UNIT IV
ITERATIVE IMPROVEMENT

The Simplex Method-The Maximum-Flow Problem – Maximm Matching in Bipartite Graphs- The Stable marriage Problem.

PART A

1**. What is feasible solution and feasible region?**

A solution that satisfies all the constraints of the problem is called the feasible solution; the set of all such feasible points (solution) is called the feasible region.

Linear programming problems with the empty feasible region are called infeasible.

Define optimal solution.

A feasible solution that satisfies the objective function (maximizes or minimizes) is called the optimal solution.

1.When the feasible region in linear programming becomes unbounded

If feasible region for the objective function may or may not attain a finite optimal value, then such problems are called unbounded.

2.Define or state extreme point theorem. Nov/Dec 2017

Any linear programming problem with a nonempty bounded feasible region has an optimal solution; moreover, an optimal solution can always be found at an extreme point of the problem's feasible region.

3.What are the requirements for the standard form of linear programming problem? OR when a linear program is said to be unbounded Nov/Dec 2019

The standard form has the following requirements:

- It must be a maximization problem.
- All the constraints (except the non negativity constraints) must be in the form of linear equations with nonnegative right-hand sides.
- All the variables must be required to be nonnegative.

4.Write the general linear programming problem in standard form.

The general linear programming problem in standard form with m constraints and n unknowns ($n \geq m$) is

maximize $c_1x_1 + \ldots + c_nx_n$
subject to $a_{i1}x_1 + \ldots + a_{in}x_n = b_i$,
where $b_i \geq 0$ for $i = 1, 2, \ldots, m$
$x_1 \geq 0, \ldots, x_n \geq 0$.

It can also be written in compact matrix notations:

maximize cx
subject to Ax = b
where $x \geq 0$,

$$c = [c1\ c2\ \ldots\ cn],\ x = \begin{bmatrix} x1 \\ x2 \\ \vdots \\ xn \end{bmatrix},\ A = \begin{bmatrix} a11 & a12 & \ldots & a1n \\ \vdots & \vdots & & \vdots \\ am1 & am2 & \ldots & amn \end{bmatrix},\ b = \begin{bmatrix} b1 \\ b2 \\ \vdots \\ bm \end{bmatrix}$$

5. What is basic and non basic solution?

If the system obtained has a unique solution—as any non degenerate system of linear equations with the number of equations equal to the number of unknowns does is called a basic solution; its coordinates set to zero before solving the system are called nonbasic, and its coordinates obtained by solving the system are called basic.

6. Define objective row.

The last row of a simplex tableau is called the objective row.

7. Define maximum flow problem.

The problem of maximizing the flow of a material through a transportation network is called the maximum flow problem.

8. Define flow network. Or What are the essential properties of a flow graph ?(April/May 2021)

A digraph satisfying the following properties is called a flow network or simply a network.

- It contains exactly one vertex with no entering edges is called source and assumed to be numbered 1.
- It contains exactly one vertex with no leaving edges is called the sink and assumed to be numbered n.
- The weight u_{ij} of each directed edge (i, j) is a positive integer, called the edge capacity. (defines upper bound)

9. What is flow conservation requirement?

The total amount of the material entering an intermediate vertex must be equal to the total amount of the material leaving the vertex is called the flow-conservation requirement.

10. Define Max-Flow Min-Cut Theorem.

The value of a maximum flow in a network is equal to the capacity of its minimum cut.

11. Define preflow.

A preflow is a flow that satisfies the capacity constraints but not the flow- conservation requirement.

12. What is maximum cardinality matching?

A matching in a graph is a subset of its edges with the property that no two edges share a vertex. A maximum matching also referred as maximum cardinality matching is a matching with the largest number of edges.

13. Define or what are bipartite graph . (Nov/Dec 2021)

In a bipartite graph, all the vertices can be partitioned into two disjoint sets V and U, not necessarily of the same size, so that every edge connects a vertex in one of these sets to a vertex in the other set.

14. Define stable marriage problem.

A marriage matching M is a set of n (m, w) pairs whose members are selected from disjoint n-element sets Y and X in a one-one fashion, i.e., each man m from Y is paired with exactly one woman w from X and vice versa. The stable marriage problem is to find a stable marriage matching for men's and women's given preferences.

17. What is iterative improvement method ? NOV-2018 What is meant by iterative improvement technique ? (April/May 2021) (NOV/DEC 2021)

The iterative method is a computational technique. It consists of following steps

1. Start with some feasible solution.
2. Repeat following steps until no improvement is found

   Change current feasible solution to a feasible solution with a better value of objective function.
3. Return the last feasible solution as an optimal solution.

18. Enlist various applications of iterative improvement method?

Various application of iterative improvement method are-

1. Simplex method
2. Ford Flukersons algorithm for maximum flow
3. Matching of graph vertices.
4. Stable marriage problem.

19. What is linear programming problem?

The standard for of linear programming is- P=ax + by + cz

A linear programming (LP) problem is a problem in which we have to find the maximum (or minimum) value of a linear objective function.

20. What is bipartite graph ? Nov/Dec 2017

The graph G = (V, E) in which the vertex set V is divided into two disjoint sets X and Y in such a way that every edge e € E has one end point in X and other end point in Y.

For example

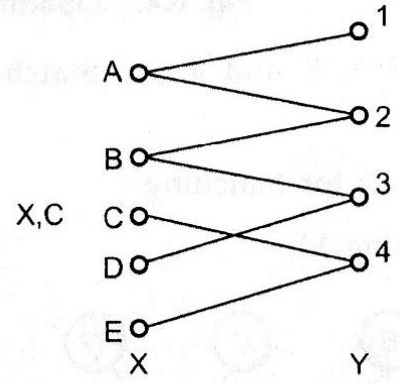


Fig. 6.4.1 Bi-partite graph

21. What is two colorable graph ?

The two colorable graph is a graph that can be colored with only two colors in such a way that no edge connect the same color . the bipartite graph is two colorable graph

22. What is maximum cardinality matching ? APR-2018

It is a matching with largest number of matching edges.

23. What is maximum matching problem ?

The maximum matching problem is a problem of finding maximum matching in a graph

24. What is entering variable ?

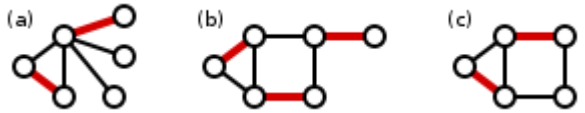
The entering variable is the smallest negative entry in the bottom row of the table.

25. What is departing variable

The departing variable is the smallest negative ratio of RHS/a_{ij} in the column determined by entering variable.

26.What do you mean by perfect matching in bipartite graph? (MAY 2015, Apr/May -2017)

A **perfect matching** is a matching which matches all vertices of the graph. That is, every vertex of the graph is incident to exactly one edge of the matching. Figure (b) above is an example of a perfect matching. Every perfect matching is maximum and hence maximal. In some literature, the term **complete matching** is used. In the above figure, only part (b) shows a perfect matching. A perfect matching is also a minimum-size edge cover. Thus, $\nu(G) \leq \rho(G)$, that is, the size of a maximum matching is no larger than the size of a minimum edge cover.



27.Define flow cut(AU MAY 2015)

Maximum flow

Definition. The **capacity** of an edge is a mapping $c : E \rightarrow \mathbf{R}^+$, denoted by c_{uv} or $c(u, v)$. It represents the maximum amount of flow that can pass through an edge.

Definition. A **flow** is a mapping $f : E \rightarrow \mathbf{R}^+$, denoted by f_{uv} or $f(u, v)$, subject to the following two constraints:

1. Capacity Constraint:

$$\forall (u, v) \in E : \quad f_{uv} \leq c_{uv}$$

2. Conservation of Flows:

$$\forall v \in V \setminus \{s, t\} : \quad \sum_{\{u:(u,v)\in E\}} f_{uv} = \sum_{\{u:(v,u)\in E\}} f_{vu}.$$

Definition. The **value of flow** is defined by

$$|f| = \sum_{v \in V} f_{sv},$$

where s is the source of N. It represents the amount of flow passing from the source to the sink.

Maximum Flow Problem. Maximize $|f|$, that is, to route as much flow as possible from s to t.

Minimum cut

Definition. An **s-t cut** $C = (S, T)$ is a partition of V such that $s \in S$ and $t \in T$. The **cut-set** of C is the set

$$\{(u, v) \in E \ : \ u \in S, v \in T\}.$$

Note that if the edges in the cut-set of C are removed, $|f| = 0$.

Definition. The **capacity** of an *s-t cut* is defined by

$$c(S, T) = \sum_{(u,v)\in S\times T} c_{uv} = \sum_{(i,j)\in E} c_{ij} d_{ij},$$

where $d_{ij} = 1$ if $i \in S$ and $j \in T$, 0 otherwise.

Minimum s-t Cut Problem. Minimize $c(S, T)$, that is, to determ ine S and T such that the capacity of the *S-T cut* is minimal.

28. How is a transportation network represented? Apr-18

Transportation networks generally refer to a set of links, nodes, and lines that **represent** the infrastructure or supply side of the **transportation**. The links have characteristics such as speed and capacity for roadways; frequency and travel time data are defined on **transit** links or lines for the **transit system**.

29. What is solution space? Give an example. Dec-18

Solution space is defined by the path from root node to any node in the tree.

Answer states are those solution states s for which the path from root node to s defines a tuple that is a member of the. Set of solutions. – These states satisfy implicit constraints. **State space tree** is the **tree** organization of the solution **space**.

30.Define the capacity constraint in the context of maximum flow problem Apr/May 2019

Maximum Flow Problem

Problem of maximizing the flow of a material through a transportation network (e.g., pipeline system, communications or transportation networks)

Formally represented by a connected weighted digraph with n vertices numbered from 1 to n with the following properties:

- Contains exactly one vertex with no entering edges, called the ***source*** (numbered 1)
- Contains exactly one vertex with no leaving edges, called the ***sink*** (numbered n)

Has positive integer weight *uij* on each directed edge (*i.j*), called the***edge capacity,*** indicating the upper bound on the amount of the material that can be sent from i to j through this edge.

A digraph satisfying these properties is called a **flow network** or simply a network

31. State the principle of duality?

The principle of duality in Boolean algebra states that if you have a true Boolean statement (equation) then the dual of this statement (equation) is true. The dual of a boolean statement is found by replacing the statement's symbols with their counterparts.

PART B

C_j	3	2	9	0	0	

1. Explain the simplex method with an example or linear programming Apr/May -2017
Or List the steps in Simplex Method and give the efficiency of the same. Nov/Dec 2017 Apr/May -2018 Nov/Dec 2018

The standard for linear programming is **P=ax + by + cz**

A **linear programming(LP) problem** is a problem in which we have to find the maximum (or minimum) value of a linear **objective function.**

The desired largest (or smallest)value of the objective function is called the **optimal value** and a collection of values of x,y,z,….that gives the optimal value constitutes an **optimal solution**.

- The variable x,y,z,…are called the **decision variables**.
- A basic solution for which all variables are non negative is called a basic feasible solution.

SIMPLEX METHOD:

1.Use simplex method to solve the linear programming problem

Max $Z=3x_1+2x_2$

subject to the constraints

$4x_1+3x_2 \leq 12$

$4x_1+x_2 \leq 8$

$4x_1-x_2 \leq 8$, $x_1,x_2 \geq 0$

Solution :

By introducing **non - negative slack variables** s_1,s_2 and s_3 the standard form of the LPP becomes ↑ (cost of basic variables)

Maximize $Z=3x_1+2x_2+\underline{0s_1+0s_2+0s_3}$

Basic variables

Subject to

$4x_1+3x_2+s_1=12$

$4x_1+x_2+s_2=8$

$4x_1-x_2+s_3=8$

$$\begin{pmatrix} x_1 & x_2 & s_1 & s_2 & s_3 \\ 4 & 3 & 1 & 0 & 0 \\ 4 & 1 & 0 & 1 & 0 \\ 4 & -1 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ s_1 \\ s_2 \end{pmatrix} = \begin{pmatrix} x_B \\ 12 \\ 8 \\ 8 \end{pmatrix}$$

Initial iterataion

c_B	B	x_1	x_2	s_1	s_2	s_3	x_B	Ratio= x_B/ x_1
0	s_1	4	3	1	0	0	12	12/4=3
0	s_2	4	1	0	1	0	8	8/4=2
0	s_3	**4**	1	0	0	1	8	8/4=2 (min)
Z_j		0	0	0	0	0		
Z_j-C_j		-3	-2	0	0	0		

Z_j-C_j=min{-3,-2}=-3

x_1→enter the basis

leave the basis=min{ x_B/ x_1 }

=min{+2,+2,+3}=2

s_3 → leave the basis

key element=4

- Here the basic variables are s1,s2 and s3 whereas the non basic variable are X and Y.
- If we put x=0 and y=0 then we get s1=12,s2=8 and s3=8.
- Hence the basic feasible solution can be written as
  - (x, y, s1, s2, s3) = (0, 0, 12, 8, 8)
- After setting up the initial simplex table the next task is to check the optimality and then if current solution is not optimal, improve the current solution.
- The improved solution is one that has a large z value than the current solution, to improve the current solution we obtain new basic variable into the solution.
- This variable is called the entering variable.
- This also implies that one of the current basic variable has to leave.
- This leaving variable is called departing variable.

Following are points to remember.

1. The **entering variable** is the smallest negative entry in the bottom row of the table.
2. The **departing variable** is the smallest negative ratio of RHS/a_{ij} in the column determined by entering variable.
3. The intersection of entering variable's column and departing variable's row is called **Pivot**. Make pivot value as 1 if it is not.

As the current solution (x, y, s1, s2, s3)=(0, 0,12, 8, 8) corresponds to z-value of 0 …..(1)

First iterataion

C_j		3	2	0	0	0		
c_B	B	x_1	x_2	s_1	s_2	s_3	x_B	Ratio= x_B/ x_1
0	s_1	0	4	1	0	-1	4	4/4=1
0	s_2	0	**2**	0	1	-1	0	0/2=0(min)
3	x_1	1	-1/4	0	0	1/4	2	-2*4=8
Z_j		3	-3/4	0	0	3/4	6	
Z_j-C_j		0	-11/4	0	0	3/4		

↑ (minus)

New row=oldrow-

keyrow*4

4-(1*4)=0
3-(-1/4*4)=4
1-0=1
0-0=0
0-(1/4*4)=-1
12(2*4)=4

New row=oldrow-keyrow*4

4-(1*4)=0
1-(-1/4*4)=2
0-0=0
0-(1/4*4)=-1
8-(2*4)=0

Z_j-C_j=min{-11/4}=-11/4
X_2→enter the basis
leave the basis=min{ x_B/ x_1 }
=min{+1,0}=0
S_2 → leave the basis
key element=2

second iterataion

C_j		3	2	0	0	0		
c_B	B	x_1	x_2	s_1	s_2	s_3	x_B	Ratio= x_B/ x_1
0	s_1	0	0	1	-2	**1**	4	4(min)
2	x_2	0	1	0	1/2	-1/2	0	-0*2=0
3	x_1	1	0	0	1/8	1/8	2	16
Z_j		3	2	0	11/8	-5/8	6	
Z_j-C_j		0	0	0	11/8	-5/8		

↑ (minus)

New row=oldrow-keyrow*4

0-0=0
4-4=0
1-0=1
0-(1/2*4)=-2
-1-(-1/2*4)=1
4-0=4

New row=old- -keyrow*-1/4

1-0=1
1/4-(1*-1/4)=0
0-0=0
0-(1/2*-1/4)=1/8
¼-(-1/2*-1/4)=1/8
2-0=2

Z_j-C_j=min{-5/8}=- 5/8
S_3→enter the basis
leave the basis=min{ x_B/ x_1 }
=min4,16}=4
S_1 → leave the basis
key element=1

Third iterataion

c_B	B	x_1	x_2	s_1	s_2	s_3	x_B	Ratio= x_B/ x_1
0	S_3	0	0	1	-2	1	4	
2	x_2	0	1	1/2	-1/2	0	2	
3	x_1	1	0	-1/8	3/8	0	3/2	
Z_j		3	2	5/8	1/8	0	17/2	
Z_j-C_j		0	0	5/8	1/8	0		

New row=oldrow-keyrow*-1/2
0-0=0
1-0=0
0-(1*-1/2=1/2
½-(-2*-1/2)=-1/2
-1/2-(1*-1/2)=0
0-(4*-1/2)=2
New row=old-key-keyrow*1/1
1-0=1
0-0=0
0-(1*1/8)=-1/8
1/8-(-2*1/8)=3/8
1/8-(1*1/8)=0
2-(4*1/8=3/2

Since all $Z_j-C_j \geq 0$
The optimal solution is,
$x_1=3/2$
$x_2=2$
$x_3=0$
Max Z =17/2

1.A.Use simplex method to Min $Z= x_2-3x_3+2x_5$ subject to the constraints

$3x_2-x_3+2x_5 \leq 7$
$-2x_2+4x_3 \leq 12$
$-4x_2+3x_3+8x_5 \leq 10$

And $x_2,x_3,x_5 \geq 0$ or

solve the following set of equations using simplex algorithm: Apr/May -2019
maximize:18 x1 +12.5 x2
subject to :x1 + x2 <= 20
x1<=12
x2<=16
x1,x2>=0

Solution :

By introducing **non - negative slack variables** s_1,s_2 and s_3 the standard form of the LPP becomes

↑ (cost of basic variables)

Minimization $Z = 3x_2 - x_3 + 2x_5$

Maximize $Z = -3x_2 + x_3 - 2x_5$

Subject to

$3x_2 - x_3 + 2x_5 + s_1 = 7$

C_j	-1	3	2	0	0	0	

$-2x_2 + 4x_3 + s_2 = 12$

$-4x_2 + 3x_3 + 8x_5 + s_3 = 10$

$$\begin{pmatrix} x_2 & x_3 & x_5 & s_1 & s_2 & s_3 \\ 3 & -1 & 2 & 1 & 0 & 0 \\ -2 & 4 & 0 & 0 & 1 & 0 \\ -4 & 3 & 8 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} x_B \\ 7 \\ 12 \\ 10 \end{pmatrix}$$

Initial iterataion

↑ (minus)

C_j		-1	3	2	0	0	0		
c_B	B	x_2	X_3	X_5	s_1	s_2	s_3	x_B	Ratio= x_B / x_3
0	s_1	3	-1	2	1	0	0	7	-7/1=7(min)
0	s_2	-2	**4**	0	0	1	0	12	12/4=3
0	s_3	-4	3	8	0	0	1	10	10/3=3.3
Z_j			0	0	0	0	0	0	
$Z_j - C_j$		+1	-3	+2	0	0	0		

$Z_j - C_j = \min\{-3\} = -3$

$X_3 \rightarrow$ enter the basis

leave the basis $= \min\{ x_B / x_1 \}$

$= \min 3, 3.3\} = 3$

$S_2 \rightarrow$ leave the basis

key element=4

first iteration

c_B	B	x_2	X_3	X_5	s_1	s_2	s_3	x_B	Ratio= x_B/ x_3
0	s_1	**5/2**	0	2	1	1/4	0	10	0*2/5=4(min)
3	X_3	-1/2	1	0	0	1/4	0	3	-3*2=-6
0	S_3	-5/2	0	8	0	-3/4	1	1	-2/5
Z_j		-3/2	3	0	0	3/4	0	9	
Z_j-C_j		-1/2	0	2	0	3/4	0		

↑ (minus)

New row=oldrow-keyrow*-1

3-(-1/2*-1)=5/2

-1-(1*-1)=0

2-0=2

1-0=1

0-(1/4*-1)=0

7-(3*-1)=10

New row=oldrow-keyrow*3

-4-(-1/2*3)=-5/2

3-3=0

8-0=8

0-0=0

0-(1/4*3)=-3/4

1-0=1

10-(3*3)=1

Z_j-C_j=min{-1/2}=-1/2

X_2→enter the basis

leave the basis=min{ x_B/ x_2 }

=min{4}=4

S_1 → leave the basis

key element=**second iteration**

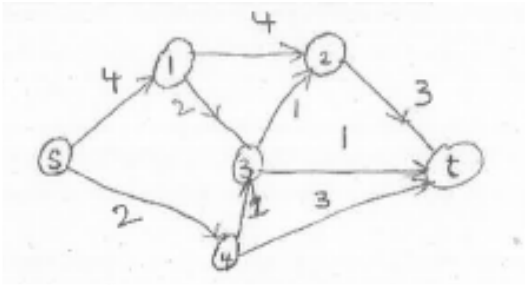
C_j		-1	3	2	0	0	0		
c_B	B	x_2	X_3	X_5	s_1	s_2	s_3	x_B	Ratio= x_B/ x_3

-1	x_2	1	0	4/5	2/5	1/10	0	4	
3	X_3	0	1	2/5	1/5	3/10	0	5	
0	s_3	0	0	10	1	-1/2	1	11	
Z_j		-1	3	2/5	1/5	8/10	0	11	
Z_j-C_j		0	0	12/5	1/5	8/10	0		

Since all $Z_j-C_j \geq 0$
answer is

$$\min (Z)=-11$$

2. Explain in detail about maximum flow problem. Or Determine the max-flow in the following network.



Apr/May 2019

Maximum Flow Problem

Problem of maximizing the flow of a material through a transportation network (e.g., pipeline system, communications or transportation networks)

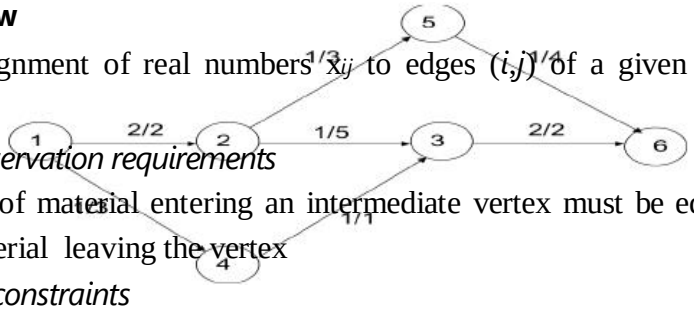
Formally represented by a connected weighted digraph with *n* vertices numbered from 1 to *n* with the following properties:

- Contains exactly one vertex with no entering edges, called the ***source*** (numbered 1)
- Contains exactly one vertex with no leaving edges, called the ***sink*** (numbered *n*)
- Has positive integer weight u_{ij} on each directed edge (*i.j*), called the***edge capacity,*** indicating the upper bound on the amount of the material that can be sent from *i* to *j* through this edge.
- A digraph satisfying these properties is called a **flow network** or simply a network.

Example of Flow Network

Node (1) = source
Node(6) = sink

Definition of a Flow



A *flow* is an assignment of real numbers x_{ij} to edges (i,j) of a given network that satisfy The following:

- *flow-conservation requirements*

The total amount of material entering an intermediate vertex must be equal to the total amount of the material leaving the vertex

- *capacity constraints*

$$0 \le x_{ij} \le u_{ij} \quad \text{for every edge } (i,j) \in E$$

Flow value and Maximum Flow Problem

Since no material can be lost or added to by going through intermediate vertices of the network, The total amount of the material leaving the source must end up at the sink:

$$\sum_{j:\,(1,j)\,\epsilon\,E} x_{1j} = \sum_{j:\,(j,n)\,\epsilon\,E} x_{jn}$$

The *value* of the flow is defined as the total outflow from the source (= the total inflow into Thesink). The *maximum flow problem* is to find a flow of the largest value (maximum flow) for a given network.

Maximum-Flow Problem as LP problem

Maximize $v = \sum_{j:\,(1,j) \in E} x_{1j}$

subject to

$$\sum_{j:\,(j,i) \in E} x_{ji} - \sum_{j:\,(i,j) \in E} x_{ij} = 0 \quad \text{for } i = 2, 3, \ldots, n-1$$

$$0 \le x_{ij} \le u_{ij} \quad \text{for every edge } (i,j) \in E$$

Augmenting Path (Ford-Fulkerson) Method

- Start with the zero flow ($x_{ij} = 0$ for every edge).
- On each iteration, try to find a *flow-augmenting path* from source to sink, which a path along which some additional flow can be sent.
- If a flow- augmenting path is found, adjust the flow along the edges of this path to get a flow of increased value and try again.
- If no flow-augmenting path is found, the current flow is maximum.

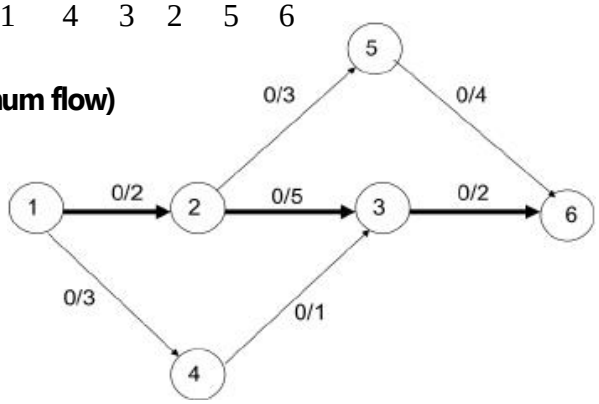
Example 1

Augmenting path: 1 2 3 6

x_{ij}/u_{ij}

Augmenting path: 1 4 3 2 5 6

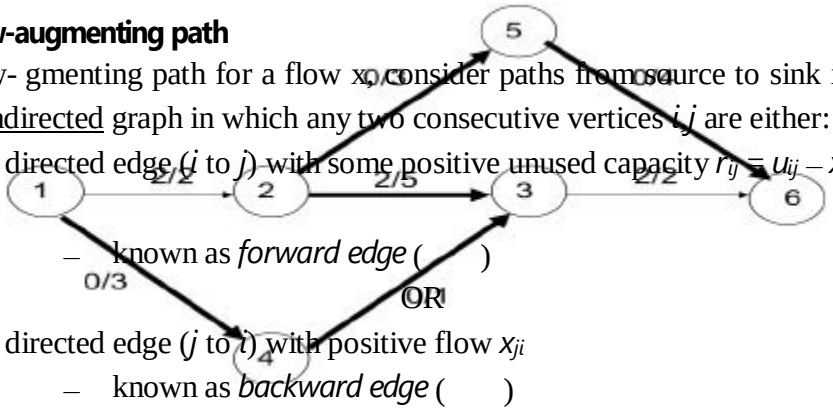
Example 1 (maximum flow)



Finding a flow-augmenting path

To find a flow- gmenting path for a flow x, consider paths from source to sink in the Underlying undirected graph in which any two consecutive vertices *i*, *j* are either:

connected by a directed edge (*i* to *j*) with some positive unused capacity $r_{ij} = u_{ij} - x_{ij}$



– known as *forward edge* ( )

connected by a directed edge (*j* to *i*) with positive flow x_{ji}

– known as *backward edge* ( )

If a flowaugmenting path is found, the current flow can be increased by *r* units by increasing x_{ij} by*r* on each forward edge and decreasing x_{ji} by *r* on each backward edge, where

r = min {r_{ij} on all forward edges, x_{ji} on all backward edges}

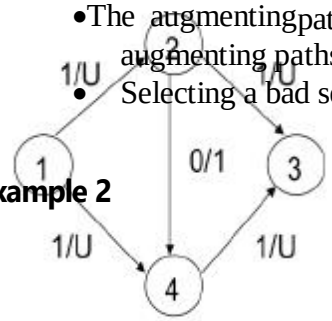
- Assuming the edge capacities are integers, *r* is a positive integer
- On each iteration, the flow value increases by at least 1
- Maximum value is bounded by the sum of the capacities of the edges leaving the source; hence the augmenting-path method has to stop after a finite number of iterations

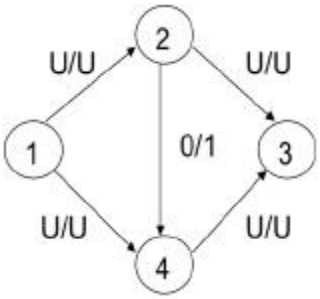
- The final flow is always maximum, its value doesn't depend on a sequence of augmenting paths used

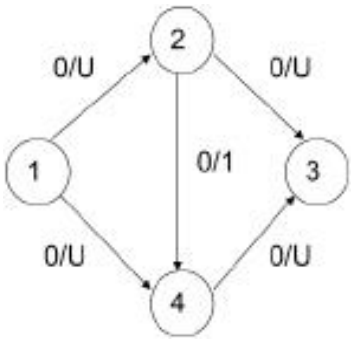
Performance degeneration of the method

- The augmenting path method doesn't prescribe a specific way for generating flow-augmenting paths
- Selecting a bad sequence of augmenting paths could impact the method's efficiency

Example 2

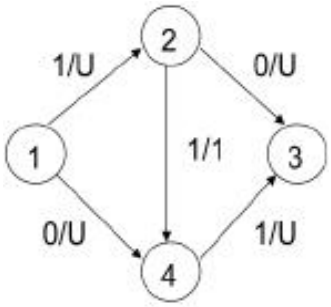




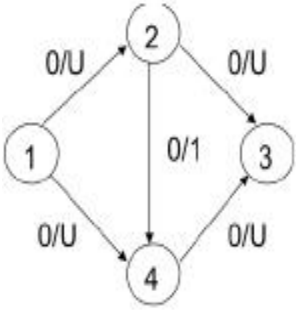


1→2→4→3

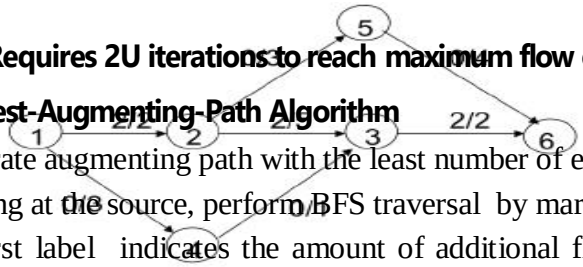
1→4←2→3 V=1



V=2



V=2U



Requires 2U iterations to reach maximum flow of value 2U

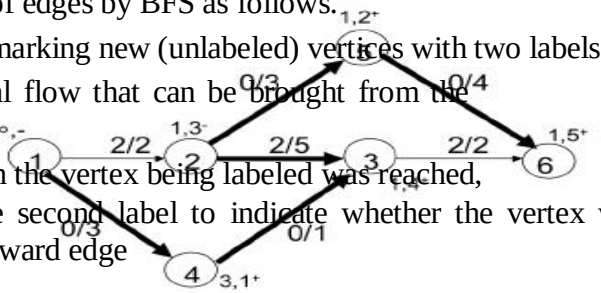
Shortest-Augmenting-Path Algorithm

Generate augmenting path with the least number of edges by BFS as follows.

Starting at the source, perform BFS traversal by marking new (unlabeled) vertices with two labels:

first label indicates the amount of additional flow that can be brought from the source to the vertex being labeled

second label – indicates the vertex from which the vertex being labeled was reached, with "+" or "–added to the second label to indicate whether the vertex was reached via a forward or backward edge



Vertex labeling

- The source is always labeled with ∞,-
- All other vertices are labeled as follows:

If unlabeled vertex j is connected to the front vertex i of the traversal queue by adirected edge from i to j with positive unused capacity

$r_{ij} = u_{ij} - x_{ij}$ (forward edge),vertex j is labeled with l_j, i^+, where $l_j = \min\{l_i, r_{ij}\}$

If unlabeled vertex j is connected to the front vertex i of the traversal queue by adirected edge from j to i with positive flow x_{ji} (backward edge), vertex j is labeled

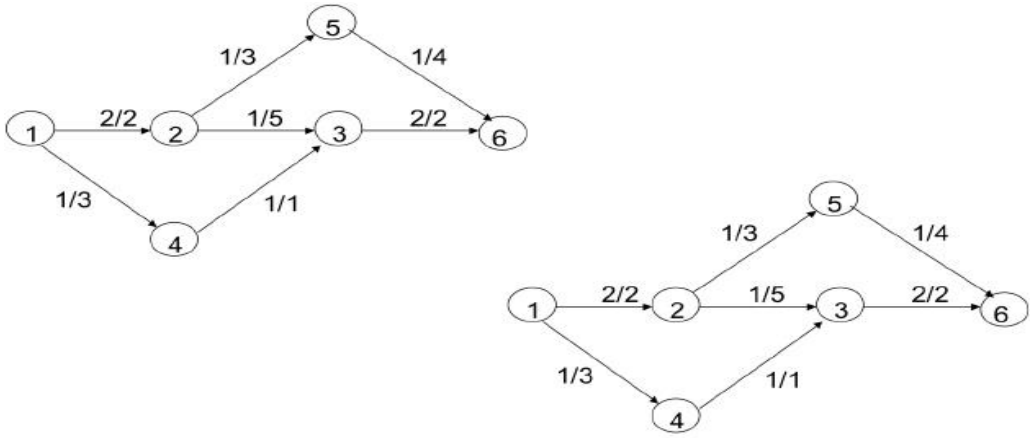
l_j, i^-, where $l_j = \min\{l_i, x_{ji}\}$

- If the sink ends up being labeled, the current flow can be augmented by the amount indicated by the sink's first label.
- The augmentation of the current flow is performed along the augmenting path traced by following the vertex second labels from sink to source; the current flow quantities areincreased on the forward edges and decreased on the backward edges of this path.
- If the sink remains unlabeled after the traversal queue becomes empty, the algorithm returns the current flow as maximum and stops.

