprints match then there is a strong circumstantial evidence that both objects are the same.

Checking identities Let us assume that an algebraic expression is given, and the problem is to check whether the expression evaluates to zero or not. The principle of checking identities is to plug the random variables of a given algebraic equation and check whether the expression evaluates to zero. If it is not zero, then the given expression is not an identity. Otherwise, there is a strong circumstantial evidence that the expression is identically zero.

Random sampling and ordering Performance of an algorithm sometimes improves by randomizing the input distribution or order. It can be observed that for certain ordering of the inputs the performance of the algorithm can be higher or just acceptable. Here, randomization leads to randomized ordering, partitioning and sampling.

Foiling the adversary Randomized algorithm can be viewed as a game between a person and an adversary that is a person proposing an algorithm and an adversary who tries to foil the algorithm by designing suitable inputs so that the algorithm takes a longer time.

Creation of randomness is a very important component of randomized algorithms. The randomness is also created by generating random numbers using different methods.

- **TYPES :**

  There are two types of randomized algorithms

  - Las Vegas Algorithm
  - Monte Carlo Algorithm

A Las Vegas algorithm either will terminate when the correct answer of probability > ½ or will not give any output. It may terminate and give correct answer or may not terminate at all. But if the algorithm produces an answer, it will be always correct. Also one can improve the probability of the correct answer by having more number of trials.

Monte Carlo algorithms are applied to decision problems that always give either yes or no kind of answers.

- If the answer is 'yes', then the result is correct with probability larger than ½.
- If the answer is 'no', then the algorithm does not give any answer. However, circumstantially one may conclude that the correct answer is no.

RANDOMIZED QUICKSORT:

Based on the complexity analysis, the average run time is (n log n). When the elements are already sorted, the performance of the algorithm time is reduced to (. One way to avoid this is to introduce a pre-processing step to randomize the input elements so that the algorithm would always perform in time.

The informal algorithm for randomized quicksort is given as follows:

Step 1: Pick an element of the array randomly as a pivot element.

Step 2: Use the pivot element to position the list into sub-lists.
Step 3: Recursively sort the sub-lists.

Step 4: Combine all the sorted sub-lists.

The formal algorithm for randomized quicksort is given as follows:

Algorithm of randomized quicksort A [first, last]:

```
%% Input: Array A
%% Output: Pivot element
Begin
      if first<last then
        k=random(first, last)    %% Generate the random number in this range
        swap (a[first]),k)
```

```
pivot  Rsplit (A)        %% Select the pivot randomly
       randomized quicksort (A[first…pivot])
       randomized quicksort(A[pivot+1,last])
      end if
End
```

Complexity Analysis of Randomized Quicksort:

Let the input array A be {x1, x2, x3,…xn } and Xij be the indicator random variable indicating whether two elements xi and xj be compared or not.

In other words, the number of comparisons to be performed is the upper bound of the run time of the problem. However, in the randomized version the analysis has to involve indicator variable. Thus this is denoted as follows

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

where X indicates the total number of comparisons made by the algorithms. Comparison of x_i and x_j would happen only under the following two conditions:

1. A sub-problem of quick sort contains x_i and x_j.
2. Either x_i or x_j is chosen as a pivot element.

Now, one has to find the upper bound on $E(X)$. Here, X is a random variable that tracks the number of comparisons made. Here x_i is the smallest element in one subarray and x_j is the smallest element of another subarray. If so, X_{ij}=1, oterwise X_{ij}=0. Therefore, the analysis has to consider only the indicator random variables.

Rewriting the equation,

$$E(X) = E\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} x_{ij}\right] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} E[x_{ij}]$$

$$= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \Pr(x_i \text{ is compared } x_j)$$

As said earlier, the question of comparing x_i and x_j would arise only if either is chosen as a pivot element. Otherwise, they would not be compared at all.

$$\therefore \Pr[x_i \text{ is compared to } x_j]$$

$$= \Pr[x_i \text{ is chosen as a pivot}] + \Pr[x_j \text{ is chosen as a pivot}]$$

$$= \frac{1}{j-i+1} + \frac{1}{j-i+1}$$

It can be observed that the pivot is chosen from a set which has $j - i + 1$ elements. In addition, both the events are equally likely. This implies that

$$\Pr[x_i \text{ is compared to } x_j] = \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1}$$

In other words, the probability $x_{ij} = 1$ is $\frac{2}{j-i+1}$

As discussed earlier,

$$E(x) = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} E[x_{ij}] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

Substitution of $j - i = k$ in this equation, it can be rewritten as follows:

$$= \sum_{i=1}^{n-1}\sum_{k=1}^{n} \frac{2}{k+1} \le \sum_{i=1}^{n-1}\sum_{k=1}^{n} \frac{2}{k} = 2\sum_{i=1}^{n-1}\sum_{k=1}^{n} \frac{1}{k} = \sum_{i=1}^{n-1} O(n \log n)$$

$$= O(n \log n)$$

PSPACE [POLYNOMIAL SPACE]:

In Computational complexity theory, PSPACE is the set of all decision problems that can be solved by a Turing machine using polynomial amount of space.

If we denote by SPACE (f (n))

$$PSPACE = \bigcup_{k \in N} SPACE(n^k)$$

The class PSPACE is closed under operations union and complementation. We want algorithm which uses small amount of memory. So that large amount of data can be manipulated without storing all data to computer hard-disk at a time. An algorithm take sub linear space because i/p n bits takes linear space.

Two –Tape Turing machine is used:

- Read-only tape containing i/p.
- Work-tape that can be freely used.
  Space required by work tape contribute to space complexity, L=SPACE (log n) & NL=NSPACE (log n).

PSPACE-COMPLETENESS:

All the decision problem that can be solved in P i/p length & if every other problem solved in polynomial space can be converted to polynomial time. A language 'A' is PSPACE-complete if it satisfies the two condition:

- 'A' is in PSPACE (APSPACE)
- Language belongs to PSPACE can be polynomial time reducible to 'A'.

APPLICATION:

- Hex (board game).
- First order logic of equality.
- First order theory of well-ordered sets.
- Lambda Calculus etc...