Example: Shortest-Augmenting-Path Algorithm

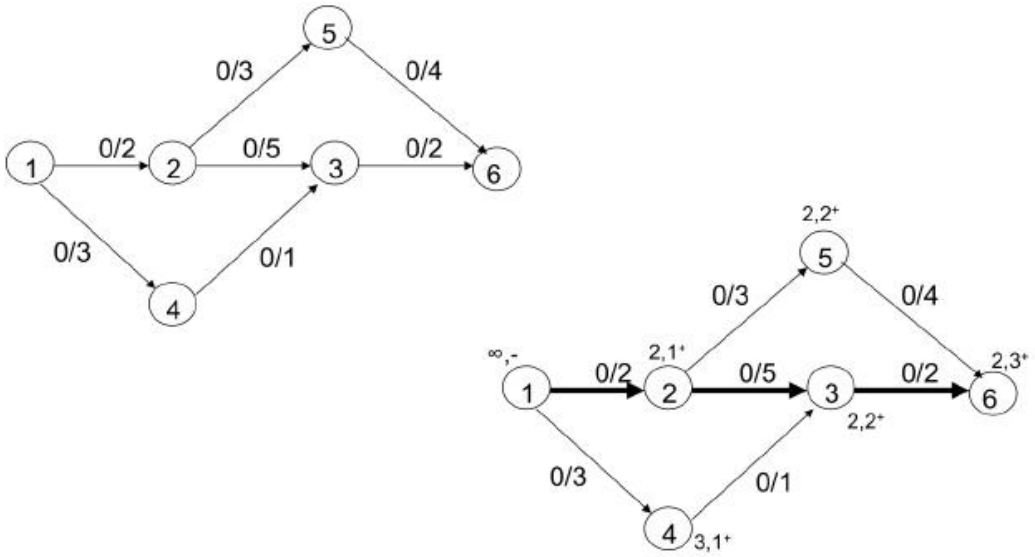
Queue: 1 2 4 3 5 6

Augment the flow by 2 (the sink's first label) along the path 1 2 3 6



ueue: 1 4 3 2 5 6

Augment the flow by 1 (the sink's first label) along the path 1 4 3 2 5 6



Queue: 1 4

No augmenting path (the sink is unlabeled) the current flow is maximum

Definition of a Cut

Let X be a set of vertices in a network that includes its source but does not include its sink, and letX, the complement of X, be the rest of the vertices including the sink. The *cut* induced by this partition of the vertices is the set of all the edges with a tail in X and a head in X.

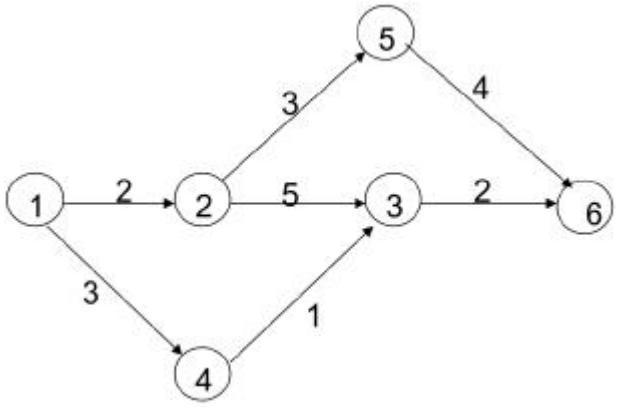
Capacity of a cut is defined as the sum of capacities of the edges that compose the cut.

→e'll denote a cut and its capacity by C(X,X) and c(X,X)

Note that if all the edges of a cut were deleted from the network, there would be no directed path from source to sink

Minimum cut is a cut of the smallest capacity in a given network

Examples of network cuts



If X = {1} and X = {2,3,4,5,6}, C(X,X) = {(1,2), (1,4)}, c = 5

If X ={1,2,3,4,5} and X = {6}, C(X,X) = {(3,6), (5,6)}, c = 6

If X = {1,2,4} and X ={3,5,6}, C(X,X) = {(2,3), (2,5), (4,3)}, c = 9

Max-Flow Min-Cut Theorem

1. The value of maximum flow in a network is equal to the capacity of its minimum cut
2. The shortest augmenting path algorithm yields both a maximum flow and a minimum cut:

Maximum flow is the final flow produced by the algorithm

Minimum cut is formed by all the edges from the labeled vertices to unlabeledvertices on t he last iteration of the algorithm.

All the edges from the labeled to unlabeled vertices are full, i.e., their flow amountsare equal to the edge capacities, while all the edges from the unlabeled to labeledvertices, if any, have zero flow amounts on them.

ALGORITHM *ShortestAugmentingPath(G)*

//Implements the shortest-augmenting-path algorithm

//Input: A network with single source 1, single sink *n*, and positive integer capacities *uij* on

// its edges *(i, j)*

//Output: A maximum flow *x*

assign *xij*= 0 to every edge *(i, j)* in the network

label the source with ∞, – and add the source to the empty queue *Q*

while not *Empty(Q)* **do**
    $i \leftarrow$ *Front(Q)*; *Dequeue(Q)*
    for every edge from i to j **do** //forward edges
        if j is unlabeled
            $r_{ij} \leftarrow u_{ij} - x_{ij}$
            if $r_{ij} > 0$
                $l_j \leftarrow \min\{l_i, r_{ij}\}$; label j with l_j, $i+$
                Enqueue(Q, j)
    for every edge from j to i **do** //backward edges
        if j is unlabeled
            if $x_{ji} > 0$
                $l_j \leftarrow \min\{l_i, x_{ji}\}$; label j with l_j, $i-$
                Enqueue(Q, j)
    if the sink has been labeled
        //augment along the augmenting path found
        $j \leftarrow n$ //start at the sink and move backwards using second labels
        while $j \neq 1$ //the source hasn't been reached
            if the second label of vertex j is $i+$
                $x_{ij} \leftarrow x_{ij} + l_n$
            else //the second label of vertex j is $i-$
                $x_{ij} \leftarrow x_{ij} - l_n$
            $j \leftarrow i$; $i \leftarrow$ the vertex indicated by i's second label
        erase all vertex labels except the ones of the source
        reinitialize Q with the source
return x //the current flow is maximum

Time Efficiency

The number of augmenting paths needed by the shortest-augmenting-path algorithm never exceeds $nm/2$, where n and m are the number of vertices and edges, respectively.

Since the time required to find shortest augmenting path by breadth-first search is in $O(n+m)=O(m)$ for networks represented by their adjacency lists, the time efficiency of the shortest-augmenting-path algorithm is in $O(nm^2)$ for this representation.

More efficient algorithms have been found that can run in close to $O(nm)$ time, but these algorithms don't fall into the iterative-improvement paradigm.

Analysis : The algorithm for Ford-Fulkerson has a **while** loop which executes for O (E). Hence running time of Ford-Fulkerson algorithm is O(EF*) where F* is the maximum flow found b algorithm.

4. Explain the Maximum Matching in Bipartite Graph algorithm with supporting example (AU MAY 2015) or what is a bipartite graph?is the subset of a bipartite graph bipartite?outline the with an example Nov/Dec 2019,Nov/Dec 2021

Explain the maximum-bipartite-matching problem with an illustration. (April/ May 2021)

Bipartite Graph:

The graph G = (V, E) in which the vertex set V is divided into two disjoint sets X and Y in such a way that every edge e € E has one end point in X and other end point in Y.

For example

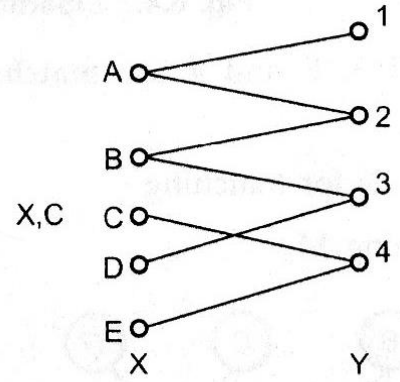


Fig. 6.4.1 Bi-partite graph

Matching:

A matching M is a subset of edges such that each node in V appears in at most one edge in M. In other words matching in a graph is a subset of edges that no two edges share a vertex.

Two-colorable Graph:

A graph can be colored with only two colors (i.e. two colorable graph) such that no edge connects the same color. The bi-partite graph is 2-colorable.

Free vertex:

μ ∈ V is a fee vertex, if no edge from matching M os incident to v (that means if v is not matched).

Alternating path:

The alternating path P is a path in graph G, such that for every pair of subsequent edges one of them is matching pair M and other is not.

Augmenting path:

The augmenting path P is a path in graph G, such that it is an alternating path with special property that its start and end vertices are free or unmatched.

Maximum matching or Maximum cardinality matching:

It is a matching with largest number of **matching edges.**

Maximum matching problem is a problem of finding maximum matching in a graph.

Let us apply iterative-improvement technique to maximum matching problem.

Let M be a matching in bipartite graph G.

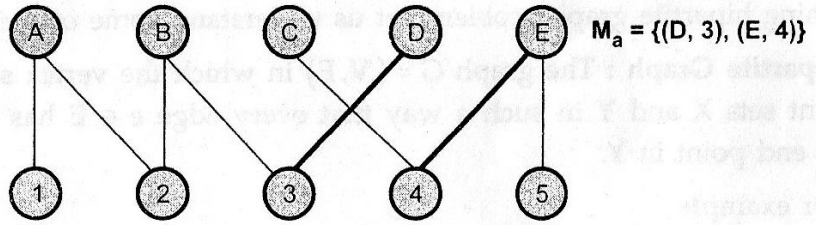


Fig. 6.4.2 Bipartite graph G

Here the vertices D, 3, E and 4 are **matched** and vertices A, 1, B, 2, c, 5 are **free** or unmatched vertices.

By adding a pair (A, 2) for matching

We get larger matching Ma

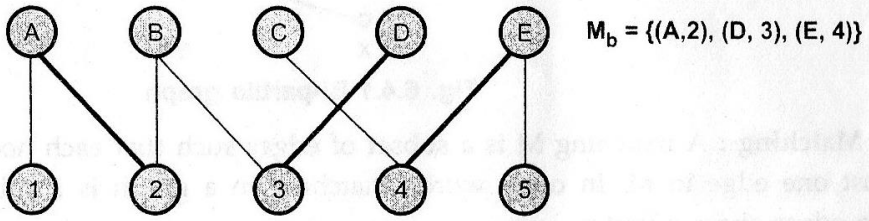


Fig. 6.4.3 Augmenting path : A,2

Something to get larger matching, for inclusion of some pair, we may need removal of existing pair. Such a matching adjustment is called **augmentation.**

Following figures illustrate this concept

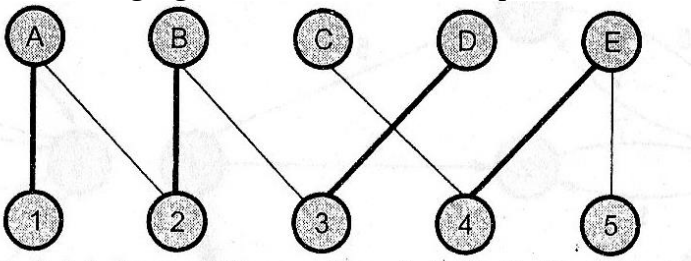
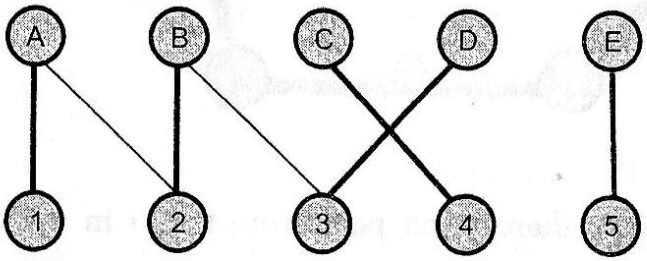


Fig. 6.4.4 Augmenting path : A,1, B, 2, D, 3, E, 4



Thus we have got maximum matching. This is also called as **perfect matching** because all vertices of graph are matched.

Theorem:

A matching M is a maximum matching if and only if there exists no augmenting path with respect to M.

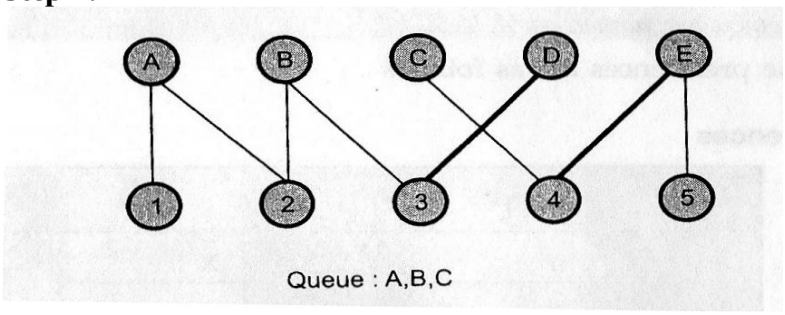
Algorithm Maximum Bipartite Matching(G)

initialize set M of edges // can be the empty set
initialize queue Q with all the free vertices in V
while not Empty (Q) **do**
w < Front(Q)
if w ε V **then**
for every vertex u adjacent to w **do** // u must be in U
if u free **then** // initialize set M of edges // can be the empty set
initialize queue Q with all the free vertices in V
while not Empty (Q) **do**
w < Front(Q)
if w ε V **then**
for every vertex u adjacent to w **do** // u must be in U
augment
M< M union (w, u)
v< w
while v is labeled **do** // follow the augmenting path
u< label of v
M< M – (v,) // (v, u) was in pervious M

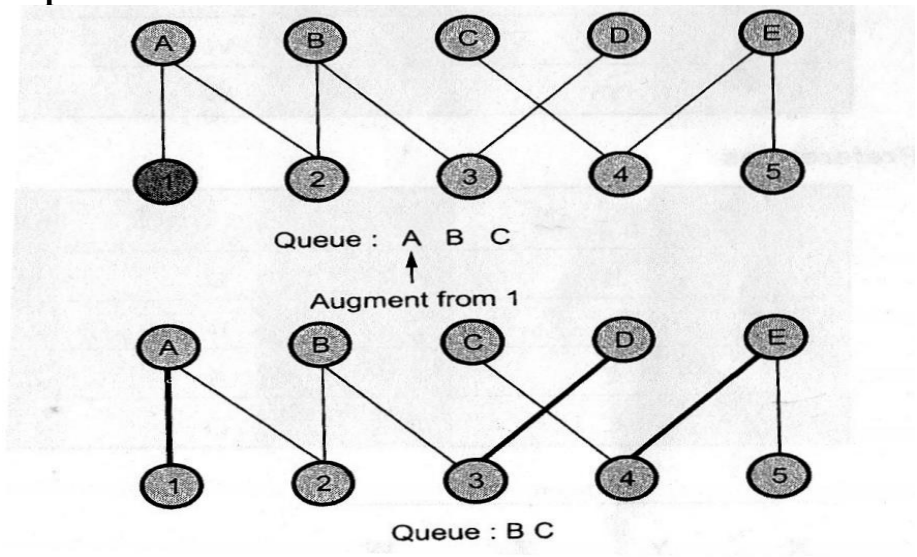
V< label of u
M< M union (v, u) // add the edge to the path
// start over vertex labels
reinitialize Q with
remove all
all the free vertices in V
break // exit for loop
else// u is matched
if (w, u) not in M **and** u is unlabeled **then**
label u with w // represents an edge in E-M
Enqueue(Q, u)
// only way for a U vertex to enter the queue
else // w ع U and therefore is matched with v
V <w's mate // (w, v) is in M
Label v with w // represents in M
Enqueue(Q, v) // only way for a mated v to enter Q
Return M // maximum matching

Application of Algorithm

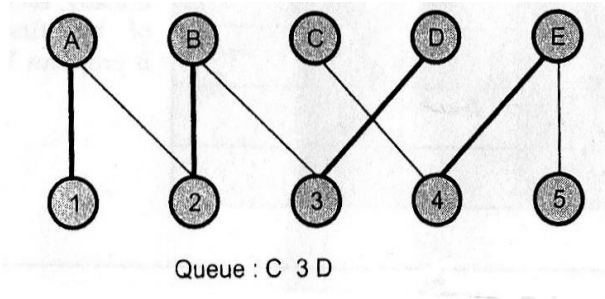
Step 1:



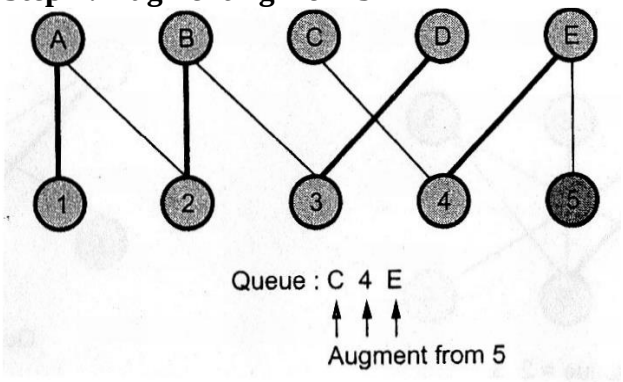
Step 2:



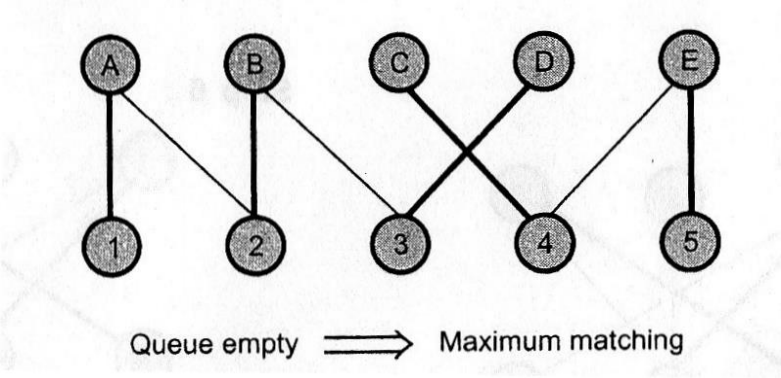
Step 3: Augmenting from 2



Step 4: Augmenting from 5

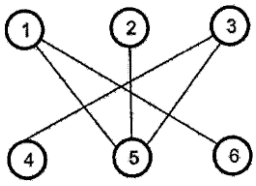


Step 5:



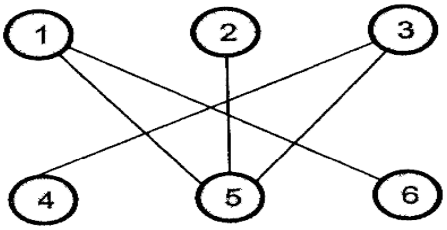
This is also a perfect matching.

Example : Apply the maximum matching algorithm to Following bi-partite graph.



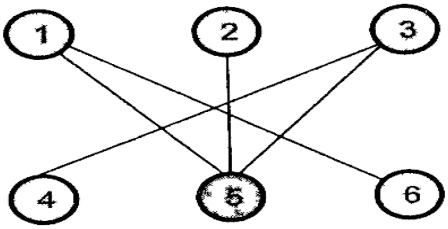
Solution:

Step 1:



Queue = 1 2 3

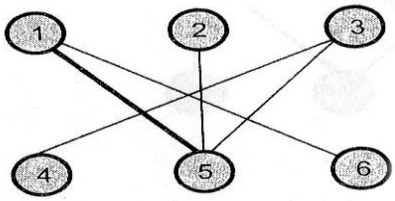
Step 2:



Queue 1 2 3

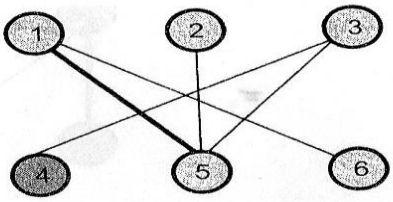
Augment from 5

Step 3:



Queue = 2 3

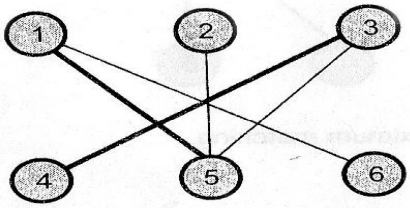
Step 4:



Queue 2 3 5

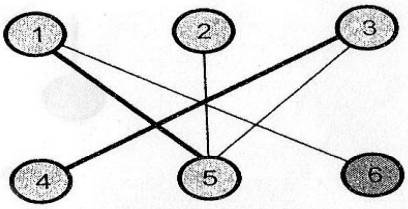
Augment from 4

Step 5:



Queue 2

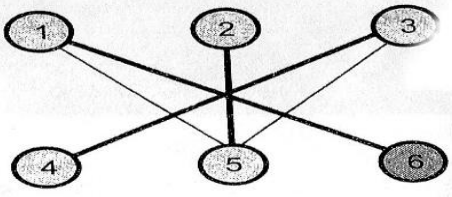
Step 6:



Queue : 2 5 1

Augment from 6

Step 7:



Queue empty
Maximum matching

5. Explain the Stable Marriage Problem with illustrative example(AU MAY 2015) or what is

stable marriage problem? Give the algorithm and analyze it. Nov/Dec 2017,2019 2021,18 apr-18, (or) Outline the steps in the stable marriage algorithm with an example. Nov/Dec 2021.

The stable marriage problem is an important algorithmic version of bipartite matching problem.

The goal of this problem is to find stable matching between two sets (men and women) with various preferences to each other.

The problem can be stated as follows –

Consider two sets M={m1, m2,…,mn} of n men and W={w1,w2,…,wn} of n women. Each man has a preference list ordering the women as potential marriage partners with no ties allowed. Similarly, each woman has a preference list of the men, also with no ties. Then we have to find out the marriage matching pair (m, w) whose members are selected from these two sets based on their preferences.

There is a set Y = $\{m_1,\ldots,m_n\}$ of n men and a set X = $\{w_1,\ldots,w_n\}$ of n women. Each man has a ranking list of the women, and each woman has a ranking list of the men (with no ties in these lists).

A *marriage matching* M is a set of n pairs (m_i, w_j).

A pair (m, w) is said to be a *blocking pair* for matching M if man m and woman w are not matched in M but prefer each other to their mates in M.

A marriage matching M is called *stable* if there is no blocking pair for it; otherwise, it's called *unstable*.

The *stable marriage problem* is to find a stable marriage matching for men's and women's given preferences.

An instance of the stable marriage problem can be specified either by two sets of preference lists or by a ranking matrix, as in the example below.

men's preferences	women's preferences
1st 2nd 3rd	1st 2nd 3rd
Bob: Lea Ann Sue	Ann: Jim Tom Bob
Jim: Lea Sue Ann	Lea: Tom Bob Jim
Tom: Sue Lea Ann	Sue: Jim Tom Bob

ranking matrix

	Ann	**Lea**	**Sue**
Bob	2,3	1,2	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

{(Bob, Ann) (Jim, Lea) (Tom, Sue)} is unstable
{(Bob, Ann) (Jim, Sue) (Tom, Lea)} is stable

Freemen:
Bob, Jim, Tom

Bob proposed to Lea
Lea accepted

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

	Ann	Lea	Sue

Bob	2,3	1,2	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

Freemen:
Jim, Tom

Jim proposed to Lea
Lea rejected

Freemen:
Jim, Tom
Jim proposed to Sue
Sue accepted

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

Freemen: Tom
Tom proposed To Sue Sue rejected

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

Freemen:
Tom
Tom proposed to Lea
Lea replaced Bob with Tom

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

Freemen:
Bob

Bob proposed to Ann
Ann accepted

The algorithm terminates after no more than n^2 iterations with a stable marriage output

The stable matching produced by the algorithm is always *man-optimal*: each man gets the highest rank woman on his list under any stable marriage. One can obtain the *woman-optimal* matching by making women propose to men

A man (woman) optimal matching is unique for a given set of participant preferences

Algorithm

Step 1: Initially all the men and all the women are free, but having their preferences list with them.

Step 2:

Propose: One of the free man m proposes to women w. This woman is normally the highest preferred one from his preference list.

Response: If the woman w is free then she accepts the proposal of matched m. If she is not free, she compares the m with her current mate. If she prefers m with current mate then she accepts his proposal making former mate free otherwise simply rejects m's proposal.

Step 3: Finally , returns the matching pairs of (m,w)

6. Maximize p=2x+3y+z Subject to x+y+z<=40
2x+y-z>=10
-y+z>=10
x>=0,y>=0,z>=0 (AU MAY 2015)

How to Solve General Maximization Problems

General maximization problems are linear programming problems in which you are asked to *maximize* an objective function. It may be non-standard: one or more of the constraints may be a ≥ constraint.

Rewrite the following LP problem as a system of linear equations.

Maximize p = 2x + 3y + z subject to
x + y + z ≤40
2x + y - z ≥10
- y + z ≥10
x ≥0, y ≥0, z ≥0

Use slack or surplus variables s, t and u respectively, and type all equations with the variables in the order shown above

The first constraint is x + y + z ≤40.

To turn it into an equation, we must add a slack variable s to the left-hand side, getting x + y + z + s = 40.

The next constraint is 2x + y - z ≥10,

and we must subtract the surplus variable t to the left-hand side, getting

2x + y - z - t = 10.

The last constraint is - y + z ≥10,

and we must subtract the surplus variable u to the left-hand side, getting

- y + z - u = 10.

Finally, the objective is p = 2x + 3y + z.

We must subtract 2x + 3y + z from both sides to get the desired equation:

-2x - 3y - z + p = 0.

Step 2: *Set up the initial tableau.*

Step 1 We convert the LP problem into a system of linear equations by putting in the slack variables and rewriting the objective function:

x		+		y	+	z	+	s			=	40
2z		+		y	-	z		- t			=	10
		-		y	+	z			- u		=	10
	2x			3y		z				+ p	=	0

Thus, the initial tableau is as follows.

	x	y	z	s	t	u	p	Ans
s	1	1	1	1	0	0	0	40
t	2	1	-1	0	-1	0	0	10
u	0	-1	1	0	0	-1	0	10
p	-2	-3	-1	0	0	0	1	0

The current tableau is

	x	y	z	s	t	u	p	Ans
s	1	1	1	1	0	0	0	40
t	2	1	-1	0	-1	0	0	10
u	0	-1	1	0	0	-1	0	10
p	-2	-3	-1	0	0	0	1	0

Reading across the first row (active variable s), we find s = 40/1 = 40.
Reading across the second row (active variable t), we find t = 10/(-1) = -10.
Reading across the third row (active variable u), we find u = 10/(-1) = -10.
Reading across the bottom row (active variable p), we find p = 0/1 = 0
Since all the other variables are inactive, their values are zero.

Notice that the values of the surplus variables t and u are currently negative. This is not permitted, since all variables are required to be *non-negative.* This tells us that the current basic solution (x, y, z) = (0, 0, 0) is *not feasible,* (it does not satisfy the second and third constraints). We use asterisks to mark the rows corresponding to those negative basic variables:

	x	y	z	s	t	u	p	Ans
s	1	1	1	1	0	0	0	40
***** **t**	2	1	-1	0	-1	0	0	10
***** **u**	0	-1	1	0	0	-1	0	10
p	-2	-3	-1	0	0	0	1	0

Our first order of business is to get into the feasible region, or, equivalently,

Phase I: ***Get rid of the stars.***

We can (eventually) get rid of all the stars by pivoting one or more times. The only way this differs from the procedure for pivoting in standard maximization problems is the way in which we select the pivot column.

- For standard maximization problems, the pivot column was chosen by selecting the most negative number in the bottom row.
- In Phase I, on the other hand, the pivot column is chosen by selecting *the largest positive number in the first starred row.*

Once we have selected the pivot column, we select the actual pivot as usual by using the lowest test ratio in the pivot column. (**Note:** If the lowest ratio occurs both in a starred row and an unstarred row, pivot in a starred row rather than the unstarred one.) Thus, in Phase I, we don't worry about negative numbers in the bottom row at all (there might not even be any there to begin with). That is all there is to Phase I.

In the above tableau, the first starred row is Row 2, and the largest positive number it contains is its first entry, 2. Thus, we have the following pivot column, colored in green

	x	**y**	**z**	**s**	**t**	**u**	**p**	**Ans**	
s	**1**	1	1	1	0	0	0	40	Test ratio 40/1 = 40
*** t**	**2**	1	-1	0	-1	0	0	10	10/2=5 (Smallest).
*** u**	**0**	-1	1	0	0	-1	0	10	
p	**-2**	-3	-1	0	0	0	1	0	

This is the first starred row.

To select the pivot, identify the smallest test ratio: Test ratio 40/1 = 40

Test Ratio: 10/2 = 5 (Smallest).

Thus, we pivot on the "2" in Row 2, and we get the following tableau

	x	**y**	**z**	**s**	**t**	**u**	**p**	**Ans**
s	0	1	3	2	1	0	0	70
X	2	1	-1	0	-1	0	0	10
*** u**	0	-1	1	0	0	-1	0	10
p	0	-2	-2	0	-1	0	1	10

Q What happened to the star in Row 2?

A It is gone, because the basic solution we now get from Row 2 is

x = 10/2 = 5,

which is no longer negative! Thus, we have eliminated one of the stars.

Since there is still one starred row left, we are not done with Phase I.

From the above curent tableau

Since Row 3 is the only starred row (and so it is the first starred row) we locate the largest positive number in Row 3 (it is the "1" in the z-column), giving us the pivot column (shown in red):

	x	**y**	**z**	**s**	**t**	**u**	**p**	**Ans**	
s	0	1	**3**	2	1	0	0	70	Test Ratio: 70/3
X	2	1	**-1**	0	-1	0	0	10	
*** u**	0	-1	**1**	0	0	-1	0	10	Test Ratio: 10/1
p	0	-2	**-2**	0	-1	0	1	10	

Since the smallest test ratio is in the u-row (Row 3), we select the entry in Row 3 Column 3 as pivot. Using the correct pivot, now obtain the second tableau.

	x	y	z	s	t	u	p	Ans	
s	0	1	**3**	2	1	0	0	70	R1-3R3
X	2	1	**-1**	0	-1	0	0	10	R2+3R3
*** u**	0	-1	**1**	0	0	-1	0	10	
p	0	-2	**-2**	0	-1	0	1	10	R4+2R3

This gives

	x	y	z	s	t	u	p	Ans
s	0	4	0	2	1	3	0	40
X	2	0	0	0	-1	-1	0	20
Z	0	-1	1	0	0	-1	0	10
p	0	-4	0	0	-1	-2	1	30

The basic solution we now get from Row 3 is z = 10/1 = 10,
which is no longer negative, so the star disappears.

Q Ok the stars are gone. Now what?

A Since there are no more stars, we are now in the feasible region, and can proceed to:

Phase II: *Do the simplex method as for standard maximization problems.*

Revert back to selecting the pivot column using the most negative number in the bottom row (excluding the Answer column). Continue pivoting until there are no negative numbers in the bottom row (with the possible exception of the Answer column).

If you look at the tableau you just obtained, you will see that there is still a negative number there: the -4 in the "y"-column, (press the above Help button to see the current tableau if you do not have it) you will now need to select a pivot in that column in order to pass to the next tableau. Using our usual rule for selecting a pivot in a given column, we see that the pivot is the "4" in Row 1 Column 2, so we pivot on that getting

	x	y	z	s	t	u	p	Ans
Y	0	4	0	2	1	3	0	40
X	2	0	0	0	-1	-1	0	20
Z	0	0	4	2	1	-1	0	80
p	0	0	0	0	0	1	1	70

7. (i) Summarize the Simplex method.(8) Apr-18

Step 0 [Initialization] Present a given LP problem in standard form and set up initial tableau.

Step 1 [Optimality test] If all entries in the objective row are nonnegative then stop: the Tableaure presents an optimal solution.

Step 2 [Find entering variable] Select the most negative entry in the objective row. Mark its column to indicate the entering variable and the **pivot column**.

Step 3 [Find departing (leaving) variable] For each positive entry in the pivot column, Calculate the θ-ratio by dividing that row's entry in the rightmost column (solution) by its entry in the pivot column.

(If there are no positive entries in the pivot column then stop: the problem isunbounded.)

Find the row with the smallest θ-
ratio, mark this row to indicate the departingvariable and the **pivot row.**

Step 4 [Form the next tableau] Divide all the entries in the pivot row by its entry in the

Pivot column. Subtract from each of the other rows, including the objective row, the new pivot row multiplied by the entry in the pivot column of the row in question. Replace the label of the pivot row by the variable's name of the pivot column and go back to Step 1.

Standard form of LP problem

Must be a **maximization** problem

All constraints (except the nonnegativity constraints) must be in the form of linearEquation sAll the variables must be required to be nonnegative

Thus, the general linear programming problem in standardform with m constraints and n unknowns ($n \geq m$) is

Maximize $c1\,x1 + ... + cn\,xn$

Subject to $ai\ 1x1 + ... + ain\,xn = bi\,,\ i = 1,...,m, x1 \geq 0, ... , xn \geq 0$

Example

maximize $3x + 5y$

$x + 3y \leq 6$

$x \geq 0,\ y \geq 0$

maximize $3x + 5y + 0u + 0v$subject to

$x + y \leq 4$ subject to $x + y + u = 4$

$x + 3y + v = 6$

$x \geq 0,\ y \geq 0,\ u \geq 0,\ v \geq 0$

Variables u and v, transforming inequality constraints into equality constrains, are called *sl ackvariables*

(ii) State and Prove Maximum Flow Min cut Theorem. (or) Elaborate the maximum flow problem with an example and relevant diagrams (8) (Nov/Dec 2021)

Maximum Flow Problem

Problem of maximizing the flow of a material through a transportation network (e.g., pipeline system, communications or transportation networks)

Formally represented by a connected weighted digraph with n vertices numbered from 1 to n with the following properties:

- Contains exactly one vertex with no entering edges, called the ***source*** (numbered 1)
- Contains exactly one vertex with no leaving edges, called the ***sink*** (numbered n)

Has positive integer weight uij on each directed edge $(i.j)$, called the***edge capacity,*** indicating the upper bound on the amount of the material that can be sent from i to j through this edge.

A digraph satisfying these properties is called a **flow network** or simply a network

Flow value and Maximum Flow Problem

Since no material can be lost or added to by going through intermediate vertices of the network, The total amount of the material leaving the source must end up at the sink:

$$\sum_{j:\,(1,j)\,\epsilon\,E} x1j \;=\; \sum_{j:\,(j,n)\,\epsilon\,E} xjn$$

The *value* of the flow is defined as the total outflow from the source (= the total inflow into thesink). The *maximum flow problem* is to find a flow of the largest value (maximum flow) for a givennetwork.

Max-Flow Min-Cut Theorem

1. The value of maximum flow in a network is equal to the capacity of its minimum cut
2. The shortest augmenting path algorithm yields both a maximum flow and a minimum cut:

Maximum flow is the final flow produced by the algorithm

Minimum cut is formed by all the edges from the labeled vertices to unlabeledvertices on t he last iteration of the algorithm.

All the edges from the labeled to unlabeled vertices are full, i.e., their flow amounts re equal to the edge capacities, while all the edges from the unlabeled to labeled vertices, if any, have zero flow amounts on them.

PART-A

32. What is state space tree?

Backtracking and branch bound are based on the construction of a state space tree, whose nodes reflect specific choices made for a solution's component .Its root represents an initial state before the search for a solution begins. The nodes of the first level the tree represent the made for the first component of solution, the nodes of the second evel represent the Choices for the second components & so on

33. State Extreme point theorem.

Any linear programming problem with a nonempty bounded feasible region has an optimal solution; moreover, an optimal solution can always be found at an extreme point of the problem's feasible region.

Extreme point theorem states that if S is convex and compact in a locally convex space, then S is the closed convex hull of its extreme points: In particular, such a set has extreme points. Convex set has its extreme points at the boundary. Extreme points should be the end points of the line connecting any two points of convex set.

34. Define the iterative improvement technique

Greedy techniques iteratively construct an optimal solution by building optimal solutions from smaller problems. Iterative improvement techniques build an optimal solution by iterative refinement of a feasible solution for the complete problem.

Feasible solutions are solutions that satisfy the **constraints** of the problem, for example using the denominations in the making change problem.

The **objective function** is the function that problem seeks to maximize or minimize.

Iterative improvement is frequently used in numerical problems, for example root finding or finding the maximum of a function. We will concentrate on iterative improve to graph problems.

Iterative improvements have difficulties:

1. Finding the initial solution (guess to the solution) can be easy, for example the empty set, or on the other hand it can be difficult.
2. The algorithm for refinements the guess may be difficult. The refinement must remain feasible and improve the objective function. Meaning they should not jump around and possibly diverge from the optimal solution.
3. The refinement may find a local optimal solution but not the global optimal solution, for example find the maximum by always going up hill.

IMPORTANT QUESTION

UNIT-4(NEW SYLLABUS)

PART-A

1. What is feasible solution and feasible region? Define optimal solution.
2. When the feasible region in linear programming becomes unbounded?
3. Define extreme point theorem.
4. What are the requirements for linear programming problem?
5. Write the general linear programming problem in standard form.
6. What is basic and non basic solution?
7. Define objective row.
8. Define maximum flow problem.
9. Define flow network.
10. What is flow conservation requirement?
11. Define Max-Flow Min-Cut Theorem.
12. Define preflow.
13. What is maximum cardinality matching?
14. Define bipartite graph
15. Define stable marriage problem.
16. What is iterative improvement method?
17. Enlist various applications of iterative improvement method?
18. What is linear programming problem?
19. What is bipartite graph? Colorable graph?
20. Wat is maximum cardinality matching? Matching problem?
21. What is entering variable? AND what is departing variable

PART – B

1.Solve the following linear programming problem geometrically

Maximize $3x + y$
Subject to $-x + y \leq 1$
$2x + y \leq 4$
$x \geq 0, y \geq 0$

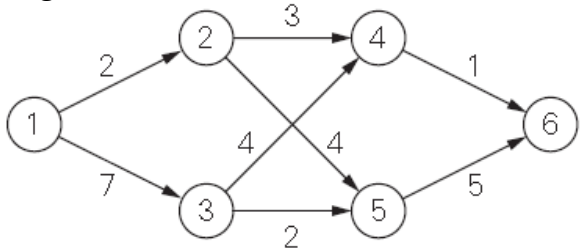
2.Trace the simplex method on the following problem.

Maximize $p=2x - 3y + 4z$
Subject to $4x - 3y + z \leq 3$
$x + y + z \leq 10$
$2x + y - z \leq 10$ where x, y and z are non negative. $\leq \geq$

3.Trace the following non-standard problem using simplex method.

Maximize $p = 2x - 3y + 4z$
Subject to $4x - 3y + z \leq 3$
$x + y - z \geq 5$
$x \geq 0, y \geq 0, z \geq 0$

4.Apply the shortest-augmenting-path algorithm to find a maximum flow and a minimum cut in the following networks.



5.Explain the maximum flow problem in detail with example. Nov/Dec 2021

6.Explain the stable marriage problem with example.

ANNA UNIVERSITY APRIL/MAY 2015

Part-A

1. What do you mean by perfect matching in bipartite graph? **Refer Part A Q. No. 26**
2. Define flow cut **Refer Part A – Q. No. 27**

Part-B

1.a(i)Maximize p=2x+3y+z
Subject to x+y+z<=40
2x+y-z>=10
-y+z>=10
x>=0,y>=0,z>=0 **Refer Part B – Q. No. 5**

(ii) Write down the optimality condition and algorithmic implementation for finding M-augmentating paths in bipartite graphs **Refer Part B – Q. No. 3**

b. Briefly describe on the stable marriage problem **Refer Part B – Q. No. 4**

ANNA UNIVERSITY NOV/DEC 2015

PART-A

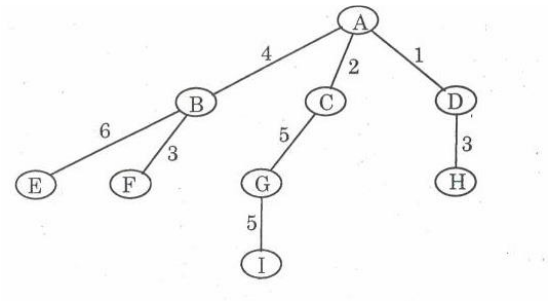
1. Determine the dual linear program for the following LP **Refer Part A Q. No. 19**
2. Determine Network Flow and cut **Refer Part A Q. No. 27**

PART-B

12.a.(i) use simplex to solve the farmers problem **Refer Part B – Q. No. 3**

(ii) Write the procedure to initialize simplex which determine if a linear simplex which determines if a linear program is feasible or not? **Refer Part B – Q. No. 3**

b.(i) illustrate the working of the maximum matching algorithm on the following weighted tree



(ii) Explain max-Flow problem (4) **Refer Part B – Q. No.4**

ANNA UNIVERSITY APRIL/MAY 2016

PART-A

1. What is state space tree? **APRIL/MAY 2016**
2. State Extreme point theorem. **APRIL/MAY 2016**

PART-B

1. (a) (i) Summarize the Simplex method.8 **APRIL/MAY 2016**

(ii) State and Prove Maximum Flow Min cut Theorem. (8) **APRIL/MAY 2016**

OR

(b) Apply the shortest Augmenting Path algorithm to the network shown below. (16) **APRIL/MAY 2016**

ANNA UNIVERSITY NOV/DEC 2016

PART-A

1. Define the iterative improvement technique. **NOV/DEC 2016**
2. what is maximum cardinality matching? **NOV/DEC 2016**

PART-B

1. a. (i) State and prove Max-Flow Min-Cut Theorem. **NOV/DEC 2016**
   (ii)Summarize the steps of the simplex method.(16) **NOV/DEC 2016**

   b. (i) Explain briefly about Stable Marriage Problem.(10) **NOV/DEC 2016**
   (ii) Determine the Time-efficiency class of the stable marriage algorithm.(6) **NOV/DEC 2016**

ANNA UNIVERSITY APRIL/MAY 2017

PART-A

1. What do you mean by perfect matching in bipartite graph? **APRIL/MAY 2017**
2. State: Planar coloring graph problem. **APRIL/MAY 2017**

PART-B

1. Describe in detail the simplex algorithm method **APRIL/MAY 2017**
2. Explain KMP string matching algorithm for finding a pattern on a text and analyze the algorithm **APRIL/MAY 2017**

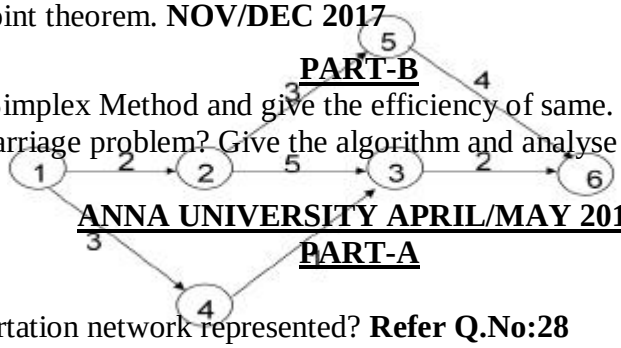
ANNA UNIVERSITY NOV/DEC 2017

PART-A

1. What are Bipartite graphs. **NOV/DEC 2017**
2. State Extreme Point theorem. **NOV/DEC 2017**

PART-B

1. List the steps of Simplex Method and give the efficiency of same. **NOV/DEC 2017**
2. What is Stable marriage problem? Give the algorithm and analyse it **NOV/DEC 2017**



ANNA UNIVERSITY APRIL/MAY 2018

PART-A

1. How is a transportation network represented? **Refer Q.No:28**
2. What is meant by maximum cardinality matching? **Refer Q.No:22**

PART-B

1. Give the summary of the simplex method. **Refer Q.No:7**
2. Prove that the stable marriage algorithm terminates after no more than n2 iterations with a stable marriage output. **Refer Q.No:5**

ANNA UNIVERSITY NOV/DEC 2018

PART-A

1. Describe iterative improvement technique. **Refer Q.No:17**
2. What is solution space? Give an example. **Refer Q.No:29**

PART-B

1. Illustrate the steps of the simplex methods with an example. (13) **Refer Q.No:1**
2. Write the stable marriage algorithm and trace it with an instance. Analyz1e its running time complexity. (13) **Refer Q.No:5**

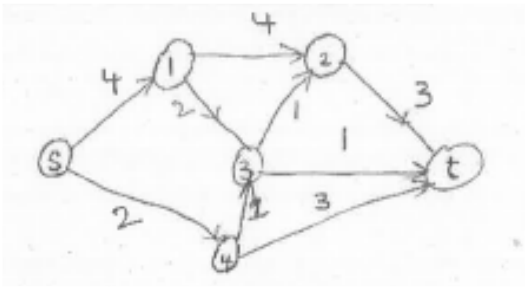
ANNA UNIVERSITY APRIL/MAY 2019

PART-A

1. State the principle of duality? **Refer Q.No.31**
2. Define the capacity constraint in the context of maximum flow problem? **Refer Q.No.30**

PART-B

1. Determine the max-flow in the following network. **Refer Q.No.2**



2. Solve the following set of equations using simplex algorithm: **Refer Q.No.1.A**
   Maximize:18 x1 +12.5 x2
   Subject to :x1 + x2 <= 20
   x1<=12
   x2<=16
   x1,x2>=0

ANNA UNIVERSITY NOV/DEC 2019

PART-A

1. When a linear program is said to be unbounded? **Refer Q.No.5**
2. What is a residual network in the context of flow networks? **Refer Q.No.27**

PART-B

1.what is iterative improvement? Elaborate the steps in the simplex method with an example(13)

Refer Q.No.1

2 (i) what is a bipartite graph? Is the subset of a bipartite graph bipartite? Outline the with an example. **Refer Q.No.4**

(ii) Outline the stable marriage problem with an example.(13) **Refer Q.No.5**

ANNA UNIVERSITY APRIL/MAY 2021

PART-A

1. What are the essential properties of a flow graph? **Refer Q.No.9**

2. What is meant by iterative improvement technique? **Refer Q.No.17**

PART-B

1.a) i) Find a stable marriage matching for the instance defined by the following ranking matrix : **Refer Q.No.5**

(ii) Determine the time efficiency of the above algorithm in the worst case.

	A	B	C	D
α	1, 3	2, 3	3, 2	4,3
β	1, 4	4, 1	3, 4	2, 2
γ	2, 2	1, 4	3, 3	4, 1
δ	4, 1	2, 2	3, 1	1, 4

b) (i) Advertising alternatives for a company include television, radio and newspaper. The table below shows the costs and estimates of audience coverage for each types of media.

	Cost per advertisement	Audience per advertisement
Television	$2000	100,000
Newspaper	$600	40,000
Radio	$300	18,000

The newspaper limits the number of weekly advertisements form a single company to ten. Moreover to balance the advertising among the three types of media, no more than half of the total number of advertisements should occur on the radio, and at least 10% should occur on television. The weekly advertising budget is $ 18,200. How many advertisements should run in each of the three types of media to maximize the total audience? Solve the problem using simplex method. **Refer Q.No.1**

PART C

(i) Explain the maximum-bipartite-matching problem with an illustration. **Refer Q.No.4**

ANNA UNIVERSITY NOV/DEC 2021

PART-A

1. Outline the steps in iterative improvement. **Refer Q. No.17**
2. Define a bipartite graph. **Refer Q.No.14**

PART-B

1. Elaborate the maximum flow problem with an example and relevant diagrams **Refer Q.No. 7**
2. Outline the steps in the stable marriage algorithm with an example. **Refer Q.No.5**

UNIT II
BRUTE FORCE AND DIVIDE-AND-CONQUER

Brute Force – Computing an – String Matching - Closest-Pair and Convex-Hull Problems - Exhaustive Search - Travelling Salesman Problem - Knapsack Problem - Assignment problem. Divide and Conquer Methodology – Binary Search – Merge sort – Quick sort – Heap Sort - Multiplication of Large Integers – Closest-Pair and Convex - Hull Problems.

PART A

1. Define Brute Force.

Brute Force is a straightforward approach to solve a problem, which is directly based on the problem statement and definition of the concepts. Brute Force strategy is one of the easiest approach.

2. What are the types of Brute Force Algorithm?

There are two types of Brute force algorithm. They are Consecutive integer checking algorithm for computing the greatest common divisor of two integer[gcd(m,n)] Definition based algorithm for matrix multiplication.

3. What are the Advantages of Brute Force Approach?Advantages

It is applicable to a variety of problems.A brute Force algorithm can be useful for solving small size instances of a problem.

A brute Force algorithm can serve an important theoretical or educational purpose.

The brute force approach provides reasonable algorithms of atleast some practical value with no limitation on instance size for sorting, searching, matrix multiplication and string matching problem.

4. What is Closest-Pair Problem? (Apr/May-2017)

The closest-pair problem calls for finding the two closest points in a set of n points.

It is the simplest of a variety of problems in computational geometry that deals with proximity of points in the plane or higher-dimensional spaces.

5. What is meant by Convex?

A set of points (finite or infinite) in the plane is called **convex** if for any two points p and q in the set, the entire line segment with the endpoints at p and q belongs to the set.

6. What is meant by convex hull?or state the convex hull problem. Nov/Dec 2019

The **convex hull** of a set S of points is the smallest convex set containing S. (The "smallest" requirement means that the convex hull of S must be a subset of any convex set containing S.)

7. Define exhaustive search.

An **exhaustive search**, also known as **generate and test**, is a very general problem-solving technique that consists of systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement.

8. What is Travelling Salesman Problem?

The **Travelling Salesman Problem** (**TSP**) is an NP-hard **problem** in combinatorial optimization studied in operations research and theoretical computer science.

Given a list of cities and their pairwise distances, the task is to find a shortest possible tour that visits each city exactly once.

9. What is knapsack? Or outline the knapsack problem Nov/Dec 2019

The knapsack problem, another well-known NP-hard problem.The Knapsack problem is, given n items of known weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ and a knapsack of weight capacity W, find the most valuable subset of the items that fits into the knapsack.

10.What is assignment problem?

The **assignment problem** is one of the fundamental combinatorial optimization **problems** in the

branch of optimization or operations research in mathematics.
It consists of finding a maximum weight matching in a weighted bipartite graph.

11. Define the divide and conquer method. Or **Write the general divide and conquer approach to solve a problem. April/May 2021**

Divide & conquer technique is a top-down approach to solve a problem.
The algorithm which follows divide and conquer technique involves 3 steps:

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

12. What is the binary search?

If 'q' is always chosen such that 'a_q' is the middle element (that is, q = [(n+1)/2)], then the resulting search algorithm is known as binary search.

13. What is the time complexity of Binary search? *June 2011 & 12*

ssful searches			cessful searches
	n)	n)	n)
	ge		Average, Worst

14. Define external path length?

The external path length E, is defines analogously as sum of the distance of all external nodes from the root.

15. Define internal path length.

The internal path length 'I' is the sum of the distances of all internal nodes from the root.

16. Is insertion sort better than the merge sort?

Insertion sort works exceedingly fast on arrays of less then 16 elements, though for large 'n' its computing time is O (n^2).

17. Give the recurrence relation of divide-and-conquer?

The recurrence relation is

$$\begin{cases} T(n) = g(n) & n \text{ is small} \\ T(n1) + T(n2) + T(nk) + f(n) & \text{otherwise} \end{cases}$$

Where T(n) is the time for DAndC on any input of size n and g(n) is the time to compute the answer directly for small inputs.
The function f(n) is the time for dividing P and combining the solutions to subproblems.

18. What are internal nodes?

The circular node is called the internal nodes.

19.Give the recurrence equation for the worst case behavior ofmerge sort?

The recurrence equation for the worst case behavior of merge sort is $T(n) = 2T(n/2) + cn$ for $n>1$, c is a constant
Total number of comparison required by the merge sort is $\Theta(n \log n)$

20. Find the number of comparisons made by the sequential search in the worst case and best case?

Worst case: The algorithm makes the largest number of key comparisons among all possible input of size n. $C_{worst(n)}=n$

Best Case: The best case inputs will be lists of size n with their first element equal to search key. $C_{best(n)}=1$

21. What are the objectives of sorting algorithms?

A sorting algorithm is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order.

Efficient sorting is important for optimizing the use of other algorithms (such as search and merge algorithms) that require sorted lists to work correctly;

More formally, the output must satisfy two conditions:The output is in non-decreasing order (each element is no smaller than the previous element according to the desired total order)
The output is a permutation (reordering) of the input.

22. What do you meant by Divide and conquer strategy? *May 2013*

Divide & conquer technique is a top-down approach to solve a problem.The algorithm which follows divide and conquer technique involves 3 steps:

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

23. What are the merits of binary search?

A binary search or half-interval search algorithm finds the position of a specified value (the input "key") within a sorted array.

In each step, the algorithm compares the input key value with the key value of the middle element of the array.

A binary search halves the number of items to check with each iteration, so locating an item (or determining its absence) takes logarithmic time It is faster than the sequential search.

It requires lesser number of key comparisons than the sequential search.

24. Is merge sort stable sorting algorithm?

Yes, merge sort is the stable sorting algorithm.

A sorting algorithm is said to be stable if it preserves the ordering of similar elements after applying sorting method.

And merge sort is a method which preserves this kind of ordering. Hence merge sort is a stable sorting algorithm.

25. Give efficiency analysis of divide and conquer?

The efficiency of divide and conquer algorithms is given by recurrences of the form.

$$T(n) = \begin{cases} T(n) & n=1 \\ aT(n/b) + f(n) & n>1 \end{cases}$$

Where a and b are known constants. We assume that T(1) is known and n is a power of b ($n=b^k$).

26. What is the idea behind binary search?

A binary search or half-interval search algorithm finds the position of a specified value (the input "key") within a sorted array.

In each step, the algorithm compares the input key value with the key value of the middle element of the array.

If the keys match, then a matching element has been found so its index, or position, is returned.

Otherwise, if the sought key is less than the middle element's key, then the algorithm repeats its action on the sub-array to the left of the middle element or, if the input key is greater, on

the sub-array to the right.

If the remaining array to be searched is reduced to zero, then the key cannot be found in the array and a special "Not found" indication is returned.

27. Give the time efficiency and drawback of merge sort algorithm? *Dec 2005*

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 . \end{cases}$$

Refer Class Notes

28. What is the difference between quick sort and merge sort? *May 2013*

Sequential technique	**binary search technique**
This is the simple technique of searching an element	This is the efficient technique of searching an element
This technique does not require the list to be sorted	This technique require the list to be sorted. Then only this method is applicable
The worst case time complexity of this technique is O(n)	The worst case time complexity of this technique is O(log n)
Every element of the list may get compared with the key element.	Only the mid element of the list is compared with key element.

29. What is the difference between sequential and binary search *Apr 2013*

Sequential technique	**binary search technique**
This is the simple technique of searching an element	This is the efficient technique of searching an element
This technique does not require the list to be sorted	This technique require the list to be sorted. Then only this method is applicable
The worst case time complexity of this technique is O(n)	The worst case time complexity of this technique is O(log n)
Every element of the list may get compared with the key element.	Only the mid element of the list is compared with key element.

30. What is the necessary precondition for the binary search ?

For the binary search the list should be sorted either in ascending or descending order

31. List out two drawbacks of binary search algorithm. *Dec 2007*

In binary search the elements have to be arranged either in ascending or descending order

Each time the mid elements has to be computed in order to partition the list in two sub lists

32. Give the control abstraction for divide and conquer. *Dec 2012*

```
divide_and_conquer ( P )
{
if ( small ( P ) ) // P is very small so that a solution is trivial
return solution ( n );
divide the problem P into k instances P1, P2, ..., Pk;
return ( combine ( divide_and_conquer ( P1 ),
divide_and_conquer ( P2 ),
...
divide_and_conquer ( Pk ) ) );
}
```

The solution to the above problem is described by the recurrence, assuming size of P denoted by n

$$T_n = \begin{cases} g(n) & n \text{ is small} \\ T_{n_1} + T_{n_2} + \cdots + T_{n_k} + f(n) \end{cases}$$

where f(n) is the time to divide n elements and to combine their solution.

33. What is called substitution method? *Jun 2010*

A substitution method is one, in which we guess a bound and then use mathematical induction to prove our guess correct.

It is basically two step process:

Step1: Guess the form of the Solution.

Step2: Prove your guess is correct by using Mathematical Induction.

Example 1.

Solve the following recurrence by using substitution method.

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

Solution:

Step1: The given recurrence is quite similar with that of MERGESORT, you guess the solution is

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

Or

$$T(n) \leq c.n \log n$$

Step2: Now we use mathematical Induction.

Here our guess does not hold for n=1 because

$$T(1) \leq c.1 \log 1$$

$$i.e. T(n) \leq 0 \text{ which is contradiction with } T(1) = 1$$

$$T(2) \leq c.2log2$$

$$2T\left(\frac{2}{2}\right) + 2 \leq c.2$$

$$2T(1) + 2 \leq c.2$$

$$0 + 2 \leq c.2$$

Now for n=2 $2 \leq c.2$ which is true. So $T(2) \leq c.nlogn$ is True for n=2

34. What is called optimal solution? *Jun 2010*

A feasible solution that maximizes the given objective function is called as optimal solution.

35. What do you mean by divide and conquer strategy? *Jun 2013* Refer Part A – Q. No. 11

36. State the principle of substitution method? *Jun 2014* Refer Part A – Q. No. 33

37. Define feasible and optimal solution? *Jun 2014*

Given n inputs form a subset such that it satisfies some given constraints then such a subset is called feasible solution.

A feasible solution that maximizes the given objective function is called as optimal solution

38. Trace the operation of binary search algorithm for the input – 15, -6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151, if you are searching for the element 9. *Dec 2010*

Input :

15	-6	0	7	9	23	54	82	101	112	125	131	142	151	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	

Iteration 0:

Left = 0
Right = 13
Mid = (Left + Right) / 2
= (0 + 13) / 2
Mid = 6
Midelement = 54
Search key = 9

Since 9 < 54, search the element 9 in the left of midelement 54.

Iteration 1:

Left = 0
Right = 5
Mid = (Left + Right) / 2
= (0 + 5) / 2
Mid = 2
Midelement = 0
Search key = 9

Since 9 > 0, search the element 9 in the right of midelement 0.

Iteration 2:

Left = 3

Right = 4
Mid = (Left + Right) / 2
= (3 + 4) / 2
Mid = 3
Midelement = 7
Search key = 9

Since 9 > 7, search the element 9 in the right of midelement 7.
Therefore 9 is found in the position 4.

39.Design a brute force algorithm for computing the value of a polynomial (April/may 2015)

Problem: Find the value of polynomial

$p(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x^1 + a_0$ at a point $x = x_0$

Algorithm:

$x := x_0$
p := 0.0
for i := n down to 0 do
power := 1
for j := power * x
$p := p + a$:= 1 to i do
power [i] * power
return p

Efficiency: $\Theta(n^2)$

40. Derive complexity of binary search algorithm. (AU april/may 2015)

Worst Case Analysis

The worst case includes all arrays that do not contain a search key.
The recurrence relation for

$C_{worst}(n) = C_{worst}(n/2) + 1$, for $n > 1$ ----- (1)

Time required to compare left sublist or right sub list

one comparison made with middle element

$C_{worst}(1) = 1$ -------- (2)

The above recurrence relation can be solved further .
assume $n=2^k$ the equation (1) becomes

$C_{worst}(2^k) = C_{worst}(2^k/2) + 1$
$C_{worst}(2^k) = C_{worst}(2^{k-1}) + 1$ ------ (3)

Using backward substation method , we can substitute
$C_{worst}(2^{k-1}) = C_{worst}(2^{k-2}) + 1$

Then equation (3) becomes
$C_{worst}(2^k) = [C_{worst}(2^{k-2}) + 1] + 1$
$C_{worst}(2^k) = C_{worst}(2^{k-2}) + 2$

Then
$C_{worst}(2^k) = [C_{worst}(2^{k-3}) + 1] + 2$
$C_{worst}(2^k) = C_{worst}(2^{k-3}) + 3$

$C_{worst}(2^k) = C_{worst}(2^{k-k}) + k$
$C_{worst}(2^k) = C_{worst}(2^0) + k$
$C_{worst}(2^k) = C_{worst}(1) + k$ ----- (4)

But as per equation (2)
as we have assumed $n = 2^k$ taking logarithm (base 2)on both sides
log 2 n = log 2 2^k
log 2 n = k. log 2 2
log 2 n = k(1) therefore log 2 2 =1
therefore k = log 2 n

$C_{worst}(1) = 1$ the we get equation (4)
$C_{worst}(2^k) = 1 + k$
$C_{worst}(n) = 1 + \log 2n$ ----- (2)

$C_{worst}(n) = \log 2n$ for n>1
The worst case time complexity of binary search is $\Theta(\log 2n)$
As $C_{worst}(n) = \log 2n + 1$
we can verify equation (1) with this value.
$C_{worst}(n) = C_{worst}[(n/2)] + 1$
In equation (1) put n = 2i
L.H.S

$C_{worst}(n) = \log 2n + 1$
$= \log 2(2i) + 1$
$= \log 2\ 2 + \log 2i + 1$
$= 1 + \log 2i + 1$
$= 2 + \log 2i$

$C_{worst}(n) = 2 + \log 2i$
R.H.S

$C_{worst}(n/2)+1 = \log 2(2i/2) + 1$
$= \log 2i + 1$
$= \log 2\ 2i + 1 + 1$
$= 2 + \log 2i$

$C_{worst}(n/2) = 2 + \log 2i$
L.H.S = R.H.S
Hence
$C_{worst}(n) = \log 2n + 1$ and
$C_{worst}(i) = \log 2i + 1$ are same
Hence
$C_{worst}(n) = \Theta(\log n)$

41. Give the General plan divide and conquer method. Nov/Dec 2017

Write the general divide and conquer approach to solve a problem. April/May 2021

A **divide and conquer algorithm** works by recursively breaking down a problem into twoor more subproblems of the same (or related) type (**divide**), until these become simple enough t obe solved directly (**conquer**).Divideandconquer algorithms work according to the following g eneral plan:

- ✓ A problem is divided into several subproblems of the same type, ideally of about equal size.
- ✓ The subproblems are solved (typically recursively, though sometimes a different algorithm is employed, especially when subproblems become small enough).
- ✓ If necessary, the solutions to the subproblems are combined to get a solution to the original problem.

Example: Merge sort, Quick sort, Binary search, Multiplication of Large Integers And Strassen's Matrix Multiplication.

42.write an algorithm for brute force closest-pair problem

ALGORITHM *BruteForceClosestPair*(P)

```
//Finds distance between two closest points in the plane by brute force
//Input: A list P of n (n ≥ 2) points p1(x1, y1), . . . , pn(xn, yn)
//Output: The distance between the closest pair of points
d ← ∞
for i ← 1 to n − 1 do
    for j ← i + 1 to n do
        d ← min(d, sqrt((xi − xj)^2 + (yi − yj)^2)) //sqrt is square root
return d
```

43.Prove that any comparison sort algorithm requires Ω (n log n) comparisons in the worst case.

The author states that any comparison sort algorithm requires Ω(nlgn) comparisons in the worst case. Taking the bubble sort algorithm as an example, in the worst case we have an upper bound O(n^2). Omega represents the lower or least bound therefore wouldn't the lower bound of a worst case be Ω(n^2) as well? How would a bubble sort have a lower bound, such as the suggested Ω(nlgn), rather than n^2 in a worst case performance? In the worst case performance bubble sort can't take AT LEAST nlgn.

44.Give the mathematical notation to determine if a convex direction is towards left or right and write the algorithm

It is not difficult to prove the geometrically obvious fact that the leftmost point p1 and the rightmost point pn are two distinct extreme points of the set's convex hull (Figure 5.8).

Let p1 pn be the straight line through point's p1 and pn directed from p1to pn.

This line separates the points of S into two sets:

S1 is the set of points to the left of this line, and S2 is the set of points to the right of this line. We say that point q3 is to the left of the line q1q2 directed from point q1 to point q2 if q1 q2 q3 forms a counterclockwise cycle.

Later, we cite an analytical way to check this condition, based on checking the sign of a determinant formed by the coordinates of the three points.

The points of S on the line p1 pn, other than p1 and pn, cannot be extreme points of the convex hull and hence are excluded from further consideration.

T (n) = 2T (n/2) + f (n), where f (n) ∈ Θ(n).

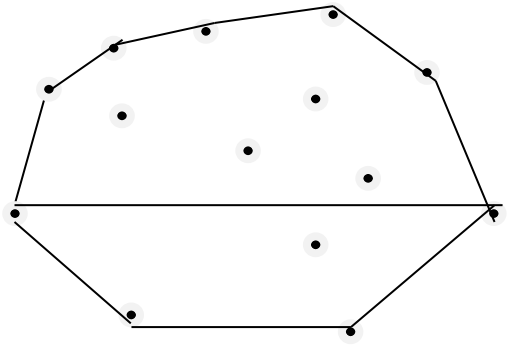


Fig Upper and lower hulls of a set of points

45. Devise an algorithm to make for 1655 using the Greedy strategy. The coins available are {1000,500,100,50,20,10,5}

Algorithm:

while(there are more coins and the instance is not solved)

{
grab the largest remaining coin; // selection procedure if(adding the coin makes the change exceed the amount owed)
reject the coin; // feasibility check
else
add the coin to the change;
if(the total value of the change equals the amount owed) //
solution check the instance is solved;
}
Solution for the given instance 1655 = 1000 + 500 +100 + 50 + 5.

46. Write the advantage of insertion sort ?

The main advantage of the insertion sort is its simplicity.
It also exhibits a good performance when dealing with a small list.
The insertion sort is an in-place sorting algorithm so the space requirement is minimal.

47. Write the disadvantage of insertion sort ?

The disadvantage of the insertion sort is that it does not perform as well as other, better sorting algorithms.
With n-squared steps required for every n element to be sorted, the insertion sort does not deal well with a huge list.
Therefore, the insertion sort is particularly useful only when sorting a list of few items.

48. What is an exhaustive search ?April/May 2018

Exhaustive search is simply a brute-force approach to combinatorial problems.
It suggests generating each and every element of the problem domain, selecting those of them that satisfy all the constraints, and then finding a desired element.

49. State Master's theorem. April/May 2018

If $f(n) \in \Theta(n^d)$ with $d \geq 0$ in recurrence equation then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

50. What are the differences between dynamic programming and divide and conquer approaches? Nov/Dec 2018

Both techniques split their input into parts, find sub solutions to the parts, and synthesize larger solutions from smaller ones.

Divide and conquer splits input at pre-specified deterministic points(eg., always in the middle)

Dynamic programming splits its every possible split rather than at pre-specified points. After trying all split points, it determines which split point is optimal.

51.Give an example for Hamiltonian circuit. Nov/Dec 2018

- complete graph with more than two vertices is Hamiltonian

- cycle graph is Hamiltonian
- tournament has an odd number of Hamiltonian paths (Rédei 1934)
- platonic solid, considered as a graph

52.Write brute force algorithm to string matching.

```
ALGORITHM BruteForceStringMatch (T[0…n-1], P[0…m-1])
//Implements brute-force string matching
//Input: An array T[0…n-1] of n characters representing
// a text and an array P[0…m-1] of m characters
// representing a pattern
//Output: The index of the first character in the text that
// starts a matching substring or -1 if the search is
// unsuccessful
for i ← 0 to n – m do
j ← 0
while j < m and P[j] = T[i + j] do
j ← j + 1
if j = m return i
return -1
```

53.What is time and space complexity of merge sort.

Space complexity of this Merge Sort here is O(n). However, if I choose to perform in-place merge sort using linked lists (not sure if it can be done with arrays reasonably) will the space complexity become O(lg(n))

Time complexity of merge sort

case	**ge case**	**case**
g2 n)	g2 n)	g2 n)

54. Write an example problem that cannot be solved by brute-force algorithm. Justify your answer. April/May 2021

It is often implemented by computers, but it cannot be used to solve complex problems such as the **travelling salesman problem** or the game of chess, because the number of alternatives is too large for any computer to handle.

PART – B

1. Discuss in detail about Brute force algorithm.

Brute force

Brute Force is a straightforward approach to solve a problem, which is directly based on the problem statement and definition of the concepts. Brute Force strategy is one of the easiest approaches.

For example**Computing an : for a given number a and a non negative integer n , find the exponentiation as follows**

an = a * a* a* ….a* for n times

Computing n! : The n! can be computed as 1 *2 * 3….*n

Performing multiplication of two matrices. Searching a key value from given list of elements.

Brute Force Algorithm

Two types of Brute force algorithm

1.Consecutive integer checking algorithm for computing the greatest common divisor of two integer[gcd (m,n)]

2. Definition based algorithm for matrix multiplication.

Advantages of Brute Force Approach

1. It is applicable to a variety of problems.
2. A brute Force algorithm cab be useful for solving small size instances of a problem.
3. A brute Force algorithm cab be serve an important theoretical or educational purpose.
4. The brute force approach provides reasonable algorithms of atleast some practical value with no limitation on instance size for sorting, searching, matrix multiplication and string matching problem.

2. Explain briefly about Brute-Force String Matching.

Brute-Force String Matching

Given a string of n characters called the text and a string of m characters ($m \leq n$) called the pattern, find a substring of the text that matches the pattern.

Find i, the index of the leftmost character of the first matching substring in the text such that t i = p 0,..., t i+j = p j,...,t i+m-1 = p m-1 :

t 0... t i... t i+j... t i+m-1... t n-1 text T

↕ ↕ ↕

p 0... p j... p m-1 pattern P

Align the pattern against the first m characters of the text and start matching the corresponding pairs of characters from left to right until all m pairs of the characters match or a mismatching pair is encountered.

If a mismatch pair is encountered, then shift the pattern one position to the right and resume character comparisons, starting again with the first character of the pattern and its counterpart in the text.

Note that the last position in the text which can still be a beginning of a matching substring is n – m (provided text's positions are indexed from 0 to n – 1).

ALGORITHM:

```
ALGORITHM BruteForceStringMatch (T[0…n-1], P[0…m-1])
//Implements brute-force string matching
 //Input: An array T[0…n-1] of n characters representing
 // a text and an array P[0…m-1] of m characters
 // representing a pattern
//Output: The index of the first character in the text that
// starts a matching substring or -1 if the search is
// unsuccessful
 for i ← 0 to n – m do
 j ← 0
while j < m and P[j] = T[i + j] do
j ← j + 1
if j = m return i
 return -1
```

Brute-Force String Matching

Example of Brute-Force string matching.

```
N O B O D Y _ N O T I C E D _ H I M
N O T
  N O T
     N O T
          N O T
            N O T
               N O T
                  N O T
                     N O T
```

Analysis of Brute-Force String Matching Algorithm

Input Size metric: Number of characters in the text i.e., n.

Basic operation: Key comparison

The number of times the basic operation is executed depends not only on the array size but also on pattern of input. n-m m-1 n-m n-m C worst (n) = ∑ ∑ 1 = ∑ [(m - 1) - 0 + 1] = ∑ m.

i=0 j=0 i=0 i=0 n-m = m ∑ 1 = m ((n - m) – 0 + 1) = mn – m 2 + m i=0 ≈ mn Є Θ(mn) C best (n) =Θ(m) C average (n)=Θ(n).

Consider the problem of counting, in a given text the number of substrings that start with an A and end with a B. For example, there are four such substrings in CABAAXBYA. Design a brute-force algorithm for this problem and determine its efficiency class.(April/May 2021)

Here is one way in which iterating backwards through the string could result in O(n) computation instead of your original O(n^2) work:

```
A = "CABAAXBYA"

count = 0 # Number of B's seen
total = 0
for a in reversed(A):
    if a=='B':
        count += 1
    elif a=='A':
        total += count

print total
```

This works by keeping track in count of the number of B's to the right of the current point.

(Of course, you could also get the same result with forwards iteration by counting the number of A's to the left instead:

```
count = 0 # Number of A's seen
total = 0
for a in A:
    if a=='A':
        count += 1
    elif a=='B':
        total += count
print total
)
```

3. Explain the brute force method to find the two closest points in a set of n points in K-Dimensional space *Nov/Dec 2017* or Explain convex-Hull problems in detail. OR What is convex hull problem?explain the brute force approach to solve convex hull with an example.derive the time complexity April/May 2019

The closest pair problem is – finding the two closest points from the set of n points.

For simplicity the closet pair problem can be considered to be in two dimensional case.

The point is specified by a pair (x,y).hence .hence P=(x,y) is a point on a two dimensional plan.

The distance between two points is denoted by Euclidean distance.

It is denoted as

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}.$$

where pi and pj are two points for which i<j

ALGORITHM *BruteForceClosestPair*(P)

//Finds distance between two closest points in the plane by brute force
//Input: A list P of n ($n \geq 2$) points $p_1(x_1, y_1), \ldots, p_n(x_n, y_n)$
//Output: The distance between the closest pair of points
$d \leftarrow \infty$
for $i \leftarrow 1$ **to** $n - 1$ **do**
  for $j \leftarrow i + 1$ **to** n **do**
    $d \leftarrow \min(d, sqrt((x_i - x_j)^2 + (y_i - y_j)^2))$ //*sqrt* is square root
return d

The basic operation in above algorithm is computing Euclidian distance between two points.Then the basic operation of the algorithm will be squaring a number.
The number of times it will be executed can be computed as follows:

$$C(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 2$$

$$= 2\sum_{i=1}^{n-1} (n - i)$$

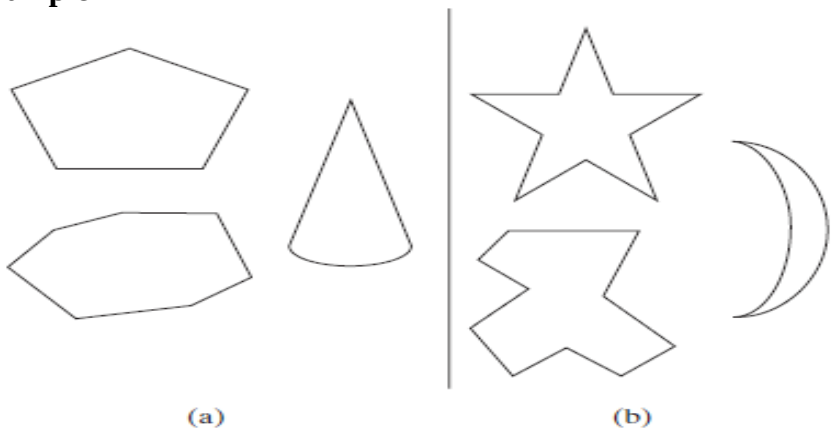
$$= 2[(n - 1) + (n - 2) + \ldots + 1] \quad = (n - 1)n \in \theta(n2).$$

Of course, speeding up the innermost loop of the algorithm could only decrease the algorithm's running time by a constant factor, but it cannot improve its asymptotic efficiency class.

Convex-Hull Problem

Definition: Given a set S { p1, p2 ,p3, …pn} of points in the plan , the convex hull H(S) is the smallest convex polygon in the plane that contains all of the points of S. the set S is called as coves set .A polygon is convex if and only if any two points from the set forming a line segment with end points entirely within the polygon

For example



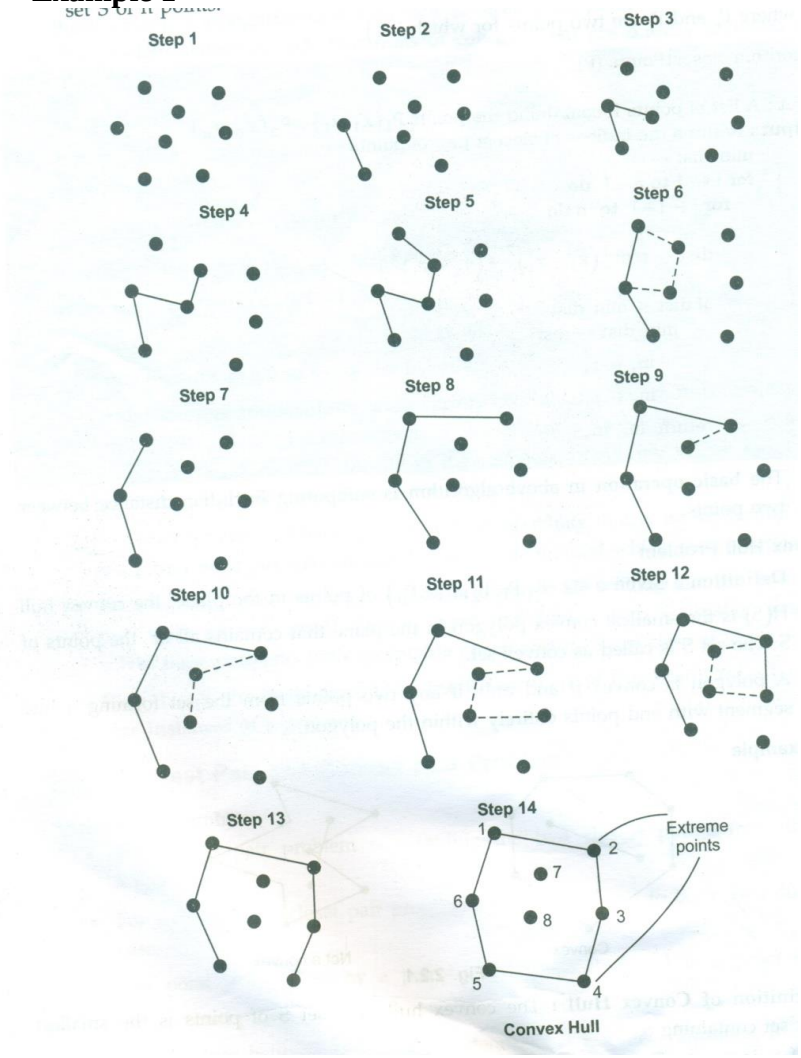
(a) Convex sets. (b) Sets that are not convex.

Definition of Convex Hull:

The convex hull of a set S of points is the smallest convex set containing S.
If S is a set of two points its convex hull is the line segment connecting these points .if S is a set of three points then its convex hull is the triangle

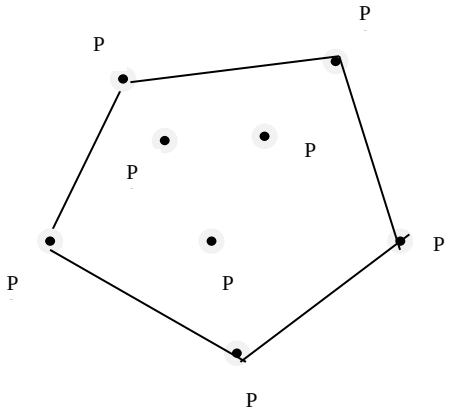
Convex hull problem is the problem of constructing the convex hull for a given set S of n points

Example 1



The points { 1, 2, 3, 4 , 5, 6 } are called extreme points

Example 2:



The convex hull for this set of eight points is the convex polygon with vertices at {p1, p5, p6, p7, and p3.}

Definition of Extreme points

- ✓ An extreme point of a convex set is a point which is not a middle point of any line segment with end points in the set
- ✓ Extreme points have several special properties other points of a convex set do not have. One of them is exploited by the simplex method.
- ✓ This algorithm solves linear programming problems, which are problems of finding a minimum or a maximum of a linear function of n variables subject to linear constraints.
- ✓ Here, however, we are interested in extreme points because their identification solves the convex-hull problem.
- ✓ Actually, to solve this problem completely, we need to know a bit more than just which of n points of a given sets are extreme points of the set's convex hull: we need to know which pairs of points need to be connected to form the boundary of the convex hull.
- ✓ Note that this issue can also be addressed by listing the extreme points in a clockwise or a counter clockwise order.

A few elementary facts from analytical geometry are needed to implement this algorithm.

1. The straight line through two points (x1, y1), (x2, y2) in the coordinate plane can be defined by the equation**ax + by = c** where a = y2 − y1, b = x1 − x2, c = x1y2 − y1x2.

2. Such a line divides the plane into two half-planes: for all the points in the other, ax + by < c.

For the points on the line itself, of course, ax + by = c.

Thus, to check whether certain points lie on the same side of the line, we can simply check whether the expression ax + by = c has the same sign for each of these points.

Time efficiency of this algorithm is in O(n3):

for each of n(n − 1)/2 pairs of distinct points, we may need to find the sign of ax + by – c for each of the other n − 2 points

4. Explain the concept of Exhaustive search with the help of an example. Or state the travelling salesman problem.elaborate the steps in solving the travelling salesman problem using brute force approach. Nov/Dec 2019

Explain how exhaustive search can be applied to the sorting problem and determine the efficiency class of such an algorithm. (April/May 2021)

Exhaustive search is simply a brute-force approach to combinatorial problems.

It suggests generating each and every element of the problem domain, selecting those of them that satisfy all the constraints, and then finding a desired element.

Three exhaustive search problems:

- The travelling salesman problem,
- The knapsack problem, and
- The assignment problem.

1.Travelling Salesman Problem

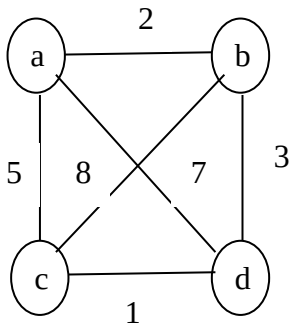
- ✓ The Travelling salesman problem (TSP) has been intriguing researchers for the last 150 years by its seemingly simple formulation, important applications, and interesting connections to other combinatorial problems.
- ✓ The travelling salesman problem (TSP) is a famous problem in the graph theory.It can be stated as follows - consider that there are n cities and **travelling salesman has to visit each city exactly once and has to return to the city from where he has started.**
- ✓ To model this problem weighted graph can be used. The vertices of such graph represent cities and the edge weight specifies the distances between the cities
- ✓ This problem can also be started **as finding shortest Hamiltonian circuit of the graph.** The shortest Hamiltonian circuit is a cycle in the given graph such that all the vertices of the

graph can be visited only once.And the tour obtained in such a way has shortest distance.

Example

Consider the graph as given below.This is a weighted graph in which weight along the edges represent the distances among the cities.

We have to find shortest Hamiltonian circuit i.e. the path in which each city is visited once and returning to the city from which it has started initially.



Tour	Length	
a -> b -> c -> d -> a	I = 2+8+1+7 = 18	
a -> b -> d -> c-> a	I = 2+3+1+5 = 11	optimal
a-> c -> b -> d -> a	I = 5+8+3+7 = 23	
a-> c -> d -> b -> a	I = 5+1+3+2 = 11	optimal
a-> d ->b -> c -> a	I = 7+3+8+5 = 23	
a-> d -> c -> b -> a	I = 7+1+8+2 = 18	

Fig: Solution to a small instance of the travelling salesman problem by exhaustive search

- ✓ Thus we have to try each possible path and find the shortest distance which gives optimal tour.
- ✓ It is easy to see that a Hamiltonian circuit can also be defined as a sequence of n + 1 adjacent vertices $v_{i0}, v_{i1}, \ldots, v_{in-1}, v_{i0}$, where the first vertex of the sequence is the same as the last one and all the other n − 1 vertices are distinct

2. Knapsack Problem

- ✓ This is another popular problem which can be solved using exhaustive search.
  It can be stated as follow : suppose that there are n objects from I = 1,2 , 3, …n. each object I as some weight w_i and values associated with each object is v_i .
  And capacity of knapsack is W. a person has to pickup the most valuable objects to fill the knapsack to its capacity.

Example: Consider a knapsack instance as follows The Knapsack capacity W=8

I	W_i	V_i
1	7	$42
2	3	$12
3	4	$40
4	5	$25

Subset	**Total Weight**	**Total Value**
Nil	0	$0
{1}	7	$42
{2}	3	$12

{3}	4	$40
{4}	5	$25
{1,2}	7+3=10	$42+$12=$54
{1,3}	7+4=11(11>10)	Not feasible
{1,4}	7+5=12(12>10)	Not feasible
{2,3}	3+4=7	$12+$40=$52
{2,4}	3+5=8	$12+$25=$37
{3,4}	4+5=9	$40+$25=**$65(Feasible)**
{1,2,3}	7+3+4=14 (14>10)`	Not feasible
{1,2,4}	7+3+5=15 (15>10)	Not feasible
{1,3,4}	7+4+5=16 (16>10)	Not feasible
{2,3,4}	3+4+5=12 (12>10)	Not feasible
{1,2,3,4}	7+3+4+5=19 (19>10)	Not feasible

Since the subset {3, 4} gives the maximum value $65, it is the feasible solution, so item 3 and item 4 can be put in the Knapsack bag.

Because in this method, each element of the problem's domain has to be searched for obtaining solution. Hence these problems are also called as NP-hard problems

3.Assignment Problem

- ✓ Consider that there are n people who need to be assigned to execute n jobs i.e only one person is assigned to execute one job at a time.
  Then problem is to find such assignment that gives smallest total cost.
  The cost can be computed as cost C[i, j, k, l]
  C[i, j, k, l] = Job i is assigned to Person 1, Job j is assigned to Person 2, Job k is assigned to Person 3, Job 4 is assigned to Person 4.
  Example 1

Person	**Job 1**	**Job 2**	**Job 3**	**Job 4**
Person 1	9	2	7	8

Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

<1, 2, 3, 4> Cost = 9 + 4 + 1 + 4 = 18

<1, 2, 4, 3> Cost = 9 + 4 + 8 + 9 = 30

<1, 3, 2, 4> Cost = 9 + 3 + 8 + 4 = 24

<1, 3, 4, 2> Cost = 9 + 3 + 8 + 6 = 26

<1, 4, 2, 3> Cost = 9 + 7 + 8 + 9 = 33

<1, 4, 3, 2> Cost = 9 + 7 + 1 + 6 = 23

... ... etc

The no. of permutation for assignment problem is n!.

Example 2

Person	**Job 1**	**Job 2**	**Job 3**	**Job 4**
Person 1	10	3	8	9
Person 2	7	5	4	8
Person 3	6	9	2	9
Person 4	8	7	10	5

The cost can be obtained by assigning the jobs in various combinations as

< 1, 2, 3, 4 > Cost = 10 + 5 + 2 + 5 = 22

< 1, 2, 4, 3 > Cost = 10 + 5 + 9 + 10 = 34

< 1, 3, 4, 2 > Cost = 10 + 4 + 9 + 7 = 30

< 1, 3, 2, 4 > Cost = 10 + 4 + 9 + 5 = 28

< 1, 2, 4, 3 > Cost = 10 + 5 + 9 + 10 = 34

< 1, 4, 2, 3 > Cost = 10 + 8 + 9 + 10 = 37

< 1, 4, 3, 2 > Cost = 10 + 8 + 2 + 7 = 27

.... etc.

Thus by trying 24 permutations (n! = 4! = 24), we can obtain feasible solution.

The feasible solution < 2, 1, 3, 4 > i.e . Cost = 3 + 7 + 2 + 5 = 17

Thus we have to generate n! instances to find solution using exhaustive search method is for solving such problems.

For solving these type of problems , many efficient algorithms are available .

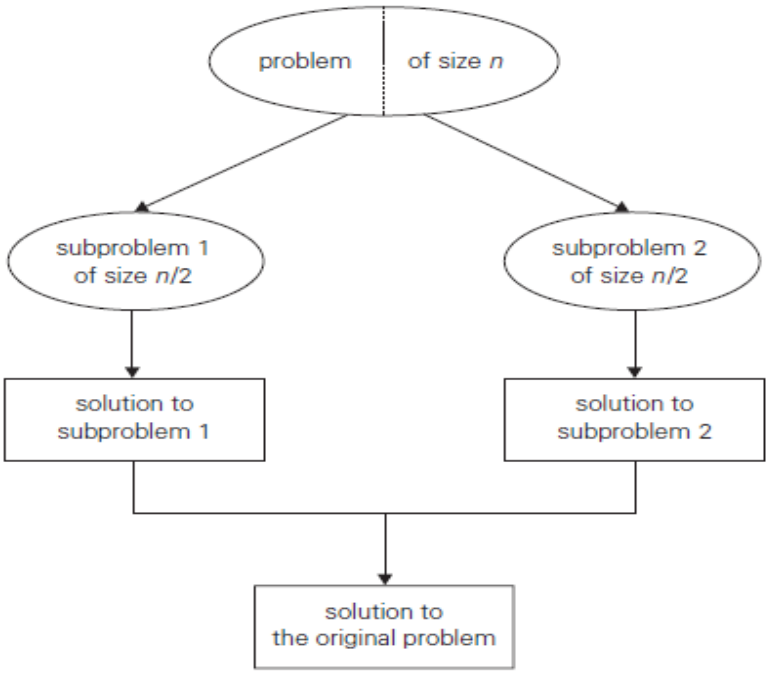
5. Explain Divide and Conquer technique. *Dec 2009*

- ✓ Divide & conquer technique is a top-down approach to solve a problem.
- ✓ The algorithm which follows divide and conquer technique involves 3 steps:
  Divide the original problem into a set of sub problems.
  Conquer (or Solve) every sub-problem individually, recursive.
  Combine the solutions of these sub problems to get the solution of original problem.
  Divide and Conquer is one of the best algorithm design technique

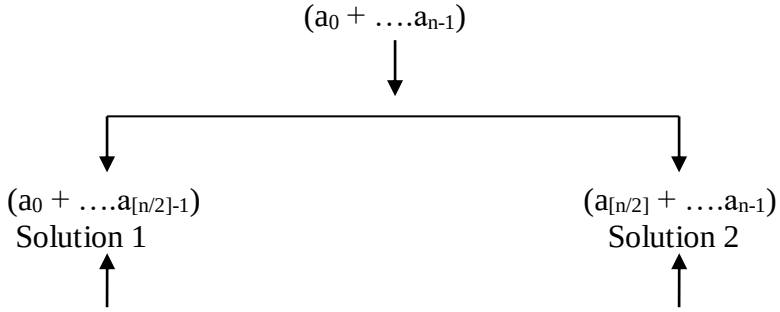
Algorithm DC(p)

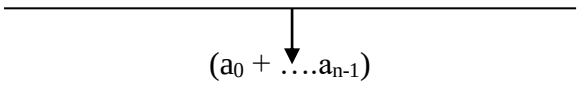
```
{
        If P is too small then
        Return solution of P.
Else
        {
Divide (p) and obtain  p1, p2, …..pn  where n ≥ 1
        Apply DC to each sub problem
Return combine (DC(p 1),
DC( p2)….Dc(pn));
        }
}
```

The diagrammatic representation of the divide and conquer technique is shown in figure which divides the problem into two smaller sub problems.



Example:To compute sum of n numbers then by divide and conquer we can solve the problem as





If we want to divide a problem of size n in to a size of n /b taking f(n) time to divide and combine , then we can set up recurrence relation for obtaining time for size n is

T (n) = a T (n/b) + f (n),

T(n/b) = Time for size n/b time required for dividing the problem in to sub problem.

T(n) = Time for size n

n = number of sub instances

The above equation is called general divide and conquer recurrence. The order of growth of T(n) depends upon the constants a, b and order of growth function f(n).

Divide and Conquer technique

Examples for divide and conquer method are,

- **Binary search**
- **Quick sort**
- **Merge sort**

Example 1 :

Consider the problem of computing the sum of number a_0 …… a_{n-1}.If n > 1, the problem is divided into two instances of the same problem.

They are

To compute the sum of the first [n/2] numbers.

To compute the sum of the remaining [n/2] numbers.

Once the two instances are computed, add their values to get the sum of original problem.

$\mathbf{a_0 + a_1 + \ldots\ldots + a_{n-1} = (a_0 + a_1 + \ldots\ldots + a_{[n/2]-1}) + (a_{[n/2]} + \ldots\ldots + a_{n-1})}$

An instance of size n can be divided into several instances of size n/b,

Where a and b are constants $a \geq 1$ and $b > 1$

The recurrence for the running time T(n) is

$$\mathbf{T(n) = aT(n/b) + f(n)\ ,}$$

which is called as general divide and conquer recurrence

- ✓ where, f(n) is a function that accounts for the time spent on dividing the problem into smaller ones and on combining their solutions.
- ✓ The order of growth of T(n) depends on the values of the constants ‘a’ and ‘b’ and the order of growth of the function f(n).

For example, the recurrence equation for the number of additions is

$a(n) = 2a(n/2) + 1$

Advantages of divide and conquer

The time spent on executing the problem using divide and conquer is smaller than other methods.

The divide and conquer approach provides an efficient algorithm in computer science.

The divide and conquer technique is ideally suited for parallel computation in which each sum problem can be solved simultaneously by its own processor.

6. Explain the Merge Sort algorithm with the help of illustrative Example. *Dec2013/14/15/16*

OR Explain the working of Merge Sort Algorithm with an example. *Nov/Dec 2017*

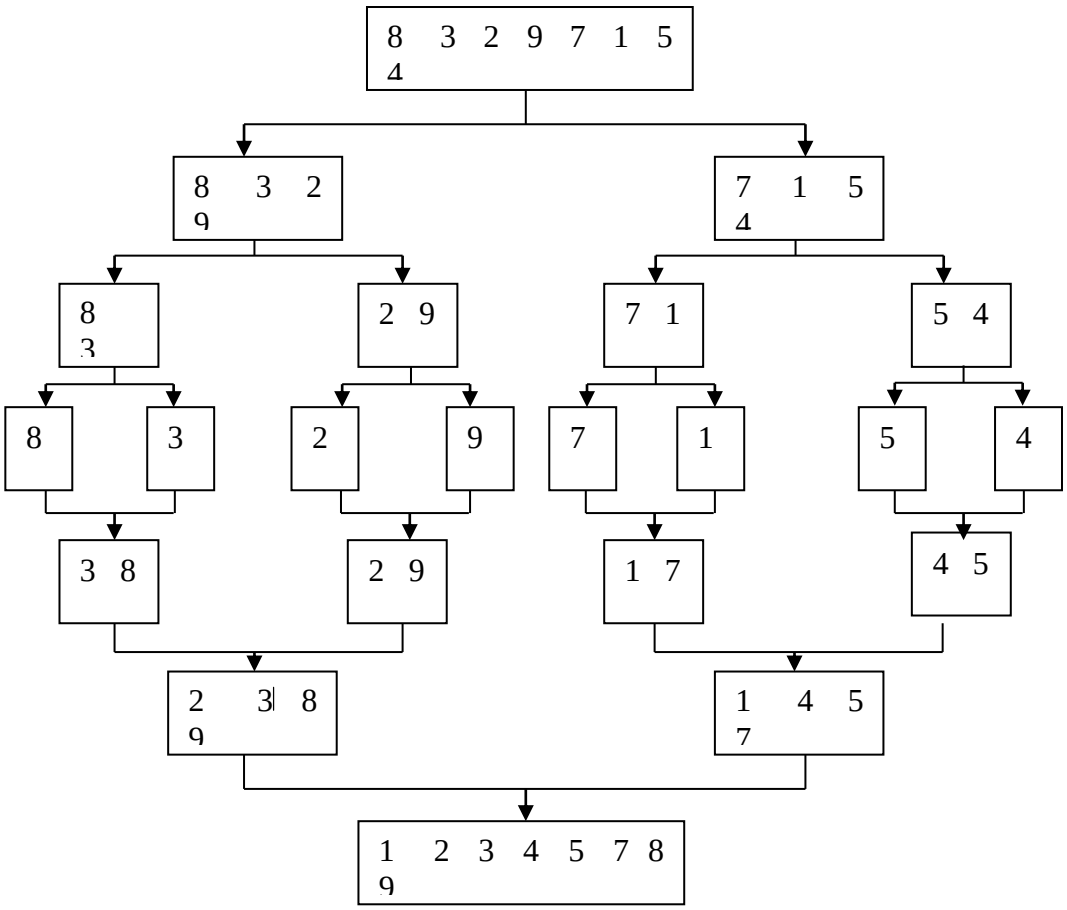
Explain Merge sort algorithm with an example. April/May 2018

- ✓ The merge sort is a sorting algorithm that uses the divide and conquer strategy.
  Division is dynamically carried out.
- ✓ Merging is the process of combining two or more files into a new sorted file.
  Merge sort on an input array with n elements consists of three steps:
  Divide: partition array into two sub lists s1 and s2 with n/2 elements each Conquer: then sort sub list s1 and sub list s2.
  Combine: merge s1 and s2 into a unique sorted group.
  Merge sort is a perfect example of a successful application of the divide and conquer technique.
  It sorts a given array A[0…..n − 1] by dividing it into two halves A[0…..[n/2]−1] and A[[n/2]…..n − 1].
  It sorts each half separately by using recursive procedure, and Then, merging the two smaller sorted arrays into a single sorted one.
  Steps to be followed
- ✓ The first step of the merge sort is to **chop** the list into two.
- ✓ If the list has **even length**, split the list into two equal sub lists.
- ✓ If the list has **odd length**, divide the list in two by making the first sub list one entry greater than the second sub list.
- ✓ Then split both the sub lists into two and go on until each of the **sub lists are of size one**.

Finally, start merging the individual sub lists to obtain a sorted list.

Example:

The operation of the algorithm for the array of element (8,3,2,9,7,1,5,4) is explained in the figure given below



An example of Merge Sort operation

ALGORITHM

Algorithm Mergesort(A[0..n − 1])
//Sorts array A[0..n − 1] by recursive mergesort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
If n > 1
copy A[0..n/2] − 1] to B[0..n/2] − 1]
copy A[[n/2]..n − 1] to C[0..[n/2]] − 1]
Merge sort(B[0..[n/2] − 1])
Mergesort(C[0..[n/2] − 1])
Merge (B, C, A) //see below

- ✓ The merging of two sorted arrays can be performed as follows.
- ✓ Two pointers are initialized to point to the first elements of the arrays being merged.
- ✓ Then the elements are compared and the smaller of both is added to a new array or list being constructed.
- ✓ Then the index of that smaller element is incremented to point to its immediate successor in the array.

The above steps are continued until one of the two given array is exhausted.
Then the remaining elements of the other array are copied to the end of the next array.

Algorithm Descriptive and Implementation

ALGORITHM Merge(B[0..p − 1], C[0..q − 1], A[0..p + q − 1])
//Merges two sorted arrays into one sorted array
//Input: Arrays B[0..p − 1] and C[0..q − 1] both sorted
//Output: Sorted array A[0..p + q − 1] of the elements of B //and C
i ← 0; j ← 0; k ← 0
while i<p and j<q do
if B[i] ≤ C[j]
A[k]← B[i];
i ← i + 1
else
A[k]← C[j];
j ← j + 1
k ← k + 1
if i = p
copy C[j..q − 1] to A[k..p + q − 1]
else
copy B[i..p − 1] to A[k..p + q − 1]

Efficiency of Merge Sort

In merge sort algorithm the two recursive calls are made. Each recursive call focuses on n/2 elements of the list .
After two recursive calls one call is made to combine two sublist i.e to merge all n elements.
Hence we can write recurrence relation as

T(n) = T(n/2) + T(n/2) + cn

T(n/2) = Time taken by left sublist
T(n/2) = time taken by right sublist

T(n) = time taken for combining two sublists
where n> 1 T (1) = 0

The time complexity of merge sort can be calculated using two methods
Master theorem

Substitution method

Master theorem

Let , the recurrence relation for merge sort is

T(n) = T(n/2) + T(n/2) + cn

let

T(n) = aT(n/b) + f(n) be a recurrence relation

i.e. T(n) = 2T(n/2) + cn ------- (1)

T(1) = 0 ---------- (2)

As per master theorem

$T(n) = \Theta (n^d \log n)$ if a = b

As equation (1),

a =2 , b = 2 and f(n) = cn and $a = b^d$

i.e $2 = 2^`$

This case gives us , $T(n) = \Theta (n \log 2 n)$

Hence the average and worst case time complexity of merge sort is $\mathbf{C_{worst}(n) = (n \log 2 n)}$

Substitution method

Let, the recurrence relation for merge sort be

T(n) = T(n/2) + T(n/2) + cn for n>1

i.e. T(n) = 2T(n/2) + cn for n>1 ------- (3)

T(1) = 0 -------(4)

Let us apply substitution on equation (3) .

Assume $n=2^k$

T(n) = 2T(n/2) + cn

$T(n) = 2T(2^k/2) + c.2^k$

$T(2^k) = 2T(2^k-1) + c.2^k$

If k = k-1 then,

$T(2^k) = 2T(2^{k-1}) + c.2^k$

$T(2^k) = 2[2T(2^{k-2}) + c.2^k -1] + c.2^k$

$T(2^k) = 2^2 T(2^{k-2}) + 2.c.2^k -1 + c .2^k$

$T(2^k) = 2^2 T(2^{k-2}) + 2.c.2^k /2 + c.2^k$

$T(2^k) = 2^2 T(2^{k-2}) + c.2^k + c.2^k$

$T(2^k) = 2^2 T(2^{k-2}) + 2c .2^k$

Similarly we can write,

$T(2^k) = 2^3 T(2^{k-3}) + 3c .2^k$

$T(2^k) = 2^4 T(2^{k-4}) + 4c .2^k$

…….

$T(2^k) = 2^k T(2^{k-k}) + k.c.2^k$

$T(2^k) = 2^k T(2^0) + k.c.2^k$

$T(2^k) = 2^k T(1) + k.c.2^k$ -------- (5)

But as per equation (4), T(1) =0

There equation (5) becomes ,

$T(2^k) = 2^k .0 +. k. c . 2^k$

$T(2^k) = k. c . 2^k$

But we assumed $n=2^k$, taking logarithm on both sides.

i.e. log 2 n = k

Therefore T(n) = log 2 n. cn

Therefore T (n) =Θ (n log2 n)

Hence the average and worst case time complexity of merge sort is

C_{worst} (n) = (n log2 n)

Time complexity of merge sort

case	**ge case**	**case**
)g2 n)	)g2 n)	)g2 n)

Application of Merge Sort

- Sorting
- Tape Sorting
- Data Processing

Demerit

The algorithm requires linear amount of extra storage.

7. Explain the Quick Sort algorithm with the help of illustrative example Or Explain the time complexity of quick sort method in detail OR Write the algorithm for Quick Sort and write its time complexity with example list are 5, 3, 1, 9, 8, 2, 4, 7. Apr/May 2017

Write the algorithm for quick sort. Provide a complete analysis of quick sort for the given set of numbers 12, 33, 23, 43, 44, 55, 64, 77 and 76. (13) Nov/Dec 2018

Or Write the quick sort algorithm and explain it wiyh example.derive the worst case and average case tiome complexity April/May 2019

✓ Quick sort is a sorting algorithm that uses the divide and conquers strategy.
The three steps of quick sort are as follows:
Divide: Split the array into two sub arrays that each element in the left sub array is less than or equal the middle element and each element in the right sub array is greater than the middle element .
The splitting of the array into two sub array is based on pivot element. All the elements that are less than pivot should be in left sub array and all the elements that are more than pivot should be in right sub array

Conquer: Recursively sort the two sub arrays.

Combine: Combine all the sorted elements in a group to form a list of sorted elements
Quick sort is also referred as Partition Exchange sort.

- ✓ The problem of sorting a set is reduced to the problem of sorting two smaller subsets.
  Quick sort divides input elements according to their position in the array.It also divides the input elements according to the value of element.
  To achieve the partition, quick sort rearrange the given array element a[0,..n-1]
  It is a situation where all the elements before the position 'S' are smaller than or equal to a[s] and all the elements after position 's' are greater than or equal to a[s].

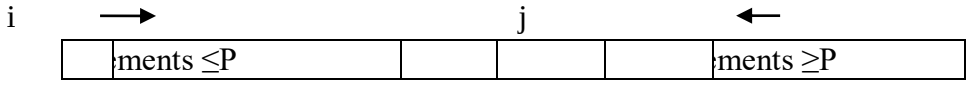
  The partition is shown as

These elements are Less than A[m]

These elements are Greater than A[m]

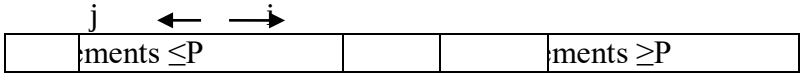
After partitioning, a[s] will be in its final position in the sorted array.
Then sorting of element of two sum arrays preceding and following a[s] can be done independently.
After both scans stop, three situations may arise, depending on whether or not the scanning indices have crossed.

1. If scanning indices i and j have not crossed, **i.e., i < j,** we simply xchange A[i] and A[j] and resume the scans by incrementing i and decrementing j, respectively:



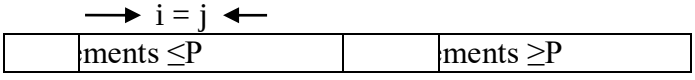
P- pivot element

2. If the scanning indices have crossed over, **i.e., i > j,** we will have partitioned the Sub array after exchanging the pivot with A[j].



3. Finally, if the scanning indices stop while pointing to the same element, **i.e.,i = j,** the value they are pointing to must be equal to p.

Thus, we have the sub array partitioned, with the split position s = i = j :



Combine the last case with the case of crossed-over indices (i > j) by exchanging the pivot with A[j] whenever i ≥ j .

Example

The array elements are

5 3 1 9 8 2 4 7

The first element of array is chosen as pivot element.

Two indices i and j are used for scanning.

P i j
5 3 1 9 8 2 4 7

P i j
5 3 1 9 8 2 4 7

P i j
5 3 1 9 8 2 4 7

Now exchange the elements 9 and 4 now array becomes,

P i j
5 3 1 4 8 2 9 7

Now also exchange a[i] and a[j], the resultant array becomes,

P i j
5 3 1 4 2 8 9 7

Now the scanning indices i and j have not crossed (ie) i < j, simply exchange i and j.

The array becomes

P j i
5 3 1 4 2 8 9 7

Since a[j] < pivot, (2<5) exchange them.

The result is

P
2 3 1 4 **5** 8 9 7

Now, the array has been sub divided into sub array with pivot element as middle.

Sub array 1

2 3 1 4

P i j
2 3 1 4

P i j
2 3 1 4

Exchange a[i] and [j]

P i j
2 1 3 4

Since i < j, exchange a and j

P j i
2 1 3 4

Since a[j] < pivot, (1 < 2) exchange i and j

P i j
1 **2** 3 4

Sub array 3
3 4

P ij
3 4

i=j, ie both points to the same element.
j j
3 4

Sub array 2
8 9 7

P i j
8 9 7

Exchange a[i] and a[j]

The sub array 2 becomes

P i j
8 7 9

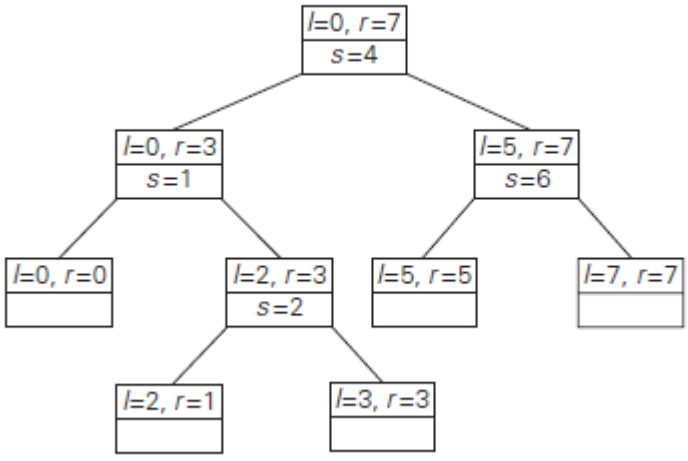
Here i < j simply exchange i and j

It becomes
P j i
8 7 9

Since a[j] < pivot, exchange them
7 **8** 9

Hence, the array elements are sorted. The sorted array is
1 2 3 4 5 7 8 9

Recursive Calls Tree



Tree of recursive calls to Quicksort with input values l and r of subarray bounds and split position s of a partition obtained.

ALGORITHM FOR QUICK SORT

```
ALGORITHM Quicksort(A[l..r])
//Sorts a subarray by quicksort
//Input: Subarray of array A[0..n − 1], defined by its left and
//right  indices l and r
//Output: Subarray A[l..r] sorted in nondecreasing order
if l<r
s ←Partition(A[l..r]) //s is a split position
Quicksort(A[l..s − 1])
Quicksort(A[s + 1..r])

ALGORITHM Hoare Partition(A[l..r])
//Partitions a subarray by Hoare's algorithm, using the first //element as a pivot
//Input: Subarray of array A[0..n − 1], defined by its left and //right  indices l and r (l < r)
//Output: Partition of A[l..r], with the split position returned //as this function's value
p ← A[l]
i ← l; j ← r + 1
repeat
repeat i ← i + 1 until A[i] ≥ p
repeat j ← j − 1 until A[j ] ≤ p
swap(A[i], A[j ])
until i ≥ j
swap(A[i], A[j ]) //undo last swap when i ≥ j
swap(A[l], A[j ])
return j
```

Efficiency of Quick Sort

The number of key comparisons made before a partition is achieved is n + 1 if the scanning indices i and j cross over.

The number of key comparisons is n, if the scanning indices i and j coincides.

Best Case Analysis(split in the middle)

If the array is always partitioned at the mid , then it brings the best case efficiency of an algorithm

The number of key comparisons in the best case satisfies the recurrence

$C_{best}(n) = 2\ C_{best}(n/2) + n$ for $n > 1$,

Or

$C(n) = C(n/2) + C(n/2) + n$ ----------(1)

Time required to sort left sub array — Time required to sort right sub array — Time required for partitioning the sub array

and $C_{best}(1) = 0$.

Using Master Theorem

Solve equation (1) using Master Theorem

If $f(n) \in \Theta(n^d)$ then

$T(n) = \Theta(n^d)$ if $a < b^d$

$T(n) = \Theta(n^d \log n)$ if $a = b^d$

$T(n) = \Theta(n^{\log_b a})$ if $a > b\ b^d$

$C(n) = 2\,C(n/2) + n$

Here $f(n) \in n^1$ therefore $d = 1$

Now , $a = 2$ and $b = 2$

As from case 2 we get $a = b^d$ i.e. $2 = 21$

We get ,

T(n) i.e $C(n) = \Theta(n^d \log n)$

$C_{best}(n) = \Theta(n \log n)$

Best case time complexity of quick sort is **Θ (n log n)**

Using substitution method

$C(n) = C(n/2) + C(n/2) + n$ ----------(1)

$C(n) = 2C(n/2) + n$

Assume $n = 2^K$ since each time the list is divide into two equal halves . then equation becomes,

$C(2^K) = 2C(2^k/2) + 2^k$

$C(2^K) = 2C(2^{k-1}) + 2^k$

Now substitute $C(2^{k-1}) = 2C(2^{k-2}) + 2^{k-1}$

$C(2^K) = 2[2C(2^{k-2}) + 2^{k-1}] + 2^k$

$C(2^K) = 2^2C(2^{k-2}) + 2.2^{k-1} + 2^k$

$C(2^K) = 2^2C(2^{k-2}) + 2^k + 2^k$

$C(2^K) = 2^2C(2^{k-2}) + 2.2^k$

If we substitute $C(2^{k-2})$ then ,

$C(2^K) = 2^2C(2^{k-2}) + 2.2^k$

$C(2^K) = 2^2[2\,C(2^{k-3}) + 2^{k-2}] + 2.2^k$

$C(2^K) = 2^3C(2^{k-3}) + 2^2.2^{k-2} + 2.2^k$

$C(2^K) = 2^3C(2^{k-3}) + 2^k + 2.2^k$

$C(2^K) = 2^3C(2^{k-3}) + 3.2^k$

Similarly we can write

$C(2^K) = 2^4C(2^{k-4}) + 4.2^k$

$C(2^K) = 2^k C(2^{k-k}) + k.2^k$

$C(2^K) = 2^k C(2^0) + k.2^k$

$C(2^K) = 2^k C(1) + k.2^k$

$C(1) = 0$

hence the above equation becomes

$C(2^K) = 2^k.0 + k.2^k$

now as we assumed $n = 2^k$ we can also say

$n = \log_2 n$ [by taking logarithm on both side]

$C(n) = n.0 + \log_2 n \cdot n$

Thus it is proved that best case time complexity of quick sort

is **Θ (n log n)**

Worst Case Analysis (sorted array)

The worst case for quick sort occurs when the pivot is a minimum or maximum of all the elements in the list .

For example,

if A[0..n − 1] is a strictly increasing array and we use A[0] as the pivot,

The left-to-right scan will stop on A[1]

The right-to-left scan continues upto A[0]

The total number of key comparisons made will be equal to

$C_{worst}(n) = (n-1) + n$

$C_{worst}(n) = (n-1) + (n-2) + \ldots + 2 + 1$

But as we know

$1 + 2 + 3 + \text{----} + n = n(n+1)/2 = ½\ n^2$

$C_{worst}(n) \in \theta(n^2)$

The time complexity of worst case of quick sort is **θ (n^2)**

Average Case Analysis (random array)

Let $C_{avg}(n)$ be the average number of key comparison made by Quick Sort.

The partition split can be happen in each position S ($0 \le S \le n-1$) with the probability 1/n.

The recurrence relation is

$$C_{avg}(n) = \frac{1}{n}\sum_{s=0}^{n-1}[(n+1) + C_{avg}(s) + C_{avg}(n-1-s)] \quad \text{for } n > 1,$$

$$C_{avg}(0) = 0, \quad C_{avg}(1) = 0.$$

$C_{avg}(n) \approx 2n \ln n \approx 1.39\ n \log_2 n$.

Thus, on the average case, Quick Sort makes 38% more comparison the best case.

Hence average case time complexity of quick sort is Θ (n log n)

Time complexity of quick sort

Best case	Average case	Worst case
Θ(n log n)	Θ(n log n)	θ (n^2)

Application

Internal sorting of large data sets.

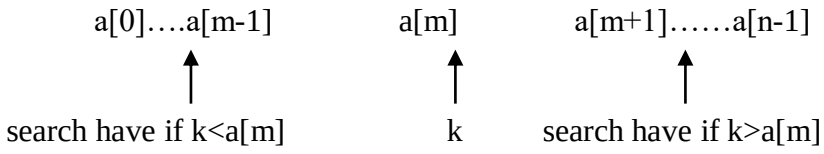
To improve the efficiency of the Quick sort various methods are used to choose the pivot element.

One such method is called, median of three partitioning that uses the pivot element as the median of left most, right most and the middle element of the array.

8. Write an algorithm to perform binary search on a sorted list of elements. Analyse the algorithm for the best case , worst case and average case. *May 2011 / Dec 2008 (*Or) **What is divide and conquer strategy and explain the binary search with suitable example problem.** *Dec 2011* **(**Or **) Differentiate sequential search from binary search technique.***May 2009 Apr/May-2017*

Write an algorithm using divide and conquer to search an element in a list of numbers. If the number is not present, the algorithm returns the closest number of the searches number. (April/May 2021)

- ✓ The binary search algorithm is one of the most efficient searching techniques which require the list to be sorted in ascending order.
- ✓ To search for an element in the list, the binary search algorithms split the list and locate the middle element of the list.
- ✓ The middle of the list is calculated as **middle := (l + r) div n-numberof element** in list
- ✓ The algorithm works by comparing a search key element 'k' with the array middle element a[m] After comparison, any one of the following three conditions occurs.
- ✓ If the search key element 'k' is greater than a[m], then the search element is only in the upper or second half and eliminate the element present in the lower half. Now the value of l is middle m+1.
  - ✓ If the search key element 'k' is less than a[m], then the search element is only in the lower or first half. No need to check in the upper half. Now the value of r is middle m-1.
    If the search key element 'k' is equal to a[m] , then the search key element k is found in the position m, Hence search operation is complete.
  - ✓ The above steps are repeated until the search element is found, which is equal to the middle element or the list consists of only one element that is not equal to the search key element.



Example:

The list of element are 3,14,27,31,39,42,55,70,81,85,93,98 and searching for k=70 in the list.

0	1	2	3	4	5	6	7	8	9	10	11	12
3	14	27	31	39	42	55	70	74	81	85	93	98

m- middle element
m = n div 2
= 13 div 2
m = 6

0	1	2	3	4	5	**6**	7	8	9	10	11	12
3	14	27	31	39	42	**55**	70	74	81	85	93	98

m

0	1	2	3	4	5	**6**	7	8	9	10	11	12

3 14 27 31 39 42 **55** 70 74 81 85 93 98
l **m**

Since k> a[m] , then , l=7.
So, the search element is present in second half.

Now the array becomes
7 8 9 10 11 12
70 74 81 85 93 98
l **r**

m = (l + r) div 2
= 19 div 2
m = 9

7 8 **9** 10 11 12
70 74 **81** 85 93 98
l **m** **r**

Since k< a[m] and 70 < 81

So, the element is present in the first half

Now, the array becomes
7 8
70 74
l **r**

m = (l + r) div 2
= (7+8) div 2
m = 7

7 8
70 74
l, m **r**

Now k = a[m] and 70 =70

Hence, the search key element 70 is found in the position 7 and the search operation is completed.

Algorithm Description and Implementation
Establish the array a[0….n-1] and the value to be found k.
Assign the l and r variables to the array limits.
While l<r do
Compute the middle position of the remaining array segment to be searched.
If the value found is greater than current middle then
Adjust l value according
else
Adjust r value according
If the array element at 'l' position is equal to the value to be found then
Return found and position
else
Return not found and -1.

Algorithm

```
// Non  recursive binary search
Algorithm BinSearch (var a, n elements, n, x: integer)
Var l,r,m:integer;
//Input: Given an array a[0….n-1] sorted in ascending order //and search key k.
//Output: An index of the array's element that is equal to k or //-1 if there is no such element.
begin
l:=0; r := n-1;
while (l≤ r) do
begin
mid := [(l + r) div 2];
if k = a[mid] then
return m;
else if k <a[m] then
r:=m-1;
else
l:=m+1;
end:
return -1
end
```

Efficiency of Binary Search

The standard way to analyze the efficiency is to count number of times search key is compared with an element of the array.

Worst Case Analysis

The worst case includes all arrays that do not contain a search key.

The recurrence relation for

$$C_{worst}(n) = C_{worst}(n/2) + 1, \quad \text{for } n > 1 \qquad \text{-----} \qquad (1)$$

Time required to compare left sublist or right sub list

one comparison made with middle element

$$C_{worst}(1) = 1 \text{ -------- } (2)$$

The above recurrence relation can be solved further .

assume $n=2^k$ the equation (1) becomes

$$\mathbf{C_{worst}(2^k) = C_{worst}(2^k/2) + 1}$$

$$C_{worst}(2^k) = C_{worst}(2^{k-1}) + 1 \text{ ------ } (3)$$

Using backward substation method , we can substitute

$$C_{worst}(2^{k-1}) = C_{worst}(2^{k-2}) + 1$$

Then equation (3) becomes

$$C_{worst}(2^k) = [C_{worst}(2^{k-2}) + 1] + 1$$

$$C_{worst}(2^k) = C_{worst}(2^{k-2}) + 2$$

Then

$$C_{worst}(2^k) = [C_{worst}(2^{k-3}) + 1] + 2$$

$$C_{worst}(2^k) = C_{worst}(2^{k-3}) + 3$$

$$C_{worst}(2^k) = C_{worst}(2^{k-k}) + k$$

$$C_{worst}(2^k) = C_{worst}(2^0) + k$$

$$C_{worst}(2^k) = C_{worst}(1) + k \quad \text{----- (4)}$$

But as per equation (2)

as we have assumed $n = 2^k$ taking logarithm (base 2)on both sides

$\log_2 n = \log_2 2^k$

$\log_2 n = k . \log_2 2$

$\log_2 n = k(1)$ therefore $\log_2 2 = 1$

therefore $k = \log_2 n$

$C_{worst}(1) = 1$ the we get equation (4)

$C_{worst}(2^k) = 1 + k$

$C_{worst}(n) = 1 + \log_2 n$ ----- (2)

$C_{worst}(n) = \log_2 n$ for $n>1$

The worst case time complexity of binary search is $\Theta(\log_2 n)$

As $C_{worst}(n) = \log_2 n + 1$

we can verify equation (1) with this value.

$C_{worst}(n) = C_{worst}[(n/2)] + 1$

In equation (1) put $n = 2i$

L.H.S

$C_{worst}(n) = \log_2 n + 1$

$= \log_2(2i) + 1$

$= \log_2 2 + \log_2 i + 1$

$= 1 + \log_2 i + 1$

$= 2 + \log_2 i$

$C_{worst}(n) = 2 + \log_2 i$

R.H.S

$C_{worst}(n/2)+1 = \log_2(2i/2) + 1$

$= \log_2 i + 1$

$= \log_2 2i + 1 + 1$

$= 2 + \log_2 i$

$C_{worst}(n/2) = 2 + \log_2 i$

L.H.S = R.H.S

Hence

$C_{worst}(n) = \log_2 n + 1$ and

$C_{worst}(i) = \log_2 i + 1$ are same

Hence

$C_{worst}(n) = \Theta(\log n)$

Average Case Analysis

To obtain average case efficiency of binary search we will consider some sample input n. if n = 1 i.e. only element 11 is there only one search is required to search some KEY.

If n = 2 and search key = 22

11	22
0	1

Two comparisons are made to search 22

Similarly n= 4, 8, 16 and search key = 44, 88

11	22	33	44
0	1	2	3

N	Total comparison (c)
1	1
2	2
4	3
8	4
16 .	5 .

Observing the above given table we can write

$\log_2 n + 1 = c$

for instance

if n= 2 then

$\log_2 2 = 1$

then $c = \log 2\ 2 + 1$

$c = 1 + 1$

$c = 2$

if n = 8 , then

$c = \log_2 n + 1$

$= \log_2 8 + 1$

$= 3 + 1$

$C = 4$

Average number of comparison made by the binary search is slightly smaller than worst case.

$C_{avg}(n) \approx \log_2 n$

The average number of comparison in the successful search is **$\Theta(\log 2n)$**

Advantages

- In this method elements are eliminated by half each time. So it is very faster than the sequential search.
- It requires less number of comparisons than sequential search to locate the search key element.

Disadvantages

- An insertion and deletion of a record requires many records in the existing table be physically moved in order to maintain the records in sequential order.
- The ratio between insertion/deletion and search time is very high.

Applications of binary search

- The binary search is an efficient searching method and is used to search desired record from database
- For solving nonlinear equations with one unknown this method is used.

Time complexity of binary search

Best case	Average case	Worst case
Θ(1)	Θ(log2n)	Θ(log2n)

Difference between sequential and binary search :

Sequential technique	binary search technique
This is the simple technique of searching an element	This is the efficient technique of searching an element
This technique does not require the list to be sorted	This technique require the list to be sorted. Then only this method is applicable
The worst case time complexity of this technique is O(n)	The worst case time complexity of this technique is O(log n)
Every element of the list may get compared with the key element.	Only the mid element of the list is compared with key element.

9. Explain the method of multiplication of large numbers with the help of illustrate example
Multiplication of Large Integer

In this method of multiplying two numbers multiplies the multiplicand by each digit of multiplier and then adds up all the properly shifted results.
This method is also called grade – school multiplication

For example

42
×34
= 168

But this method is not convenient for performing multiplication of large integers . hence let us discuss an interesting algorithm of multiplying large integer .
For example

To demonstrate the basic idea of the algorithm, let us start with a case of two-digit integers, say, 23 and 14.

These numbers can be represented as follows:

23 2. 101 + 3 .100 and 14 = 1 . 101 + 4 . 100
24 Now let us multiply both the numbers
$23 * 14 = (2 . 101 + 3 . 100) * (1 . 101 + 4 . 100)$
$= (2 * 1) 102 + (2 * 4 + 3 * 1) 101 + (3 * 4) 100$
=2*100+ (8 +3)10 +(12) 1
=200+110+12
= 322

Let us formulate this method Let

$c = a * b$

$c = c_2 10^2 + c_1 10^1 + c_0$ ------(1)

Where,

c2 = a1 ∗ b1 -->is the product of their first digits,

c0= a0 ∗ b0 -->is the product of their second digits,

c1 = (a1 + a0) ∗ (b1 + b0) − (c2 + c0) --> is the product of the sum of the a's digits and the sum of the b's digits minus the sum of c2 and c0

The 2 digit numbers are

a = a1 a0

b = b1 b0

Let perform multiplication operation with the help of formula given in equation (1)

$c = a * b$

c= 23 * 14

Where a1 = 2, a0 = 3 , b1 = 1, b0 = 4

Let us obtain c0, c1, c2 values

c2 = a1 ∗ b1

= 2 * 1

c2 = 2

c0= a0 ∗ b0

= 3 * 4

c0 = 12

c1 = (a1 + a0) ∗ (b1 + b0) − (c2 + c0)

= (2 + 3) *(1 + 4) – (2 +12)

= 5 * 5 – 14

c1 =25-14

c1 = 11

Therefore

$a * b = c2\ 10^2 + c1\ 10^1 + c0$

= 2 * 100 +11 * 10 +12

= 200 +110 +12

a * b = 322

We can generalize this formula as

$c = a * b$

$c = c2 10^2 + c1 10^1 + c0$

where , n is total number of digits in the integer

c2 = a1 ∗ b1

c0= a0 ∗ b0

c1 = (a1 + a0) ∗ (b1 + b0) − (c2 + c0)

Analysis

In this method there are 3 multiplication operations 1 digit numbers

i.e c2 = a1 ∗ b1 --> multiplication 1

c0 = a0 ∗ b0 --> multiplication 2

c1 = (a1 + a0) ∗ (b1 + b0) − (c2 + c0)->multiplication 3

The multiplication of n digit numbers requires three multiplications of n/2 digit numbers, the recurrence equation for the number of multiplication M(n) will be,

M(n)=3M(n/2), for n>1 And M(1)=1 where n = 1

Now put $n=2^k$ Solving it by backward substitutions for

$M(2^k)=3\ M(2^k /2)$

$M(2^k)=3\ M(2^{k-1})$

$=3[3\ M(2^{k-2})]$

$=3^2\ M(2^{k-3})$

…….

$=3^k\ M(2^{k-k})$

$=3^k\ M(2^0)$ Since $2^0 = 1$

$=3^k$

Using equation (3) , M(n) = 1

Therefore $M(2^k) = 3^k$ ------ (4)

as $n= 2^k$ we get k = log2n ,equation (4)

M(n)= 3 log 2n

=n log 2 3 therefore a log b c = c log2 a

≈ n 1.585

M(n) ≈ n1.585

12. Explain the working of Strassen's Matrix Multiplication with the help of divide and conquer method. April/May 2018

What is Strassen's matrix multiplication and explain how it is solves the problem using divide and conquer technique. (April/ May 2021)

The Strassen's matrix multiplication algorithm finds the product C of two 2 by 2 matrices A and B with just seven multiplication as opposed to eight required by the brute force algorithm.

It is accomplished by using the following formula.

C = A × B

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix},$$

The multiplication gives

C00 = a00 ×b00 +a01 ×b10

C01 = a00 ×b01 +a01 ×b11

C10 = a10 ×b01 +a11 ×b10

C11 = a10 ×b00 +a01 ×b11

Thus to accomplish 2 ×2 matrix multiplication there are total 8 multiplication and 4 additions

The divide and conquer approach can be used for implementing Strassen's matrix multiplication

Divide: divide matrices into sub – matrices : A0 , A1, A2 etc

Conquer: use a group of matrix multiply equations

Combine: recursively multiply sub – matrices and get the final result of multiplication after performing required additions or subtractions.

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

Where

m1 = (a00 + a11) * (b00 + b11)
m2 = (a10 + a11) * b00
m3 = a00 * (b01 - b11)
m4 = a11 * (b10 - b00)
m5 = (a00 + a01) * b11
m6 = (a10 + a00) * (b00 + b01)
m7 = (a01 + a11) * (b10 + b11)

Thus, to multiply two 2 by 2 matrices, strassen's algorithm makes seven multiplication and 18 additions/subtractions where as the brute force algorithm requires eight multiplication and for additions.

These numbers should not lead us to multiplying 2 by 2 matrices by strassen's algorithm.

Its importance stems from its asymptotic superiority as matrix order n goes to infinity.Let A and B be two n-by-n matrices where n is a power of two.

If n is not a power of two, matrices can be added with rows and column of zeros.The matrix A,B and their product is divided into 4,n/2 by n/2 submatrices each as follows

$$\left[\begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{array}\right] = \left[\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right] * \left[\begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array}\right].$$

The value C00 can be computed as either a00 * b00 or a01 * b10 or as M1 + M4 – M5 + M7 , where M1 , M4 , M5 and M7 are found by strassen's formula with the numbers replaced by the corresponding submatrices.

The seven products of n/2 and n/2 matrices are computed recursively by the strassen's matrix multiplication algorithm.

Efficiency of strassen's matrix multiplication

If M(n) is the number of multiplication made by strassen's algorithm in multiplying two n by n matrices where n is a power of 2, then the recurrence relation is

M(1) = 1

M(n)=7 M(n/2)

Since $n=2^k$ yields

$M(2^k)=7^k M(n/2^k)$

$=7[7 M(2^{k-2})]$

$=7^2 M(2^{k-2})$

$=7^i M(2^{k-i})$

$=7^k M(2^{k-k})$

$=7^k$

Since k=log2n

$M(n)= 7 \log_2 n$

$= n \log_2 7$

$\approx n^{2.807}$

Which is smaller than n^3 required by the brute force algorithm.
Numbers of multiplications are reduced by making extra additions.
To multiply two matrices of order n>1 , the algorithm needs to multiply seven matrices of order n/2 and make 18 additions of matrices of size n/2.
When n = 1 , no additions are made since two numbers are simply multiplied.
The number of additions A(n) made by the Strassen's algorithm given by recurrence

$A(n) = 7A(n/2) + 18(n/2)$, for $n > 1$

$A(1) = 0$

According to the Master Theorem.

$A(n) \in \Theta(n \log_2 n)$

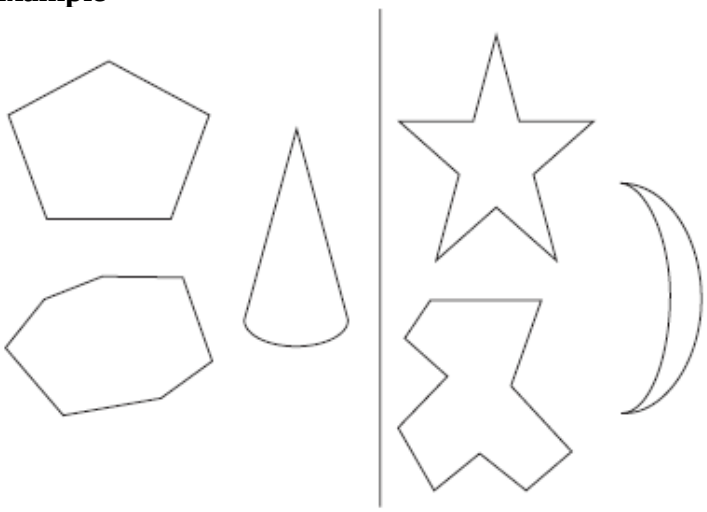
In other words, the number of additions has the same order of growth as the number of multiplications.
As a result, Strassen's algorithm in $\Theta(n \log_2 n)$, which is a better efficiency class than $\Theta(n^2)$ or the brute force method.

11. Write an algorithm for performing The Closest-Pair and Convex-Hull Problems by Divide-and-Conquer. *Dec 2005/2007/2010/2012/2013* **or Write the algorithm to find the closest pair of points using divide and conquer and explain it with an example. Derive the worst case and average case time complexity Nov/Dec 2019**

Definition

There exits a set of points on a plane which is said to be convex if for any two points A and B in the set , the entire line segment with the end points at A and B belongs to the set

Example



(a) (b)

Is a convex hull **Is not a convex hull**

The convex – hull problem of finding the smallest convex polygon that contains given n points in a plane can be solved using divide and conquer method.

This version of solving convex hull problem is called quick hull because this method is based on quick sort technique.

Algorithm

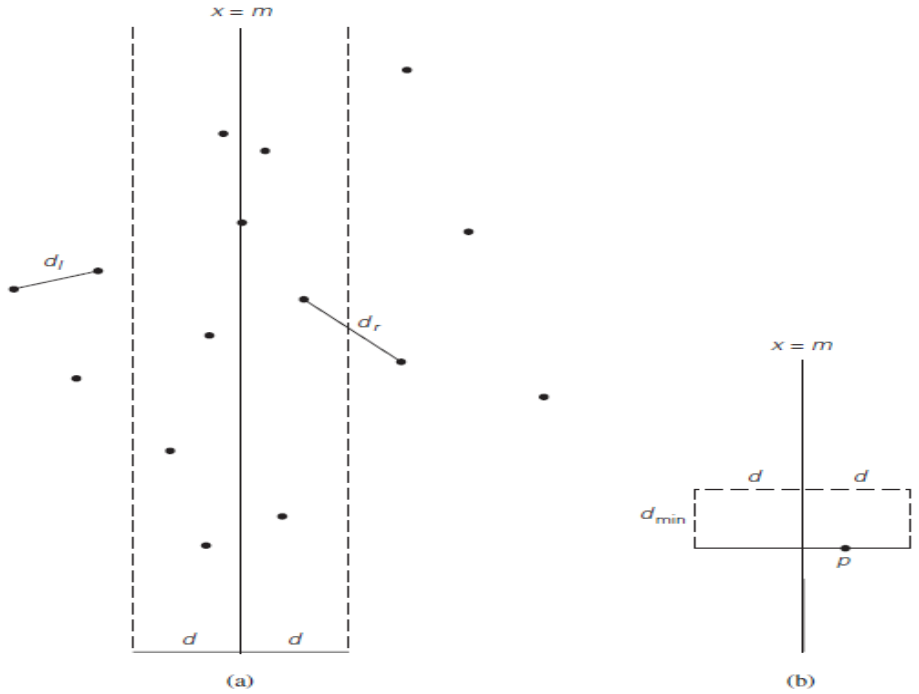
Step 1 : Sort the points (p1 ,p2 ,p3,…pn) by their x – coordinates
Step 2 : Repeatedly find the convex hull through p1 to p n/2
Step 3 : Repeatedly find the convex hull through p n/2+1 to p n
Step 4 : Merge the two convex hulls

The Closest-Pair Problem

- ✓ Let P be a set of n > 1 points in the Cartesian plane.
- ✓ For the sake of simplicity, we assume that the points are distinct.
- ✓ If $2 \leq n \leq 3$, the problem can be solved by the obvious brute-force algorithm.
- ✓ If n > 3, we can divide the points into two subsets Pl and Pr of n/2 and n/2 points, respectively, by drawing a vertical line through the median m of their x coordinates so that n/2 points lie to the left of or on the line itself, and n/2 points lie to the right of or on the line.

Then we can solve the closest-pair problem recursively for subsets Pl and Pr .
Let dl and dr be the smallest distances between pairs of points in Pl and Pr, respectively, and let d = min{ dl , dr }.

Note that d is not necessarily the smallest distance between all the point pairs because points of a closer pair can lie on the opposite sides of the separating line.



- Let S be the list of points inside the strip of width 2d around the separating line, obtained from Q and hence ordered in non decreasing order of their y coordinate.
- We will scan this list, updating the information about dmin, the minimum distance seen so far, if we encounter a closer pair of points.
- Initially, dmin = d, and subsequently dmin ≤ d. Let p(x, y) be a point on this list.
- For a point p(x, y) to have a chance to be closer to p than dmin, the point must follow p on list S and the difference between their y coordinates must be less than dmin.

ALGORITHM EfficientClosestPair(P, Q)
//Solves the closest-pair problem by divide-and-conquer
//Input: An array P of n ≥ 2 points in the Cartesian plane //sorted in non decreasing order of their x coordinates and an //array Q of the same points sorted in non decreasing order of //the y coordinates

```
//Output: Euclidean distance between the closest pair of //points
        if n ≤ 3
return the minimal distance found by the brute-force algorithm
        else
        copy the first _n/2_ points of P to array Pl
        copy the same _n/2_ points from Q to array Ql
        copy the remaining _n/2_ points of P to array Pr
        copy the same _n/2_ points from Q to array Qr
        dl←EfficientClosestPair(Pl, Ql)
        dr←EfficientClosestPair(Pr, Qr)
        d ←min{dl, dr}
        m←P[_n/2_ − 1].x
copy all the points of Q for which |x − m| < d into array S[0..num − 1]
        dminsq ←d2
        for i ←0 to num − 2 do
        k←i + 1
        while k ≤ num − 1 and (S[k].y − S[i].y)2 < dminsq
dminsq ←min((S[k].x − S[i].x)2+ (S[k].y − S[i].y)2, dminsq)
        k←k + 1
        return sqrt(dminsq)
```

The algorithm spends linear time both for dividing the problem into two problems half the size and combining the obtained solutions.

Therefore, assuming as usual that n is a power of 2, we have the following recurrence for the running time of the algorithm:

T (n) = 2T (n/2) + f (n), where f (n) ∈ Θ(n).

Applying the Master Theorem (with a = 2, b = 2, and d = 1),

we get T (n) ∈ Θ(n log n).

The necessity to presort input points does not change the overall efficiency class if sorting is done by a O(n log n) algorithm such as merge sort.

Convex-Hull Problem

✓ Let S be a set of n>1 points p1(x1, y1), . . . , pn (xn, yn) in the Cartesian plane.We assume that the points are sorted in non decreasing order of their x coordinates, with ties resolved by increasing order of the y coordinates of the points involved.

✓ It is not difficult to prove the geometrically obvious fact that the leftmost point p1 and the rightmost point pn are two distinct extreme points of the set's convex hull (Figure 5.8).

Let p1 pn be the straight line through point's p1 and pn directed from p1to pn.

This line separates the points of S into two sets:

S1 is the set of points to the left of this line, and S2 is the set of points to the right of this line. We say that point q3 is to the left of the line q1q2 directed from point q1 to point q2 if q1 q2 q3 forms a counterclockwise cycle.

Later, we cite an analytical way to check this condition, based on checking the sign of a determinant formed by the coordinates of the three points.

The points of S on the line p1 pn, other than p1 and pn, cannot be extreme points of the convex hull and hence are excluded from further consideration.

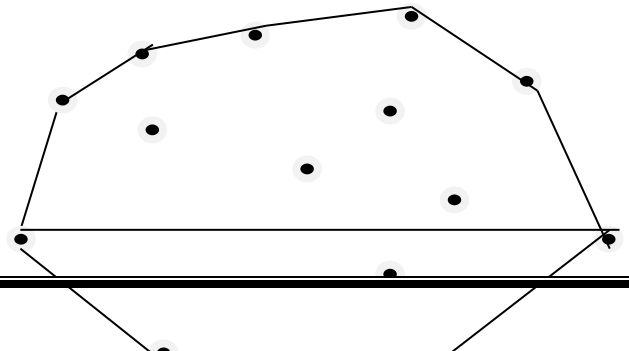


Fig Upper and lower hulls of a set of points

The boundary of the convex hull of S is made up of two polygonal chains: an "upper" boundary and a "lower" boundary.

The "upper" boundary, called the **upper hull,** is a sequence of line segments with vertices at p1, some of the points in S1 (if S1 is not empty) and pn.

The "lower" boundary, called the **lower hull**, is a sequence of line segments with vertices at p1, some of the points in S2 (if S2 is not empty) and pn.

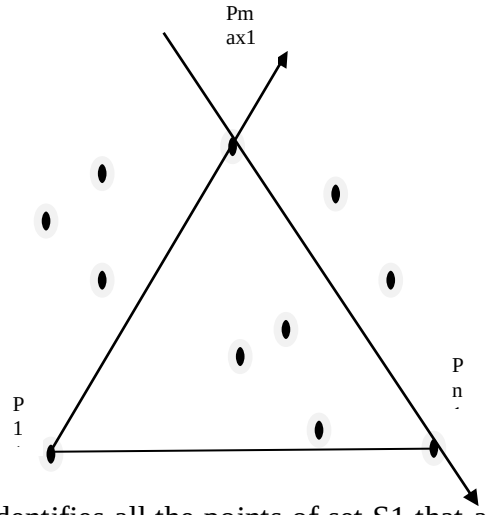
The fact that the convex hull of the entire set S is composed of the upper and lower hulls, which can be constructed independently.

Construction of upper hull:

If S1 is empty, the upper hull is simply the line segment with the endpoints at p1 and pn.

If S1 is not empty, the algorithm identifies point pmax in S1, which is the farthest from the line p1 pn .

If there is a tie, the point that maximizes the angle pmax p pn can be selected. (Note that point pmax maximizes the area of the triangle with two vertices at p1and pn and the third one at some other point of S1.)



- ✓ Then the algorithm identifies all the points of set S1 that are to the left of the line p1 pmax; these are the points that will make up the set S1,The points of S1 to the left of the line pmax pn will make up the set S1
- ✓ It is not difficult to prove the following: **pmax is a vertex of the upper hull**.
- ✓ The points inside p1 pmax pn cannot be vertices of the upper hull (and hence can be eliminated from further consideration).
- ✓ There are no points to the left of both lines **p1 pmax and pmax pn**.
- ✓ Therefore, the algorithm can continue constructing the upper hulls of p1 U S1,1 Upmax and pmax U S1,2 U pn recursively and then simply concatenate them to get the upper hull of the entire set p1 U S1 U pn.
- ✓ If q1(x1, y1), q2(x2, y2), and q3(x3, y3) are three arbitrary points in the Cartesian plane, then the area of the triangle q1q2q3 is equal to one-half of the magnitude of the determinant

x1 y1 1

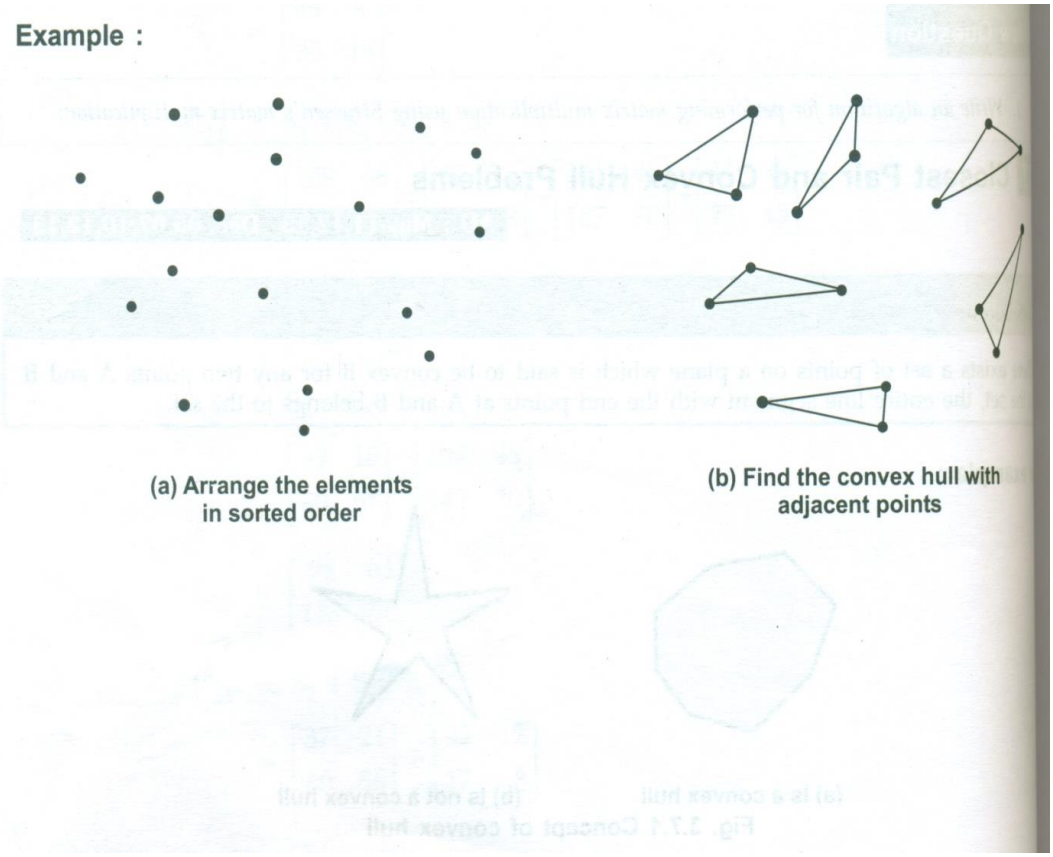
$$\begin{vmatrix} x2 & y2 & 1 \\ x3 & y3 & 1 \end{vmatrix} = x1\ y2 + x3\ y1 + x2\ y3 - x3\ y2 - x2\ y1 - x1\ y3$$

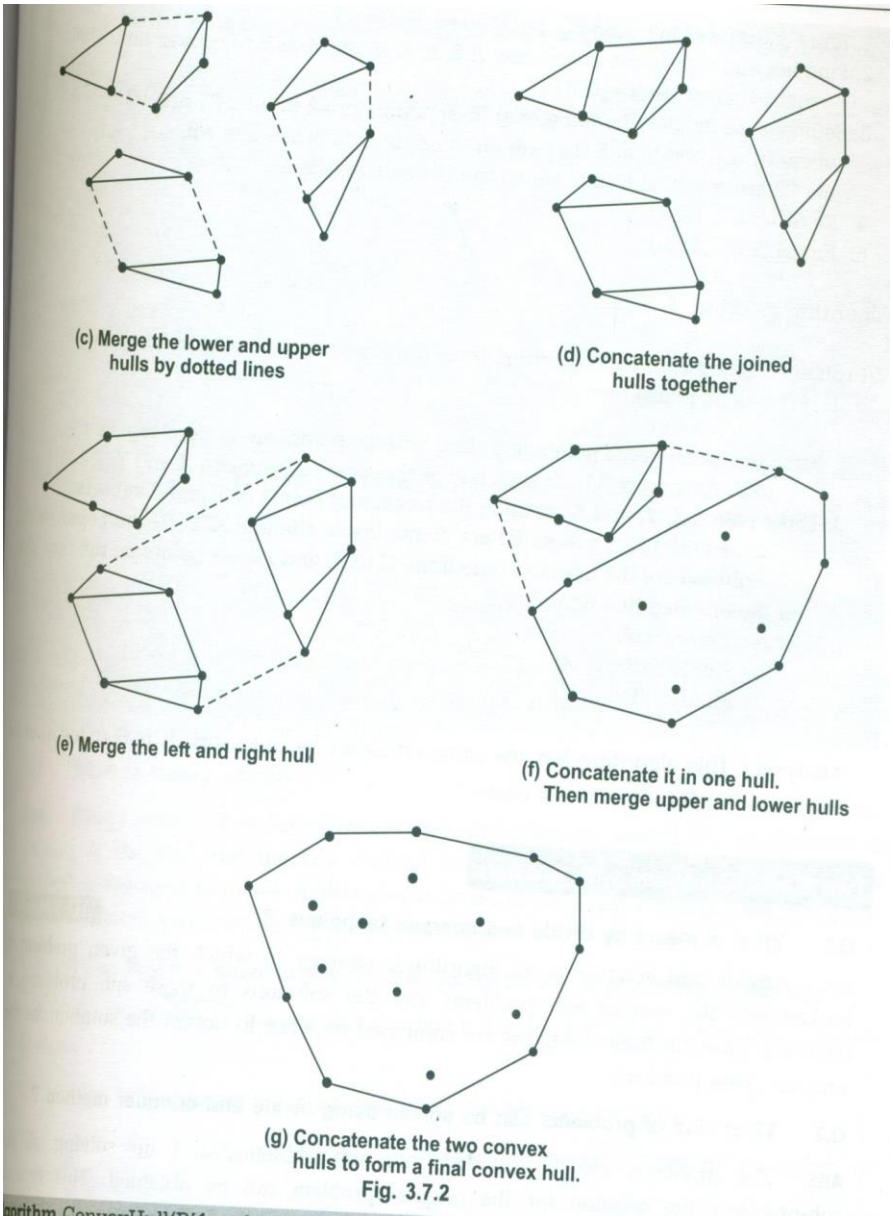
while the sign of this expression is positive if and only if the point q3 = (x3, y3) is to the left of the line q1 q2.

Using this formula, we can check in constant time whether a point lies to the left of the line determined by two other points as well as find the distance from the point to the line.

Example 2:

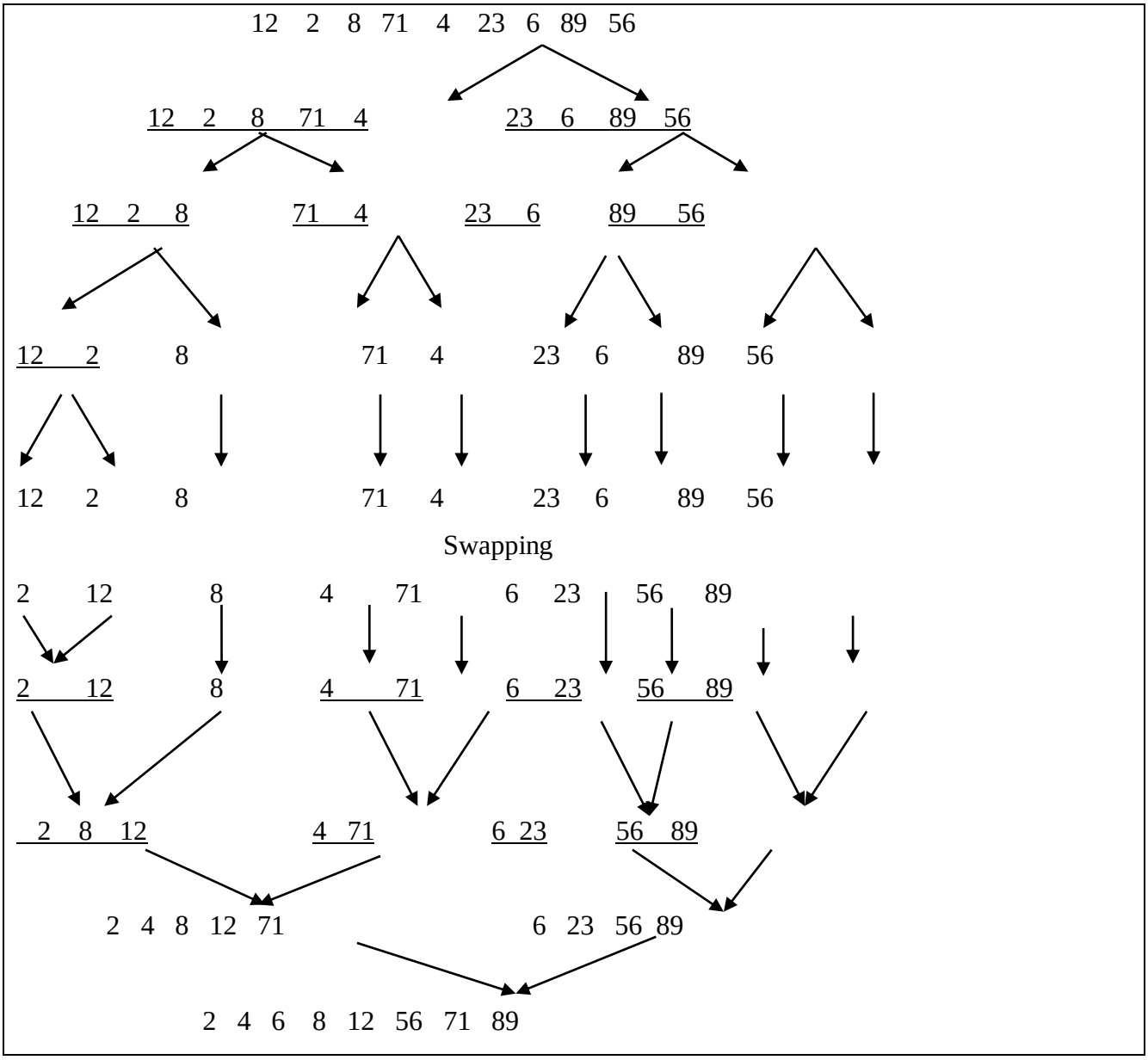
The merge procedure requires finding a bridge between two hulls that are adjacent to each other . concatenate left part of left hull and right part of right hull





Quick hull has the same Θ(n log n) worst-case efficiency as quick sort.In the average case $\Theta(n^2)$, however, we should expect a much better performance.
The average-case efficiency of quick hull turns out to be linear. 'n' its computing time is **$O(n^2)$.**

12. Sort the following set of elements using merge sort : 12, 2, 8, 71, 4, 23, 6, 89, 56
Jun 2014



13. Distinguish between quick sort and merge sort and arrange the following numbers in increasing order using merge sort (18, 29, 68, 32, 43, 37, 87, 24, 47, 50). *Jun 2013*

Quicksort

Quicksort is another divide and conquer sorting algorithm, proposed by C. A. R. Hoare. Here, the workhorse is the partition operation. Given an array of n elements, partition takes one element (known as the pivot) and places it in the correct position.

That is, the array will not be sorted, but all the elements less than the pivot will be to the left of the pivot, and all the elements greater than the pivot will be to the right. The partition operation can be performed in linear time.

Quicksort then works as follows. First, it performs the partition operation on the input array. Assume the pivot is placed in position p.

Now, quicksort sorts the sub-arrays A[0 : p - 1] and A[p + 1 : n - 1], using quicksort itself. Here again, the base case is an array of size 1.

Various ways of selecting the pivot:

The simplest is to just take the left-most element every time as the pivot. But, this leads to a fatal weakness.

In this case, if the array is already sorted, the pivot will get placed in the left most position always, and quicksort will be sorting sub-lists of size 0 and n - 1.

This leads to a time complexity of O(n^2), which is no better than bubble sort.

This can be mitigated by choosing a random element as the pivot, or using the median of three elements. In this case, the worst case time of O(n^2) is extremely unlikely.

In the best case, when the pivot is always placed in the middle, the time complexity will be O(n log n).

Also, if the pivot always gets placed somewhere in the middle 50% (25% - 75%) part of the array, quick sort will take time proportional to O(n log n), even as in the best case.

Advantage of quick sort:

Even though the asymptotic complexities are O(n log n), the constant muliplier (hidden by the Big Oh notation) is much smaller for quicksort, which subsequently is appreciably faster than mergesort in almost all cases.

Regarding space complexity, the space complexity of quicksort is O(log n), taking into account the stack space used for recursion.

Mergesort

The real work of mergesort is done by the merge operation. Given two sorted sub-arrays, together having a total of n elements,the merge operation uses an auxiliary array of size n to mergethem together into a single sorted array in linear time i.e. *O(n)* in Big Oh notation.

Having this merge operation, mergesort works as follows.

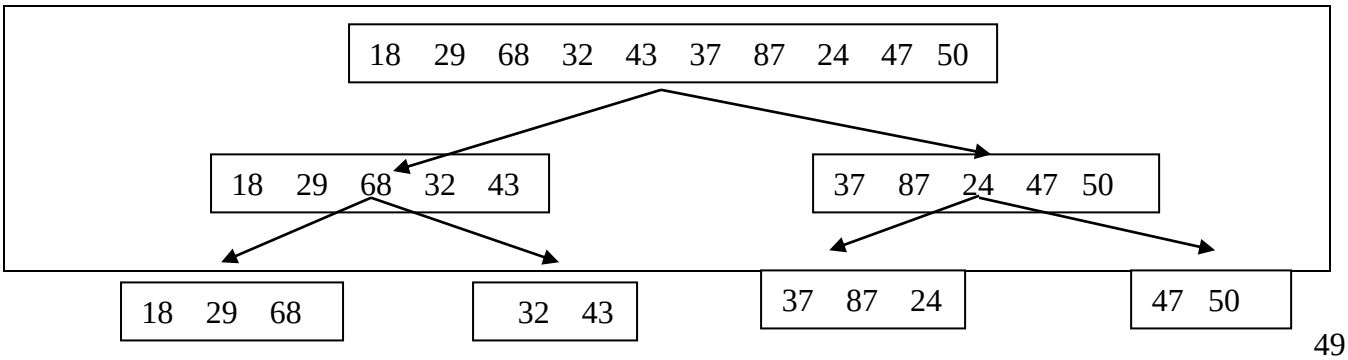
Given an array of elements A, it sorts the left and right sub-arrays using mergesort itself, and then merges them together into one single sorted array.

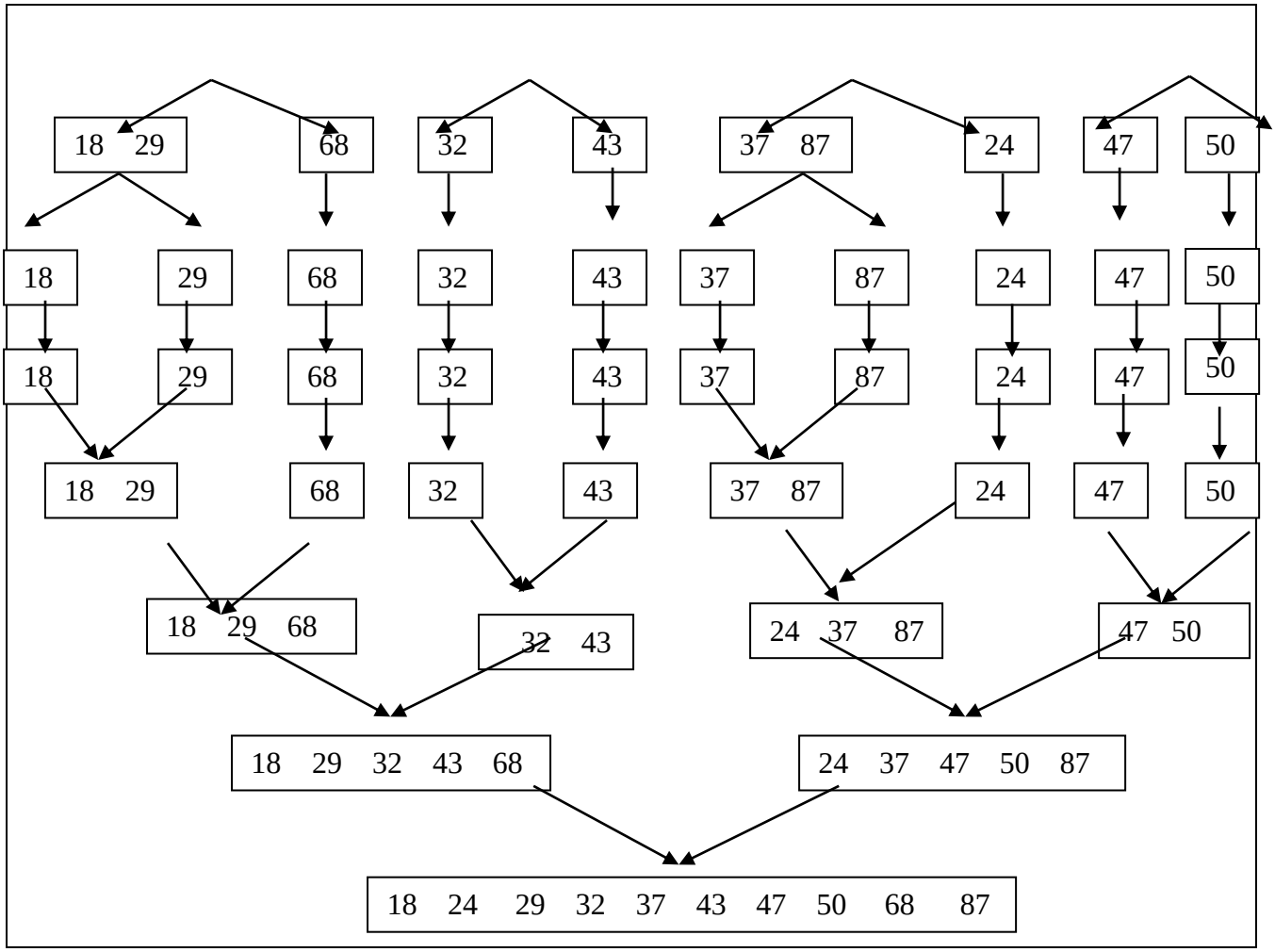
The base case is a sub-array of size 1, which is implicitly sorted by definition.

If we analyze the time complexity of mergesort, it is *O(n log n)* in all cases. That is, the time taken to sort *n* elements grows proportionally to *n log n.*

Merge sort also needs an extra array of size *n* for the merge operation. So its space complexity is *O(n)*.

Also, quicksort cannot be implemented iteratively, unlike mergesort, where an iterative implementation, sometimes called bottom-up mergesort, is possible.





14.A pair contains two numbers and second number is on right side of the first one in an array.the difference of a pair is the minus result while subtracting the second number from the first one.implement a function which gets the maximal difference of all pairs in array (using divide and conquer method) (AU april/may 2015)

We divide an array into two sub-arrays with same size.

The maximal difference of all pairs occurs in one of the three following situations:

(1) two numbers of a pair are both in the first sub-array;

(2) two numbers of a pair are both in the second sub-array;

(3) the minuend is in the greatest number in the first sub-array, and the subtrahend is the least number in the second sub-array.

It is not a difficult to get the maximal number in the first sub-array and the minimal number in the second sub-array. How about to get the maximal difference of all pairs in two sub-arrays?

They are actually sub-problems of the original problem, and we can solve them via recursion. The following are the sample code of this solution:

```
int MaxDiff_Solution1(int numbers[], unsigned length)
{
   if(numbers == NULL || length < 2)
      return 0;

   int max, min;
   return MaxDiffCore(numbers, numbers + length - 1, &max, &min);
```

```
}

int MaxDiffCore(int* start, int* end, int* max, int* min)
{
   if(end == start)
   {
      *max = *min = *start;
      return 0x80000000;
   }

   int* middle = start + (end - start) / 2;

   int maxLeft, minLeft;
   int leftDiff = MaxDiffCore(start, middle, &maxLeft, &minLeft);

   int maxRight, minRight;
   int rightDiff = MaxDiffCore(middle + 1, end, &maxRight, &minRight);

   int crossDiff = maxLeft - minRight;

   *max = (maxLeft > maxRight) ? maxLeft : maxRight;
   *min = (minLeft < minRight) ? minLeft : minRight;

   int maxDiff = (leftDiff > rightDiff) ? leftDiff : rightDiff;
   maxDiff = (maxDiff > crossDiff) ? maxDiff : crossDiff;
   return maxDiff;
}
```

In the function MaxDiffCore, we get the maximal difference of pairs in the first sub-array (leftDiff), and then get the maximal difference of pairs in the second sub-array (rightDiff).
We continue to calculate the difference between the maximum in the first sub-array and the minimal number in the second sub-array (crossDiff). The greatest value of the three differences is the maximal difference of the whole array.
We can get the minimal and maximal numbers, as well as their difference in O(1) time, based on the result of two sub-arrays, so the time complexity of the recursive solution is T(n)=2(n/2)+O(1). We can demonstrate its time complexity is O(n).

15.Explain the method used for performing Multiplication of two large integers.Explain how divide and conquer method can be used to solve the same. (16)

✓ Some applications like modern cryptography require manipulation of integers that are over 100 decimal digits long. Since such integers are too long to fit in a single word of a modern computer, they require special treatment.
In the conventional pen-and-pencil algorithm for multiplying two n-digit integers, each of the n digits of the first number is multiplied by each of the n digits of the second number for the total of n^2 digit multiplications.

The divide-and-conquer method does the above multiplication in less than n^2 digit multiplications.

Example: $23 * 14 = (2 \cdot 10^1 + 3 \cdot 10^0) * (1 \cdot 10^1 + 4 \cdot 10^0)$

$= (2 * 1)10^2 + (2 * 4 + 3 * 1)10^1 + (3 * 4)10^0$

$= 2\cdot 10^2 + 11\cdot 10^1 + 12\cdot 10^0$

$= 3\cdot 10^2 + 2\cdot 10^1 + 2\cdot 10^0$

$= 322$

The term $(2 * 1 + 3 * 4)$ computed as $2 * 4 + 3 * 1 = (2 + 3) * (1 + 4) - (2 * 1) - (3 * 4)$. Here $(2 * 1)$ and $(3 * 4)$ are already computed used. So only one multiplication only we have to do.

For any pair of two-digit numbers $a = a1a0$ and $b = b1b0$, their product c can be computed by the formula $c = a * b = c2 10^2 + c1 10^1 + c0$,

where

c2 = a1 * b1 is the product of their first digits,

c0 = a0 * b0 is the product of their second digits,

c1 = (a1 + a0) * (b1 + b0) − (c2 + c0) is the product of the sum of the a's digits and the sum of the b's digits minus the sum of c2 and c0.

Now we apply this trick to multiplying two n-digit integers a and b where n is a positive even number. Let us divide both numbers in the middle to take advantage of the divide-and-conquer technique.

We denote the first half of the a's digits by $a1$ and the second half by $a0$; for b,the notations are $b1$ and $b0$, respectively. In these notations, $a = a1a0$ implies that $a = a1 10^{n/2} + a0$ and $b = b1b0$ implies that $b = b1 10^{n/2} + b0$.

Therefore, taking advantage of the same trick we used fortwo-digit numbers, we get

$C = a * b = (a1 10^{n/2} + a0) * (b1 10^{n/2} + b0)$

$= (a1 * b1)10^n + (a1 * b0 + a0 * b1)10^{n/2} + (a0 * b0)$

$= c2 10^n + c1 10^{n/2} + c0,$

where

$c2 = a1 * b1$ is the product of their first halves,

$c0 = a0 * b0$ is the product of their second halves,

$c1 = (a1 + a0) * (b1 + b0) - (c2 + c0)$

If $n/2$ is even, we can apply the same method for computing the products $c2$, $c0$, and $c1$.

Thus, if n is a power of 2, we have a recursive algorithm for computing the product of two n-digit integers. In its pure form, the recursion is stopped when n becomes 1. It can also be stopped when we deem n small enough to multiply the numbers of that size directly.

The multiplication of n-digit numbers requires three multiplications of $n/2$-digit numbers, the recurrence for the number of multiplications $M(n)$ is $M(n) = 3M(n/2)$ for $n > 1$, $M(1) = 1$.

Solving it by backward substitutions for $n = 2^k$ yields

$M(2^k) = 3M(2^{k-1})$

$= 3[3M(2^{k-2})]$

$= 3^2 M(2^{k-2})$

$= \dots$

$= 3^i M(2^{k-i})$

$= \dots$

$= 3^k M(2^{k-k})$

$= 3^k$.

(Since $k = \log_2 n$)

$$M(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585}.$$

(On the last step, we took advantage of the following property of logarithms: $a^{\log_b c} = c^{\log_b a}$.)

Let *A(n)* be the number of digit additions and subtractions executed by the above algorithm in multiplying two *n*digit decimal integers. Besides 3*A(n/2)* of these operations needed to computethe three products of *n/*2digit numbers, the above formulas require five additions a nd one subtraction. Hence, we have the recurrence

$A(n) = 3 \cdot A(n/2) + cn$ for $n > 1$, $A(1) = 1$.

By using Master Theorem, we obtain $A(n) \in \Theta(n^{\log_2 3})$,

which means that the total number of additions and subtractions have the same asymptotic order of growth as the number of multiplications.

Example: For instance: a = 2345, b = 6137, i.e., n=4.

Then C = a * b = $(23*10^2+45)*(61*10^2+37)$

$C = a * b = (a1 10^{n/2} + a0) * (b1 10^{n/2} + b0)$

$= (a1 * b1)10^n + (a1 * b0 + a0 * b1)10^{n/2} + (a0 * b0)$

$= (23 * 61)10^4 + (23 * 37 + 45 * 61)10^2 + (45 * 37)$

$= 1403 \cdot 10^4 + 3596 \cdot 10^2 + 1665$

$= 14391265$

16. Let x1< x2 < . . . < xn be real numbers representing coordinates of n villages located along a straight road. A post office needs to be built in one of these villages. a) Design an efficient algorithm to find the post-office location minimizing the average distance between the villages and the post office. (april/May 2021)

Algorthm :

Algorithm PostOffice(P)

```
m <- (x1+xn) / 2
i <- 1
while xi < m do
   i <- i+1
if xi - x1 < xn - xi-1
   return xi
else return xi-1
```

IMPORTANT QUESTIONS

Part A

1. What is the time complexity of Binary search? *June 2011 & 12*
2. Give the recurrence equation for the worst case behavior of merge sort? *Dec 2010*
3. What do you meant by Divide and conquer strategy? *May 2013*
4. Give the time efficiency and drawback of merge sort algorithm? *Dec 2005*
5. What is the difference between quick sort and merge sort? *May 2013*
6. What is the difference between sequential and binary search *Apr 2013*
7. List out two drawbacks of binary search algorithm. *Dec 2007*
8. Give the control abstraction for divide and conquer. *Dec 2012*
9. What is called substitution method? *Jun 2010*
10. What is called optimal solution? *Jun 2010*
11. What do you mean by divide and conquer strategy? *Jun 2013*
12. State the principle of substitution method? *Jun 2014*
13. Define feasible and optimal solution? *Jun 2014*
14. Trace the operation of binary search algorithm for the input – 15, -6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151, if you are searching for the element 9. *Dec 2010*
15. Define Brute Force method
16. State the concept of Closest-Pair and Convex-Hull Problems
17. What is mean by Exhaustive Search and give the types
18. What is Travelling Salesman Problem?
19. What is knapsack?
20. What is assignment problem?
21. Is merge sort stable sorting algorithm?
22. What is the difference between quick sort and merge sort?
23. Define Stassen's Matrix Multiplication

Part B

1. Write an algorithm for performing The Closest-Pair and Convex-Hull Problems by Divide-and-Conquer*2013,12,11*
2. Write an algorithm to perform binary search on a sorted list of elements. Analyse the algorithm for the best case , worst case and average case. *May 2011 / Dec 2008*

   Or

   What is divide and conquer strategy and explain the binary search with suitable example problem. *Dec 2011*

   Or

   Differentiate sequential search from binary search technique. *May 2009*
3. Briefly explain the procedure for strassen's matrix multiplication. *Mar2014*
4. Trace the steps of merge sort algorithm for the elements 122, 25, 70, 175, 89, 90, 95, 102, 123 and also compute its time complexity. *Dec 2012*
5. Explain the binary search algorithm with an example. And find the best, average and worst case complexity. *Dec 2012*
6. Explain merge sort problem using divide and conquer technique example. *Apr2010*
7. Write a pseudo code using divide and conquer technique for finding the position of the largest element in the array of N numbers. *Jun 2014*
8. Sort the following set of elements using merge sort :12,2,8,71,4,23, 6, 89, 56 *Jun14*
9. Distinguish between quick sort and merge sort and arrange the following numbers in increasing order using merge sort (18, 29,68, 32, 43, 37, 87, 24, 47, 50). *Jun13*
10. Explain the method of multiplication of large numbers with the help of illustrate example
11. Write an algorithm for performing matrix multiplication using Strassen's Matrix Multiplication
12. Write an algorithm for performing The Closest-Pair and Convex-Hull Problems by Divide-

and-Conquer. *May 2011/2012/2013*

ANNA UNIVERSITY APRIL/MAY 2015

PART-A

1. Design a brute force algorithm for computing the value of a polynomial **Refer Q. No. 39**
2. Derive complexity of binary search algorithm. **Refer Q. No. 40**

PART-B

1. a) A pair contains two numbers and second number is on right side of the first one in an array.the difference of a pair is the minus result while subtracting the second number from the first one.implement a function which gets the maximal difference of all pairs in array (using divide and conquer method) **Refer Q. No. 21**

b) Explain convex hull problem and the solution involved behind it. **Refer – Q. No. 2**

ANNA UNIVERSITY NOV/DEC 2015

PART-A

1. Give the mathematical notation to determine if a convex direction is towards left or right and write the algorithm. **Refer Q. No. 45**
2. Prove that any comparison sort algorithm requires Ω (n log n) comparisons in the worst case. **Refer Q. No. 44**

PART-B

1.a.(i) Write the algorithm to perform binary search and compute its run time complexity.(8) **Refer Q. No. 7**

(ii)Compute multiplication of given two matrices using mstrassen's matrics multiplication method: **Refer Q. No. 9**

$$A=\begin{bmatrix}1&0&2&1\\4&1&1&0\\0&1&3&0\\5&0&2&1\end{bmatrix}\quad B=\begin{bmatrix}0&1&0&1\\2&1&0&4\\2&0&1&1\\1&3&5&0\end{bmatrix}.$$

b (i) Write doun the algorithm to construct a convex hull based on divide and conquer strategy.(8) **Refer Q. No. 10**

(ii)Find the optimal solution to the fractional knapsack problem with given data: **ReferQ.No. 3(2)**

Item	Weight	Benefit
A	2	60
B	3	75
C	4	90

ANNA UNIVERSITY APRIL/MAY 2016

PART-A

1. Give the General strategy divide and conquer method. **Refer Q. No. 41**
2. What is Closest-Pair Problem? **Refer Q. No. 4**

PART-B

1. (a) Explain the method used for performing Multiplication of two large integers.Explain how

Divide and conquer method can be used to solve the same.(16) **Refer Q. No. 14**

OR

(b) State and Explain Mergesort algorithm and give the recurrence relation and Efficiency (16) **Refer Q. No. 4(1)**

ANNA UNIVERSITY NOV/DEC 2016

PART-A

1. write an algorithm for brute force closest-pair problem **Refer Q. No. 43**
2. what is worst case complexity of binary search? **Refer Q. No. 13**

PART-B

1.(a) There are 4 people who need to be assigned to execute 4 jobs(one person per job)and the Problem id to find an assignment with the minimum total cost. The assignment costs is given Below, solve the assignment problem by exhaustive search. (16) **Refer Q. No. 3(3)**

	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

b. Give the algorithm for quick sort. with an example show that quicksort is not a stable sorting algorithm.(16) **Refer Q. No. 6**

ANNA UNIVERSITY APRIL/MAY 2017

PART -A

1. What is Closest-Pair Problem? **Refer Q. No. 4**
2. Devise an algorithm to make for 1655 using the Greedy strategy. The coins available are { 1000,500,100,50,20,10,5} **Refer Q. No. 45**

PART -B

1. What is divide and conquer strategy and explain the binary search with suitable example problem. **Refer Q. No. 7**

2. Solve the following using Brute- Force algorithm
   Find whether the given string follows the specied pattern and return 0 or 1 accordingly
   Examples:

   (i). Pattern: “abba”, input:”redblueeredblue” should return 1
   (ii). Pattern: “aaaa”, input:”asdasdasdasd” should return 1
   (iii). Pattern: “aabb”, input:”xyzabcxzyabc” should return 0 . **Refer Q. No. 7**

PART -C

1. Write the algorithm for Quick Sort and write its time complexity with example list are 5, 3, 1,9, 8, 2, 4, 7. **Refer Q. No. 6**

ANNA UNIVERSITY NOV/DEC 2017

PART-A

1. Give the general plan of divide and conquer algorithm. **Refer Q. No.41**
2. Write advantage of insertion sort? **Refer Q. No.46**

PART-B

1. Explain the brute force method to find the two closest points in a set of n points in K-Dimensional

space **Refer Q. No.2**

2. Explain the working of Merge Sort Algorithm with an example**. Refer Q. No.5**

ANNA UNIVERSITY APRIL/MAY 2018

PART-A

1. What is an exhaustive search ? **Refer Q. No.48**
2. State Master's theorem. **Refer Q. No.49**

PART-B

1. Explain Merge sort algorithm with an example. **Refer Q. No.6**
2. Explain the working of Strassen's Matrix Multiplication with the help of divide and conquer method. **Refer Q. No.10**

ANNA UNIVERSITY NOV/DEC 2018

PART-A

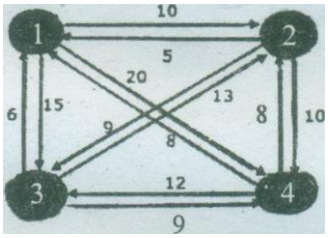
1. What are the differences between dynamic programming and divide and conquer approaches? **Refer Q. No.50**

2. Give an example for Hamiltonian circuit. **Refer Q. No.51**

PART-B

1. (a) Solve travelling salesman problem using brute force approach for the given example. How the solution can be obtained using branch and bound method?

Refer Q. No.4 (10 + 3)



Or

(b) Write the algorithm for quick sort. Provide a complete analysis of quick sort for the given set of numbers 12, 33, 23, 43, 44, 55, 64, 77 and 76. **Refer Q. No.7**

ANNA UNIVERSITY APRIL/MAY 2019

PART-A

1. Write brute force algorithm to string matching. **Refer Q. No.52**
2. What is time and space complexity of merge sort? **Refer Q. No.53**

PART-B

1. What is convex hull problem? explain the brute force approach to solve convex hull with an example. derive the time complexity(2+7+4) **Refer Q. No.3**

2. Write the quick sort algorithm and explain it wiyh example.derive the worst case and average case tiome complexity.(5+4+4) **Refer Q. No.7**

ANNA UNIVERSITY NOV/DEC 2019

PART-A

1. State the convex hull problem **Refer Q. No.6**
2. Outline the knapsack problem **Refer Q. No.9**

PART-B

1. State the travelling salesman problem. Elaborate the steps in solving the travelling salesman problem using brute force approach. **Refer Q. No.4**
2. Write the algorithm to find the closest pair of points using divide and conquered and explain it with an example. Derive the worst case and average case time complexity. **Refer Q. No.11**

PART-C

1. Sort the following numbers using quick sort. **Refer Q. No.7**
   999,888,777,666,555,444,333,222,111,11,22,33,44,55,66,77,88,99
   Illustrate each step in sorting process.

ANNA UNIVERSITY NOV/DEC 2021

PART-A

1. Write an example problem that cannot be solved by brute-force algorithm. Justify your answer. It is often implemented by computers, but it cannot be used to solve complex problems such as the **travelling salesman problem** or the game of chess, because the number of alternatives is too large for any computer to handle. **Refer Q.No.54a**

2. Write the general divide and conquer approach to solve a problem. **Refer Q.No.41**

PART-B

1.a) i) Consider the problem of counting, in a given text the number of substrings that start with an A and end with a B. For example, there are four such substrings in CABAAXBYA. Design a brute-force algorithm for this problem and determine its efficiency class. **Refer Q.No.2**

2.a) (i) Let x1 < x2 < ... < xn be real numbers representing coordinates of n villages located along a straight road. A post office needs to be built in one of these villages. Design an efficient algorithm to find the post-office location minimizing the average distance between the villages and the post-office. **Refer Q.No.16**

(ii) Explain how exhaustive search can be applied to the sorting problem and determine the efficiency class of such an algorithm. **Refer Q.No.4**

2.b) (i) Write an algorithm using divide and conquer to search an element in a list of numbers. If the number is not present, the algorithm returns the closest number of the searches number. **Refer Q.No.8**

(ii) What is Strassen's matrix multiplication and explain how it is solves the problem using divide and conquer technique. **Refer Q.No.12**

ANNA UNIVERSITY NOV/DEC 2021

PART-A

1. What is Travelling Salesman Problem?
2. How binary search algorithm works. **Refer Q. No. 40**

PART-B

.1. (a) Elaborate how a brute force algorithm works with an example. (13)

(b) Apply the merge sort algorithm to sort the following numbers in ascending order : 999,99,888,88,777,77,666,66,555,55,444,44,333,33,222,22,111, 11
Illustrate each step of the sorting process.

PART-C

1. Outline the steps to solve the travelling sales man problem using branch and bound technique.

UNIT-III
DYNAMIC PROGRAMMING AND GREEDY TECHNIQUE
PART A

1. Write the difference between Greedy method and Dynamic programming. *May 2011*

Compare and contrast dynamic programming and greedy method. (April/May 2021)

Greedy Method	Dynamic Programming
It is used to find optimal solution among all feasible solution	Enumerable all decision sequences and then pick the best.
Solutions to the sub problems do not have to be known at each stage.	Choice can be made of what looks best for the moment.

2. Write an algorithm to find shortest path between all pairs of nodes. *May 2011*

```
Algorithm AllPaths(cost, A, n)
// cost[1:n,1:n] is the cost adjacency matrix of a graph with
// n vertices; A[i,j] is the cost of a shortest path from vertex
// i to vertex j. cost[i,i] = 0.0, for 1 ≤ i ≤ n.
{
    for i := 1 to n do
        for j := 1 to n do
            A[i,j] := 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]);
}
```

3. Write any two characteristics of Greedy Algorithm?

- To solve a problem in an optimal way constructs the solution from given set of candidates.
- As the algorithm proceeds, two other sets get accumulated among this one set contains the candidates that have been already considered and chosen while the other set contains the candidates that have been considered but rejected.

4. What is an optimal solution? *May 2010*

A feasible solution either maximizes or minimizes the given objective function is called as optimal solution

5. What is Knapsack problem? *Dec 2011*

- A bag or sack is given capacity C and n objects are given.
- Each object has weight w_i and profit p_i .
- Fraction of object is considered as x_i (i.e) $0 \le x_i \le 1$.
- If fraction 1 then entire object is put into sack.
- When we place this fraction into the sack we get $w_i x_i$ and $p_i x_i$.

6. Define weighted tree.

A directed binary tree for which each edge is labeled with a real number (weight) is

called as weighted tree.

7. What is the Greedy choice property?

- The first component is greedy choice property (i.e.) a globally optimal solution can arrive at by making a locally optimal choice.
- The choice made by greedy algorithm depends on choices made so far but it cannot depend on any future choices or on solution to the sub problem. It progresses in top down fashion.

8. What is greedy algorithms? *Dec 2011*

- Greedy method is the most important design technique, which makes a choice that looks best at that moment.
- A given 'n' inputs are required us to obtain a subset that satisfies some constraints that is the feasible solution.
- A greedy method suggests that one can device an algorithm that works in stages considering one input at a time.

9. State the general principle of greedy algorithm? *Dec 2010 Apr/May-2017*

- Determine the optimal substructure of the problem.
- Develop a recursive solution.
- Prove that at any stage of recursion one of the optimal choices is greedy choice. Thus it is always safe to make greedy choice.
- Show that all but one of the sub problems induced by having made the greedy choice is empty.
- Develop a recursive algorithm and convert into iterative algorithm.

10. What is the limitation of Greedy algorithm? *May 2010*

- An optimization problem:
  - Given a problem instance, a set of constraints and an objective function.
  - Find a feasible solution for the given instance for which the objective function has an optimal value
  - either maximum or minimum depending on the problem being solved.
- A feasible solution satisfies the problem's constraints
- The constraints specify the limitations on the required solutions.

11. State or Define the principle of optimality. Apr/May 2019 *Dec 2010,Nov/Dec 2017*

The principle of optimality states that an optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision.

12. What is dynamic programming?or what do you mean by dynamic programming? Apr/May-2017

- Dynamic programming is typically applied to optimization problems. For a given problem we may get any number of solutions. from all those solutions we seek for optimum solution (minimum value or maximum value solution).
- And such an optimal solution becomes the solution to the given problem.

13. Compare feasible and optimal solution *May 2008*

Optimal solution

A feasible solution either maximizes or minimizes the given objective

function is called as optimal solution

Feasible Solution

For solving the particular there exits n inputs and we need to obtain a subset that satisfies some constraints . then any subset that satisfies these constrains is called feasible solution.

14. What are the Applications of Dynamic Programming?

- Multistage Graph
- Optimal Binary Search Tree (OBST)
- 0/1 Knapsack Problem
- Travelling Salesman Problem.
- All Pair Shortest Path Problem

15. Define Warshall's algorithm.

Warshall's algorithm is an application of dynamic programming technique which is used to find the transitive closure of a directed graph.

16. Define Floyd's algorithm.or What does Floyd's algorithm do? Nov/Dec 2017

- Floyd's algorithm is an application of dynamic programming, which is used to find the all pairs shortest path problem.
- It is applicable to both directed and undirected weighted graph, but they do not contain a cycle of negative length.

17.What are optimal binary search trees OBST? *May 2010*

- Let { a1, a2,….an} be a set of identifiers such that a1<a2<a3…let p(i) be the probability with which we can search for a_i is Successful search.
- Let , q_i be the probability of searching an element x such that $a_i<x<a_i+1$ where $0\leq i\leq n$ is unsuccessful search . thus p(i) is probability of successful search and q(i) is the probability of unsuccessful search.
- Then a tree which is build with optimum cost from

$\sum_{i=1}^{n} p(i) \sum_{i=1}^{n} q(i)$ is called optimal binary search tree

18. What is a Feasible solution ? *Dec 2013 / May 2014*

For solving the particular there exits n inputs and we need to obtain a subset that satisfies some constraints . then any subset that satisfies these constrains is called feasible solution.

19. State the applications of Huffman 's tree

Application of Huffman trees:

1. Huffman encoding is used in file compression algorithm
2. Huffman's code is used in transmission of data in an encoded form
3. This encoding is used in game playing method in which decision trees need to be formed

subset paradigm	**ordering paradigm**
At each step the decision about the input is	In this paradigm , the decision is made by

made. That means at each steps it is decided whether the particular input is in an optimal solution or not	considering the inputs in some order . this paradigm is useful for solving those problems that do not call for selection of optimal subset in greedy manner

20 . Differentiate between subset paradigm and ordering paradigm *Dec 2012*

21. What is the drawback of greedy algorithm ? *May 2012*

- Greedy method is comparatively efficient than divide and conquer but there is no as such guarantee of getting optimum solution
- In Greedy method , the optimum selection is without revising previously generated solutions

22. Write control abstraction for the ordering paradigm. *May 2012*

```
Algorithm store (n, limit)
{
        j = 0;
        For( i ← 1 to n ) do
        {
                Write ("append program", i);
                Write ("permutation for tap ",j);
                j = ( j+1) mod limit ;
        }
}
```

23. What is minimum Spanning tree? *Dec 2010 APR 2018*

- A Minimum Spanning tree of a weighted graph connected graph G is its spanning tree of the smallest weight, where the weight of a tree is defined as the sum of the weights on all its edges.
- The total number of edges in minimum spanning tree (MST) is |V|-1 where V is the number of vertices.

24. Give any two properties of dynamic programming approaches.

Optimal substructure:

The dynamic programming technique makes use principle of optimality to find the optimal solution from subproblems.

Overlapping Sub-problems:

- The dynamic programming is a technique in which the problem
- is divided into subproblems.
- The solutions of subproblems are shared to get the final solution
- to the problem.
- It avoids repetition of work

25. Give the commonly used designing steps for dynamic programming algorithm?

Dynamic programming design involves 4 major steps

- Characterize the structure of optimal solution.
- Recursively defines the value of an optimal solution.
- By using bottom up technique compute value of optimal solution.
- Compute an optimal solution from computed information.

26. What does dynamic programming have in common with divide and conquer?

- Both the divide and conquer and dynamic programming solve the problem by

breaking it into number of sub-problems.

- In both these methods solutions from sub-problems are collected together to form a solution to given problem.

27. Define Catalan number.

The total number of binary search trees with n keys is equal to nth catalan number.

$C(n) = (2n \text{ to } n)1/(n+1)$ for $n>0$

$C(0) = 1$

28. State time and space efficiency of OBST.

Space Efficiency : Quadratic

Time Efficiency : Cubic

29. Compare Greedy technique with dynamic programming method. *Dec 2012* – Q. No. 1

30. What is 0 / 1 knapsack problem. *Dec 2012*

Given a knapsack with maximum capacity W, and a set S consisting of n items

- Each item i has some weight wi and benefit value bi (all wi, bi and W are integer values)
- Problem: How to pack the knapsack to achieve maximum total value of packed items?
  Problem, in other words, is to find $\sum b_i$ subject to $\sum w_i \leq W$
- The problem is called a "0-1" problem, because each item must be entirely accepted or rejected.

31. What is an optimal solution? *May 2010*Refer Part A – Q. No. 4

32. Define optimal binary search tree. *May 2010*Refer Part A – Q. No. 17

33. Define feasible and optimal solution. *Jun 2014* Refer Part A – Q. No. 13

34. State the principle of optimality. *Jun 2014 / Dec 2010* Refer Part A – Q. No. 11

35. List out the advantages of dynamic programming. *Jun 2014*

Dynamic programming enables you to develop sub solutions of a large program.

The sub solutions are easier to maintain use and debug.And they possess overlapping also that means we can reuse These sub solutions are optimal solutions for the problem

36. What is knapsack problem? *Jun 2013*Refer Part A – Q. No. 5

37. Write a note on Greedy approach. *Mar 2014* Refer Part A – Q. No. 8

38. Define dynamic programming. *Mar 2014*Refer Part A – Q. No. 12

39. What is memory function? *Mar 2014*

Memory functions use a dynamic programming technique called memorization in order to relieve the inefficiency of recursion that might occur.

It is based on the simple idea of calculating and storing solutions to sub problems so that the solutions can be reused later without recalculating the sub problems again.

The best known example that takes advantage of memorization is an algorithm that computes the Fibonacci numbers.

```
Recursive_Fibonacci (n)
 {
   if (n == 0)
     return 0
   else if ( n == 1)
     return 1
   else
```

```
    return Recursive_Fibonacci(n-1)+ Recursive_Fibonacci(n-2)
}
```

An algorithm that uses recursion, runs in an exponential CPU time:

40. State the general principle of greedy algorithm. *Dec 2010* Refer Part A – Q. No. 9

41. Compare divide and conquer with dynamic programming and Dynamic programming with greedy technique. *Dec 2010*

Divide and Conquer	**Dynamic Programming**
The divide-and-conquer paradigm involves three steps at each level of the recursion: • Divide the problem into a number of sub problems. • Conquer the sub problems by solving them recursively. If the sub problem sizes are small enough, however, just solve the sub problems in a straightforward manner. • Combine the solutions to the sub problems into the solution for the original problem.	The development of a dynamic-programming algorithm can be broken into a sequence of four steps. a. Characterize the structure of an optimal solution. b. Recursively define the value of an optimal solution. c. Compute the value of an optimal solution in a bottom-up fashion. d. Construct an optimal solution from computed information
They call themselves recursively one or more times to deal with closely related sub problems.	Dynamic Programming is not recursive.
D&C does more work on the sub-problems and hence has more time consumption.	DP solves the sub problems only once and then stores it in the table.
In D&C the sub problems are independent of each other.	In DP the sub-problems are not independent.
Example: Merge Sort, Binary Search	Example : Matrix chain multiplication

Dynamic Programming	**Greedy Technique**
Focuses on principle of optimality.	Greedy method focuses on expanding partially constructed solutions.
It provides specific answers.	It provides many results such as feasible solution.
Less efficient	More efficient

42.Write doun the optimization techniques used for warshall's algorithm.state the rules and assumption which are implied behind that (AU april/may 2015)

Dynamic programming algorithms are used for optimization (for example, finding the shortest path between two points, or the fastest way to multiply many matrices). A dynamic programming algorithm will examine the previously solved subproblems and will combine their solutions to give the best solution for the given problem. The alternatives are many, such as using a greedy algorithm, which picks the locally optimal choice at each branch in the road. The locally optimal choice may be a poor choice for the overall solution. While a greedy algorithm does not guarantee an optimal solution, it is often faster to calculate. Fortunately, some greedy algorithms (such as minimum spanning trees) are proven to lead to the optimal solution.

43. Define the single source shortest path problem.

Dijkstra's algorithm solves the single source shortest path problem of finding shortest paths from a given vertex(the source), to all the other vertices of a weighted graph or digraph. Dijkstra's algorithm provides a correct solution for a graph with non negative weights.

44. State Assignment problem.

There are n people who need to be assigned to execute n jobs, one person per job. (That is,each p erson is assigned to exactly one job and each job is assigned to exactly one person.) The cost that would accrue if the ith person is assigned to the jth job is a known quantity

$C[i, j]$ for each pair

$i, j = 1, 2, \ldots , n$. The problem is to find an assignment with the minimum total cost.

45.How to calculate the efficiency of dijkstra's Algorithm?

- The time efficiency of Dijkstra's algorithm depends on the structure used for implementing the priority queue and for representing as input graph.
- The efficiency is $\Theta(|V|^2)$ for graphs represented by their weight matrix and the priority queue implemented as an unordered array.
- The efficiency is $\Theta(|E|\log|V|)$ for graphs represented by the adjacency linked list and the priority queue implemented as a min heap.
- Better efficiency can be achieved if priority queue is implemented using a sophisticated data structure called the **Fibonacci Heap.**

46.State how binomial co-efficient is computes

Computing a Binomial Coefficient is a typical example of applying dynamic programming in mathematics, particularly in combinatory.

Binomial Coefficient is a Coefficient of any of the term in the expansion of $(a+b)^n$.

The binomial coefficient is denoted by C(n, k) or $\binom{n}{k}$

The binomial coefficient is the number of combinations or subsets of K elements from an n element set($0 \leq k \geq n$).

The name binomial coefficient comes from the participation of these numbers in the binomial formula.

The binomial formula is

$(a+ b)^n = C(n.0)a^n + \text{---} + C(n, i)\, a^{n-1}\, b^n + \text{-----} + C(n. n)b^n$.

The binomial coefficient has several properties are.

The three important properties are

C(n, k)=C(n-1,k-1)+C(n-1,k), for N>k>0

C(n,0)= 1

C(n, n)=1

47.What is best algorithm suited to identity the topology for a graph.mention its efficiency factors.

An alternative algorithm for topological sorting is based on depth-first search. The algorithm loops through each node of the graph, in an arbitrary order, initiating a depth-first search that terminates when it hits any node that has already been visited since the beginning of the topological sort or the node has no outgoing edges (i.e. a leaf node):

The usual algorithms for topological sorting have running time linear in the number of nodes plus the number of edges, asymptotically, O(|V|+|E|)

48.Define multistage graphs. Give an example. NOV-2018

The multistage graph problem is to find a minimum cost path from S to t.
Problem description:
A multistage graph G=(V,E) is a directed graph in which the vertices are portioned into K> 2 disjoint sets Vi, 1<i<=K.
if (u,v) is an edge in E, then u E Vi and VEVi+1 for some i., 1< = i< =K.
The sets V1 and Vk are such that (V1)= VK/=1, Let S and t respectively the vertex in b1 and bk.
The vertex S is the source, and t is the sin R. Let C (i, j) be the cost of edge (i,j)
The cost of a path from S to t is the sun of the cost of edges on the path.
Each set vi defines a stage in the graph Because of the constraints on E.
Every path from S to t starts in stage 1, goes to stage 2, then to stage 3, then to stage 4, etc., and finally terminates in stage K.

49. How dynamic programming is used to solve Knapsack problem? NOV-2018

Dynamic programming is a method for solving Optimization problems.
The idea:
Compute the solutions to the sub sub-problems once and store the solutions in a table, so that they can be reused (repeatedly) later

- Structure
- Principle of Optimality
- Bottom-up computation
- Construction of optimal solution

50. Define transitive closure of a directed graph. APR-2018

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

51.What is the constraint of for binary search tree insertion? Aprl/May 2019

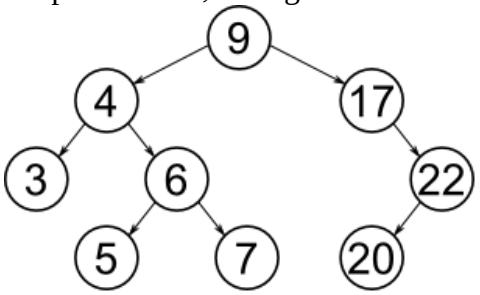
A binary search tree is a tree with one additional constraint — it keeps the elements in the tree in a particular order. Formally each node in the BST has two children (if any are missing we consider it a nil node), a left child and a right child.

52.Define Brute Force. Or What is brute force method Nov/Dec 2019

Brute Force is a straightforward approach to solve a problem, which is directly based on the problem statement and definition of the concepts. Brute Force strategy is one of the easiest approach.

53.Define a binary search tree Nov/Dec 2019

A binary search tree (BST), also known as an ordered binary tree, is a node-based data structure in which each node has no more than two child nodes. Each child must either be a leaf node or the root of another binary search tree. The left sub-tree contains only nodes with keys less than the parent node; the right sub-tree contains only nodes with keys greater than the parent node.



The BST data structure is the basis for a number of highly efficient sorting and searching algorithms, and it can be used to construct more abstract data structures including sets, multisets, and associative arrays.

54. **What is meant by optimal substructure property of a dynamic programming problem. (April/May 2021)**

In computer science, a problem is said to have optimal substructure **if an optimal solution can be constructed from optimal solutions of its subproblems**. This property is used to determine the usefulness of dynamic programming and greedy algorithms for a problem.

55. **Write the control abstraction for greedy method. (April/May 2021)**

1. A selection of solution from the given input domain is performed, i.e. s:= select(a). 2. The feasibility of the solution is performed, by **using feasible '(solution, s)'** and then all feasible solutions are obtained.

PART B

1. Define dynamic programming and explain the problems that can be solved using dynamic Programming.

Synopsis:

Introduction
Problems that can be solved using dynamic programming
Principle of optimality
Computing a Binomial Coefficient
Example

Introduction:

Dynamic Programming is an algorithm design technique.

It was invented by a U.S. Mathematican Richard Bellman in the year 1950, as a general method for optimizing multistage decision processes.

Dynamic Programming is a technique for solving problems with overlapping sub problems. The smaller sub problems are solved only once and recording the results in a table from which the solution to the original problem is obtained.

Problems that can be solved using dynamic programming:

Various problems those can be solved using dynamic programming are

For computing n^{th} Fibonacci number

1. Computing binomial coefficient
2. Warshall's algorithm
3. Floyd's algorithm
4. Optimal binary search trees

Principle of optimality:

The dynamic programming makes use of principle of optimality when finding solution to given problem.

The principle of optimality states that " in an optimal sequence of choices or decisions , each subsequence must also be optimal".

When it is not possible to apply principle of optimality, it is almost impossible to obtain the solution using dynamic programming approach.

Example:

While constructing optimal binary search tree we always select the value of k which is obtained from minimum cost. Thus it follows principle of optimality

Computing a Binomial Coefficient:

Computing a Binomial Coefficient is a typical example of applying dynamic programming in mathematics, particularly in combinatory.

Binomial Coefficient is a Coefficient of any of the term in the expansion of $(a+b)^{n.}$

The binomial coefficient is denoted by C(n, k) or $\binom{n}{k}$

The binomial coefficient is the number of combinations or subsets of K elements from an n element set($0 \leq k \geq n$).

The name binomial coefficient comes from the participation of these numbers in the binomial formula.

The binomial formula is

$(a+ b)^n = C(n.0)a^n + \text{----} + C(n, i)\, a^{n-1}\, b^n + \text{----} + C(n. n)b^{n.}$

The binomial coefficient has several properties are.

The three important properties are

$C(n, k)=C(n-1,k-1)+C(n-1,k)$, for $N>k>0$

$C(n,0)= 1$

$C(n, n)=1$

Example: Compute C(4,2)

Solution:

n=4, k=2

C(4,2) = C(n-1,k-1)+C(n-1,k)

C(4,2) =C(3,1) +C(3,2) --------(1)

As there are two unknowns : C(3,1) and C(3,2) in above equation we will compute these sub instance of C(4, 2)

Therefore n=3 , k=1

C(3,1) =C(2,0) +C(2,1)

C(n, 0) = 1 we can write

C(2, 0) = 1

C(3, 1) = 1 + C(2,1) -------(2)

Hence let us compute C (2, 1)

Therefore n=2 , k=1

C(2,1) = C(n-1,k-1)+ C(n-1,k)

C(2,1) =C(1,0) +C(1,1)

But as C (n, 0) = 1 and C(n, n) = 1 we get

C(1,0) = 1 and C(1,1) = 1

C(2,1) = C(1, 0) + C(1,1)

= 1 +1

C(2,1) = 2 ----------(3)

Put the equation (2) and we get

C(3,1) = 1 +2

C(3,1) = 3 ----------(4)

Now to solve equation 1 we will first compute C(3, 2) with n = 3 and k = 2.

Therefore

C(3,2) = C(n-1,k-1)+ C(n-1,k)

C(3,2) = C(2, 1) + C(2,2)

But as C(n,n) = C(2,2) = 1 , we will put values of C(2, 1) obtained in equation (3)

And C(2,2) in C(3,2) we get,

C(3,2) = C(2, 1) + C(2,2)

= 2+ 1

C(3,2) = 3 ----------(5)

Put equation (4) and (5) in equation , then we get

C(4,2) = C(3, 1) + C(3,2)

C(4,2) = 3 + 3

C(4,2) = 6 is the final answer

2. How dynamic programming approach is used in binomial coefficient?

The recurrence equation C(n,k)=C(n-1,k-1)+C(n-1,k) expresses the problem of computing C(n,k) in terms of C(n-1,k-1) and C(n-1,k) lends itself to solve by the dynamic programming technique. The values of the binomial coefficient are recorded in a table of n+1 rows and K+1 columns, numbered from 0 to n and from 0 to k respectively, which is shown in figure below

	0	1	2	3	4	5	…….	k-1	K
0	1								
1	1	1							
2	1	2	1						
3	1	3	3	1					
4	1	4	6	4	1				

5	1								
:									
K	1								1
:									
n-1	1							C(n-1,k-1)	C(n-1,k)
N	1								C(n,k)

To compute the value of C(n,k), the table of figure is filled by row , starting with row 0 and ending with row n.
Each row i($0 \leq i \leq n$) is filled left to right, starting with 1 because C(n,0)=1.
Rows 0 through k also end with 1 on the table's main diagonal (ie)

C(i,i)=1 for $0 \leq i \leq k$

The other entries of the table is computed by using the formula C(n,k)=C(n-1,k-1)+C(n-1,k), for n>k>0 adding the contents of the cells in the preceding row and the previous column in the preceding row and the same column.

Algorithm

Algorithm Binomial(n,k)
//Computes C(n,k) by the dynamic programming algorithm
// Input: A pair of nonnegative integers $n \geq k \geq 0$.
//Output: The value of C(n,k)
for i←0 to n do
  for j ←0 to min(i,k) do
    if j=0 or j=k
      C[i,j] ← 1
    else
      C[i,j] ← C[i-1,j-1]+ C[i-1,j]
  return C[n,k]

Analysis:

The basic operation is addition i.e.

C[i,j] ← C[i – 1, j-1] +C[i- 1, j]

Let A(n,k) denotes total additions made in computing C(n,k).

In the table, first K+1 rows of the table form a triangle and the remaining n-k rows form a rectangle.

So the recurrence relation for A(n,k) is divided into the parts.

The recurrence relation is,

$$A(n,k) = \sum_{i=1}^{k} \sum_{j=1}^{i-1} 1 + \sum_{i=k+1}^{n} \sum_{j=1}^{k} 1$$

$$A(n,k) = \sum_{i=1}^{k} [(i-1) - 1 + 1] + \sum_{i=k+1}^{n} (k - 1 + 1)$$

Therefore $\sum_{i=1}^{nk} 1 = (n - 1 + 1)$

$$= \sum_{i=1}^{k}[(i-1) + \sum_{i=k+1}^{n} k$$

$$= [\,1 + 2 + 3 + \ldots.(k-1)] + k \sum_{i=k+1}^{n} 1$$

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

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

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

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

$$= k^2/2 - k/2 + nk - k^2$$

$$= \Theta(nk)$$

$A(n,k) \in \theta(nk)$

As no edge from a to d , value is 0

Hence time complexity of binomial coefficient is **Θ(nk)**

3. Explain the Warshall's Algorithm.

Warshall's algorithm constructs the transitive closure of given diagraph with n vertices through a series of n × n boolean matrices.

The computations in **Warshall's algorithm** are given by following sequence,

R(0), . . . , R(k−1), R(k), . . . R(n).

Thus the idea in Warshall's algorithm is **building of boolean matrices.**

Digraph : the graph in which all the edges are directed then it is called digraph or directed graph .

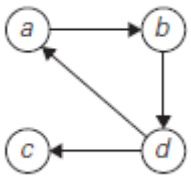
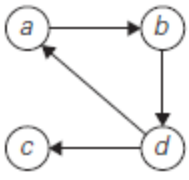


Fig: Digraph

Adjacency matrix : it is a representation of a graph by using matxi. If there exists an edge between the vertices V_i and v_j directing from v_i to v_j then entry in adjacency matrix in ith row and jth colums is 1



$A =$

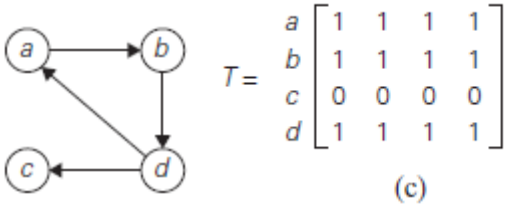
	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

Edge from b to d

(b)

adjacency matrix.

Transitive closure : Transitive closure is basically a boolean matrix (matrix with 0 and 1 values) in which the existence of directed paths of arbitrary lengths between vertices is mentioned.



(c)

transitive closure.

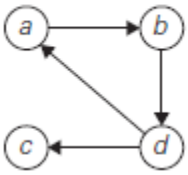
The transitive closure can be generated with Depth First Search (DFS) or with Breadh First Search (BFS).

This traversing can be done on any vertex.

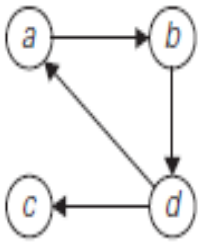
While computing transitive closure we have to start with some vertex and have to find all the edges which are reachable to every other vertex . the reachable edges for all the vertices has to obtained

Procedure to be followed :

- Start with computation of $R^{(0)}$. In $R^{(0)}$ any path with intermediate vertices is not allowed . the means only direct edges towards the vertices are considered. in other words the path length of one edge is allowed in $R^{(0)}$. Thus V $R^{(0)}$ is adjacency matrix for the diagraph.
- Construct $R^{(1)}$ in which first vertex is used as intermediate vertex and a path length of two edges is allowed. Note that $R^{(1)}$ is build using $R^{(0)}$ which is already computed.
- Go on building $R^{(k)}$ by adding one intermediate vertex each time and with more path length . each $R^{(k)}$ has to be built from $R^{(k-1)}$.
- The last matrix in this series is $R^{(n)}$, in this $R^{(n)}$ all yhe n vertices are used as intermediate vertices . and the $R^{(n)}$ which is obtained is nothing but the transitive closure of given digraph.
- Let us understand this algorithm with some example
- Obtain the transitive closure for the following digraph using Warshall's algorithm.



- Let us first obtain adjacency matrix for given digraph . it is denoted by $R^{(0)}$.



$R^{(0)}$ =

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

1's reflect the existence of paths with no intermediate vertices ($R^{(0)}$ is just the adjacency matrix); boxed row and column are used for getting $R^{(1)}$.

$R^{(1)}$ =

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

1's reflect the existence of paths with intermediate vertices numbered not higher than 1, i.e., just vertex *a* (note a new path from *d* to *b*); boxed row and column are used for getting $R^{(2)}$.

$R^{(2)}$ =

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

1's reflect the existence of paths with intermediate vertices numbered not higher than 2, i.e., *a* and *b* (note two new paths); boxed row and column are used for getting $R^{(3)}$.

$R^{(3)}$ =

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

1's reflect the existence of paths with intermediate vertices numbered not higher than 3, i.e., *a*, *b*, and *c* (no new paths); boxed row and column are used for getting $R^{(4)}$.

$R^{(4)}$ =

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

1's reflect the existence of paths with intermediate vertices numbered not higher than 4, i.e., *a*, *b*, *c*, and *d* (note five new paths).

Algorithm :

Algorithm Warshall (matrix [1..n, 1..n]
//problem description : this algorithm is for computing
//transitive closure using warshall's algorithm
//input: the adjacency matrix given by matrix [1..n , 1..n]
//Output : the transitive closure of digraph

```
        R(0) ← Matrix  //  initially adjacency matrix of
                              //diagraph becomes R(0)
    for (k ← 1 to n ) do
      {

      for (i ← 1 to n ) do
            {
              for (j ← 1 to n ) do
```

```
                    {
                        R(k) [ i, j]  ← R(k-1) [ i, j]   OR  R(k-1) [ i, k]
                    AND R(k-1) [ k, j]
                    }
                }
            }
            return R(n)
```

Analysis:

Clearly time complexity of above algorithm is $\Theta(n^3)$ because in above algorithm the basic operation is computation of $R^{(k)}[i, j]$.
This operation is located within three nested for loops.

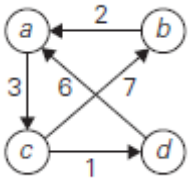
The time complexity warshall 's algorithm is $\Theta(n^3)$

4. Explain the Floyd's algorithm. Or
(i) write the Floyd algorithm to find all pair shortest path and derive its time complexity(4+3)
(ii)solve the following using floyd's algorithm.(6) Apr/May 2019

V	A	B	C	d
A	0	∞	3	∞
B	2	0	∞	∞
C	∞	7	0	1
D	6	∞	∞	0

Floyd's algorithm is used for finding the shortest path between every pair of vertices of a graph. It is all pairs shortest path algorithm.
The algorithm works for both directed and undirected graphs. This algorithm is invented by R. Floyd hence is the name.

Weighted graph: the weighted graph is a graph in which weights or distances are given along the edges. The weighted graph can be represented by weighted matrix as follows,



Here

w[i][j] = 0 if i=j
W[i][j] =∞ if there is no edge (directed edge) between i and j .
W[i][j] = weight of edge.

Formulation:

Let , D^k [i,j] denotes the weight of shortest path from v_i to v_j using $\{v_1, v_2, v_3 \ldots v_k\}$ as intermediate vertices.
Initially $D^{(k)}$ is computed as weighted matrix
There exits two case –

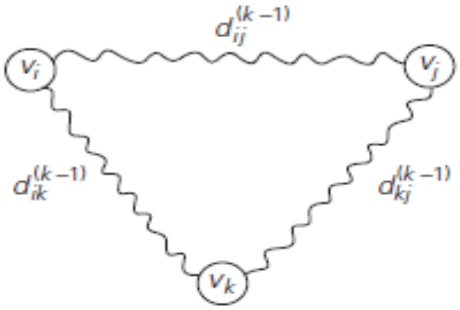
1. A shortest path from v_i to v_j with intermediate vertices from $\{v_1, v_2, v_3 \ldots v_k\}$ that does not use v_k. in this case

   $$\mathbf{D^k [i,j] = D^{(k-1)}[i,j]}$$

2. A shortest path from v_i to v_j restricted to using intermediate vertices $\{v_1, v_2, v_3 \ldots v_k\}$ which uses v_k. in this case-

   $$\mathbf{D^k [i,j] = D^{(k-1)} [i,k] + D^{(k-1)} [k,j]}$$

   The graphical representation of these two case is shortest path using vertices from $\{v_1, v_2, v_3 \ldots v_k\}$



$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} \quad \text{for } k \geq 1,\ d_{ij}^{(0)} = w_{ij}.$$

Basic concept of Floyd's algorithm:

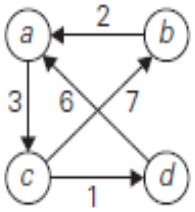
1. The Floyd's algorithm is for computing shortest path between every pair of vertices of graph.
2. The graph may contain negative edges but it should not contain negative cycles.
3. The Floyds algorithm requires a weighted graph.
4. Floyd's algorithm computes the distance matrix of a weighted graph with n vertices through a series of n × n matrices :

   D(0), . . . , D(k−1), D(k), . . . , D(n).

   In each matrix $D^{(k)}$ the shortest distance "d_{ij}" has to be computed between vertex v_i and v_j

5. In particular the series starts with $D^{(0)}$ with no intermediate vertex. That means $D^{(0)}$ is a matrix in which vi and v_j i.e.i^{th} row and j^{th} column contains the weights given by direct edges . in $D^{(1)}$ matrix – the shortest distance going through one intermediate vertex (starting vertex as intermediate) with maximum path length of 2 edges is given continuing in this fashion we will compute $D^{(n)}$, contains the lengths of shortest paths among all paths that can use all n vertices as intermediate. Thus we get all pair shortest paths from matrix $D^{(n)}$

Obtain the all pair – shortest path using Floyd's algorithm for the following weighted graph,



$D^{(0)}$ =

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

Lengths of the shortest paths with no intermediate vertices ($D^{(0)}$ is simply the weight matrix).

$D^{(1)}$ =

	a	b	c	d
a	0	∞	3	∞
b	2	0	**5**	∞
c	∞	7	0	1
d	6	∞	**9**	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 1, i.e., just *a* (note two new shortest paths from *b* to *c* and from *d* to *c*).

$D^{(2)}$ =

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	**9**	7	0	1
d	6	∞	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 2, i.e., a and *b* (note a new shortest path from *c* to *a*).

$D^{(3)}$ =

	a	b	c	d
a	0	**10**	3	**4**
b	2	0	5	**6**
c	9	7	0	1
d	6	**16**	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 3, i.e., a, *b,* and *c* (note four new shortest paths from *a* to *b,* from a to *d,* from *b* to *d,* and from *d* to *b*).

$D^{(4)}$ =

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	**7**	7	0	1
d	6	16	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 4, i.e., a, *b, c,* and *d* (note a new shortest path from *c* to *a*).

Algorithm:

ALGORITHM *Floyd(W*[1..*n*, 1..*n*]*)*
//Implements Floyd's algorithm for the all-pairs shortest-paths //problem
//Input: The weight matrix *W* of a graph with no negative-length //cycle
//Output: The distance matrix of the shortest paths' lengths

```
D ←W //is not necessary if W can be overwritten
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 :

In the above given algorithm the basic operation is –

$D[i, j]\leftarrow\min\{D[i, j], D[i, k]+ D[k, j]\}$

This operation is within three nested for loops , we can write

$C(n) = \sum_{k=1}^{n} \sum_{i=1}^{n} \sum_{j=1}^{n} 1$

$C(n) = \sum_{k=1}^{n} \sum_{i=1}^{n} (n - 1 + 1)$ **t**herefore $\sum_{i=1}^{n} 1 = u - 1 + 1$

$C(n) = \sum_{k=1}^{n} \sum_{i=1}^{n} n$

$C(n) = \sum_{k=1}^{n} n^2$

$C(n) = n^3$

The time complexity of finding all pair shortest path is **$\Theta(n^3)$**

5. Write a pseudo code to find Optimal binary search trees using dynamic programming (OBST) *May 2008/2011 & Dec 2013 may 2015*

Or

Obtain a optimal binary search tree for following nodes (do, if , int , while) with following probabilities (0.1, 0.2 , 0.4, 0.3)

Or

(i) outline dynamic programming approach to solve the optimal binary search tree problem and analyse its time complexity

(ii)construct the optimal binary search tree for the following 5 keys with probabilities as indicated. **Nov/Dec 2019**

i	0	1	2	3	4	5
Pi		0.15	0.10	0.05	0.10	0.20
pj	0.05	0.10	0.05	0.05	0.05	0.10

- A binary search tree is one of the most important data structures in computer science.
- One of its principal application is to implement a dictionary, a set of elements with the operations of searching, insertion, and deletion.

Example:

Consider four keys *A*, *B*, *C*, and *D* to be searched for with probabilities 0.1, 0.2, 0.4, and 0.3, respectively.

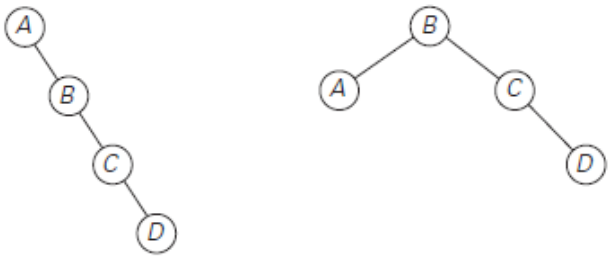


Figure depicts two out of 14 possible binary search trees containing these keys.

The average number of comparisons in a successful search in the first of these trees is

0. 1+ 0.2 . 2 + 0.4 . 3+ 0.3 . 4 = 2.9,

for the second one it is

0. 2 + 0.2 . 1+ 0.4 . 2 + 0.3 . 3= 2.1.

Neither of these two trees is, in fact, optimal.

For our tiny example, we could find the optimal tree by generating all 14 binary search trees with these keys.

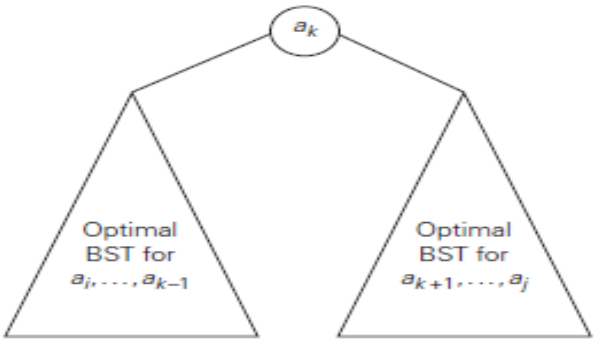
As a general algorithm, this exhaustive-search approach is unrealistic: the total number of binary search trees with *n* keys is equal to the *n*th ***Catalan number***,

$$c(n) = \frac{1}{n+1}\binom{2n}{n} \quad \text{for } n > 0, \quad c(0) = 1,$$

Optimal binary search tree

Definition

- Let {a1, a2,….an} be a set of identifiers such that a1<a2<a3…let , p(i) be the probability with which we can search for a_i is Successful search.
- Let q_i be the probability of searching an element x such that $a_i<x<a_i+1$ where $0\leq i\leq n$ is unsuccessful search.
- Thus p(i) is probability of successful search and q(i) is the probability of unsuccessful search.
- Then a tree which is build with optimum cost from $\sum_{i=1}^{n} p(i) \sum_{i=1}^{n} q(i)$ is called **optimal binary search tree.**



- For such a binary search tree , the root contains key a_k, the left subtree T_i^{k-1} contains keys $a_i, \ldots, a_{k-1}$ optimally arranged, and the right subtree T^j_{k+1} contains keys $a_{k+1}, \ldots, a_j$ also optimally arranged.
- If we count tree levels starting with 1 to make the comparison numbers equal the keys' levels, the following recurrence relation is obtained:

$$C(i, j) = \min_{i \le k \le j} \{p_k \cdot 1 + \sum_{s=i}^{k-1} p_s \cdot (\text{level of } a_s \text{ in } T_i^{k-1} + 1) + \sum_{s=k+1}^{j} p_s \cdot (\text{level of } a_s \text{ in } T_{k+1}^{j} + 1)\}$$

$$= \min_{i \le k \le j} \{\sum_{s=i}^{k-1} p_s \cdot \text{level of } a_s \text{ in } T_i^{k-1} + \sum_{s=k+1}^{j} p_s \cdot \text{level of } a_s \text{ in } T_{k+1}^{j} + \sum_{s=i}^{j} p_s\}$$

$$= \min_{i \le k \le j} \{C(i, k-1) + C(k+1, j)\} + \sum_{s=i}^{j} p_s.$$

Thus, we have the recurrence

$$C(i, j) = \min_{i \le k \le j} \{C(i, k-1) + C(k+1, j)\} + \sum_{s=i}^{j} p_s \quad \text{for } 1 \le i \le j \le n.$$

Initially we assume that C[I ,i-1] = 0 for I ranging from 1 to n+1.
Then set C[I, i] = pi where $1\leq i\leq j\leq n$.

that means we have to two tables in optimal binary search tree.

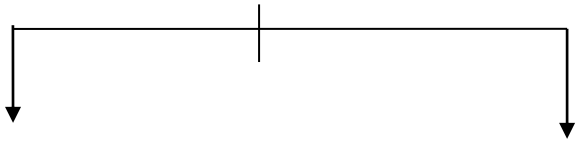


Table for cost .i.e ., cost table table for root i.e. root table

The cost table should be constructed as follows.

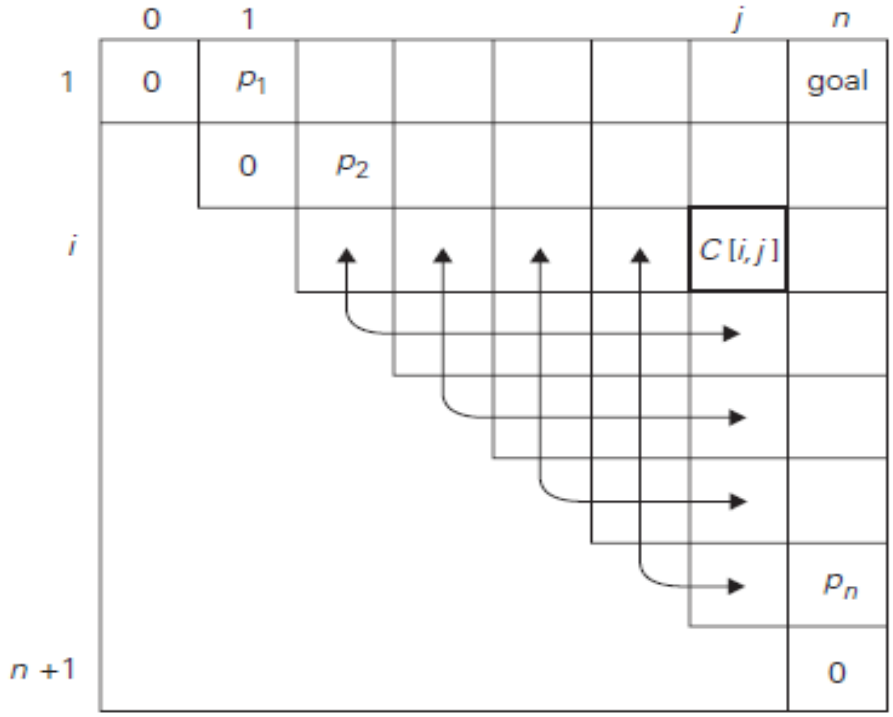


Fig :Table of the dynamic programming algorithm for constructing an optimal binary search tree.

1. Fill up C[i ,i-1] by 0 and C[n+1, n] by 0
2. Fill up C[i ,i] by p[i]
3. Fill up C[i ,j] using formula

EXAMPLE

Obtain a optimal binary search tree for following nodes (do, if , int ,while) with following probabilities (0.1, 0.2 , 0.4, 0.3)

Solution : There are 4 nodes.
Hence n = **4**

a[i]	1	2	3	4
p[i]	0.1	0.2	0.4	0.3

Initial Tables :

Cost Table : 0 1 2 3 4

1	0	0.1			
2		0	0.2		
3			0	0.4	
4				0	0.3
5					0

Root Table

	0	1	2	3
1		1			
2			2		
3				3	
4					4

Cost Table

C[1, 0] = 0
C[2, 1] = 0
C[3, 2] = 0
C[4, 3] = 0
C[5, 4] = 0

using formulae C[i, i – 1] =0 and C[n + 1,n] =0

C[1, 1] = 0.1
C[2, 2] = 0.2
C[3, 3] = 0.4
C[4, 4] = 0.3

using formulae C[i, i] =p[i]

The root table

R[1, 1] =1
R[2, 2] =2
R[3, 3] =3

using formulae R[i, i] =i

R[4, 4] =4

Now let us compute C[i, j] diagonally using formula

-

$$C(i, j) = \min_{i \le k \le j} \{C(i, k-1) + C(k+1, j)\} + \sum_{s=i}^{j} p_s \quad \text{-- (1)}$$

Compute C[1, 2]

The value of k can be either 1 or 2

Let i = 1 , j = 2 , use formula in equation (1),

k= 1

C[1,2]= C[1,0] + C[2 ,2] + p[1] +p[2]

= 0 + 0.2 + 0.1 + 0.2

= 0.5

k=2

C[1,2]=C[1,1] + C[3 ,2] + p[1] +p[2]

= 0.1 + 0 + 0.1 + 0.2

= **0.4**-> minimum value therefore consider k = 2

Therefore in cost table **C[1,2] = 0.4** and **R[1,2]=2**

Compute C[2, 3]

The value of k can be 2 or 3.

Let i = 2, j = 3, use formula equation (1)

k = 2

C[2,3] =C[2,1] + C[3 ,3] + p[2] +p[3]

=0 + 0.4+ 0.2+ 0.4

=1.0

k=3

C[2,3]= C[2,2] + C[4 ,3] + p[2] +p[3]

= 0.2 + 0 + 0.2 + 0.4

= **0.8** -> minimum value therefore consider k = 3

Therefore in cost table C[2,3] = 0.8 and R[2,3]=3

Compute C[3, 4]

The value of k can be 3 or 4.

Let , i = 3, j = 4, use formula equation (1)

k = 3

C[3,4]= C[3,2] + C[4 ,4] + p[3] +p[4]

= 0 + 0.3+ 0.4+ 0.3

= **1.0** -> minimum value therefore consider k = 3

k=4

C[3,4]= C[3,3] + C[5 ,4] + p[3] +p[4]

= 0.4 + 0 + 0.4 + 0.3= 1.1

Therefore in cost table **C[3,4] = 1.0** and **R[3,4]=3**

The table contains values obtained upto this calculations

Cost Table

0	1	2	3	4
0	0.1	**0.4**		

1	0	0.2	**0.8**	
2		0	0.4	**1.0**
3			0	0.3
4				0
5				

Root Table

	0	1	2	3
1		1	**2**	
2			2	**3**
3				3 **3**
4				4

Compute C[1,3]

The value of k can be 1,2 or 3

Consider i=1 , j=3.

k=1

C[1,3] = C[1,0] + C[2 ,3] + p[1] +p[2] + p[3]

= 0 + 0.8 + 0.1 + 0.2 + 0.4

= 1.5

k =2

C[1,3]= C[1,1] + C[3 ,3] + p[1] +p[2] + p[3]

= 0.1 + 0.4 + 0.1 + 0.2 + 0.4
= 1.2

k =3

C[1,3]= C[1,2] + C[4 ,3] + p[1] +p[2] + p[3]
= 0 .4+ 0 + 0.1 + 0.2 + 0.4
= **1.1**-> minimum value therefore consider k = 3

Therefore **C[1,3] = 1.1** and **R[1,3] = 3**

Compute C [2,4]

The value of k can be 2,3 or 4

Consider i=2, j=4 and using equation (1)

k=2

C[2,4]= C[2,1] + C[3 ,4] + p[2] +p[3] + p[4]
= 0 + 1.0 + 0.2 + 0.4 + 0.3
= 1.9

k =3

C[2,4]= C[2,2] + C[4 ,4] + p[2] +p[3] + p[4]
= 0.2 + 0.3 + 0.2 + 0.4 + 0.3
= **1.4** -> minimum value therefore consider k = 3

k =4

C[2,4]= C[2,3] + C[5 ,4] + p[2] +p[3] + p[4]
= 0 .8+ 0 + 0.2 + 0.4 + 0.3
= 1.7

Therefore **C[2,4]= 1.4** and **R[2,4]= 3**

Cost Table

	0	1	2	3	4
1	0	0.1	**0.4**	**1.1**	
2		0	0.2	**0.8**	**1.4**
3			0	0.4	**1.0**
4				0	0.3
5					0

Root Table

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

1	2	**3**	**3**
2		3	**3**
3			4
4			

Compute C[1,4]

The value of k can be 1, 2, 3 or 4.

Consider i=1, j=4 and using equation (1)

k=1

C[1,4] = C[1,0] + C[2 ,4] + p[1] + p[2] +p[3] + p[4]
= 0 + 1.4 + 0.1+ 0.2 + 0.4 +0.3
= 2.4

k =2

C[1,4]= C[1,1] + C[3 ,4] + p[1] + p[2] +p[3] + p[4]
= 0.1 + 1.0 + 0.1 + 0.2 + 0.4 +0.3
= 2.1

k =3

C[1,4]= C[1,2] + C[4 ,4] + p[1] + p[2] +p[3] + p[4]
= 0.4 + 0.3 + 0.1 + 0.2 + 0.4 +0.3
= **1.7**-> minimum value therefore consider k = 3

k=4

C[1,4]= C[1,3 + C[5 ,4] + p[1] + p[2] +p[3] + p[4]
= 1.3 + 0 + 0.1 + 0.2 + 0.4 +0.3
= 2.3

Therefore **C[1,4]= 1.7** and **R[1,4]= 3**

Cost Table

0	1	2	3	4
0	0.1	**0.4**	**1.1**	**1.7**
	0	0.2	**0.8**	**1.4**
		0	0.4	**1.0**
			0	0.3

1	0
2	
3	
4	
5	

Root Table

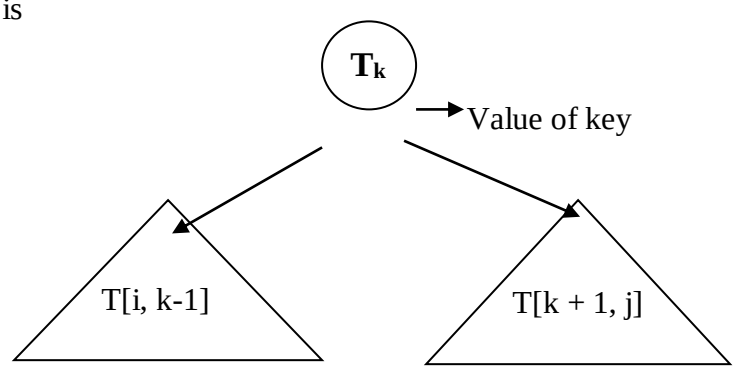
	0	1	2	3
1	1	**2**	**3**	**3**
2		2	**3**	**3**
3			3	**3**
4				4

To build a tree R[1][n] = R[1][4] =3 becomes root.

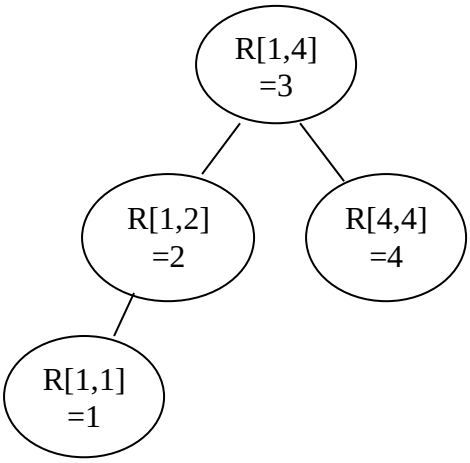
A[i]	1	2	3	4
	Do	If	Int	while

Key = 3 means int

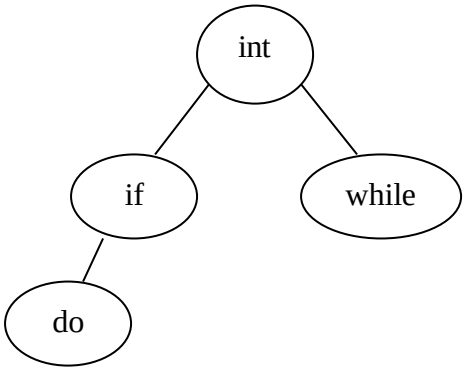
Therefore "int" becomes root of optimal binary search tree.
The tree is



Here i =1 , j=4 and k=3.



The tree can be with optimum cost C[1,4] = 1.7



Optimal binary search tree

ALGORITHM

ALGORITHM OptimalBST(P [1..n])
//Finds an optimal binary search tree by dynamic //programming
//Input: An array P[1..n] of search probabilities for a sorted //list of n keys
//Output: Average number of comparisons in successful //searches in the
//optimal BST and table R of sub trees' roots in the optimal //BST

```
for i ←1 to n do
{
        C[i, i − 1]←0
        C[i, i]←P[i]
        R[i, i]←I
}
C[n + 1, n]←0
for d ←1 to n − 1 do //diagonal count
{
        for i ←1 to n − d do
        {
                j ←i + d
                minval←∞
                for k←i to j do
                {
```

```
                    if C[i, k − 1]+ C[k + 1, j]< minval
                    {
                    minval←C[i, k − 1]+ C[k + 1, j];
                    kmin←k
                    R[i, j ]←kmin
                    }
            }
                    sum←P[i];
                    for s ←i + 1 to j do
                            sum←sum + P[s]
                            C[i, j ]←minval + sum
                    return C[1, n], R
            }
    }
```

Analysis:

The basic operation in above algorithm is computation of C[i,j] by finding the minimum valued k.

This operation is located within three nested for loops hence the time complexity C(n) can be

$$C(n) = \sum_{m=1}^{n-1} \sum_{i=1}^{n-m} \sum_{k=i}^{jn} 1$$

$$= n^3$$

$$C(n) = \Theta(n^{3})$$

Hence the time complexity of optimal binary search algorithm is **C(n) = Θ (n^3)**

6. Write the algorithm to compute the 0/1 knapsack problem using dynamic programming and explain it. *Dec 2010, Apr/May -2017* **or write a greedy algorithm to solve the 0/1 knapsack problem.Analyse its time complexity.Show that this algorithm is not optimal with an example.(5+2+6) Nov/Dec 2019**

Given n items of known weights W_1….. W_n and values a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack.

Assume that the weights and the knapsack capacity are positive integers.

The item values do not have to be integers.

To design a dynamic programming algorithm, it is necessary to derive a recurrence relation that expresses a solution to an instance of the knapsack problem in terms of solutions to its smaller sub instances.

The recurrence relation is, equation

$$F(i, j) = \begin{cases} \max\{F(i-1, j), v_i + F(i-1, j-w_i)\} & \text{if } j - w_i \geq 0, \\ F(i-1, j) & \text{if } j - w_i < 0. \end{cases}$$

The initial conditions are equation (2)

$F(0, j) = 0$ for $j \geq 0$ and $F(i, 0) = 0$ for $i \geq 0$.

Our **goal** is to find F[n,w], the maximum value of a subset of the n items that fit into the knapsack of capacity w, and an optimal subset itself.

F[0,j] = 0 as well as F[i,0] = 0 when j≥0 and i≥0.

The table can be filled by either row by row or column by column.

	0	j- W_i	j	W
0	0	0		0
i-1	0	F[i-1,j- W_i]	F[i-1,j]	

i	0		F[i,j]	
n	0			**Goal**

W_i V_i

Example

For the given instance of problem obtain the optimal solution for the knapsack problem

Item	**Weight**	**Value**
1	2	$3
2	3	$4
3	4	$5
4	5	$6

The capacity of knapsack is **W=5**

Solution :

Initially , F[0,j] = 0 and F[I,0] = 0.

There are 0 to n rows and 0 to W columns in the table.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

Let us start filling the table row by row using following formula:

$$F(i, j) = \begin{cases} \max\{F(i-1, j), v_i + F(i-1, j-w_i)\} & \text{if } j - w_i \geq 0, \\ F(i-1, j) & \text{if } j - w_i < 0. \end{cases}$$

Compute F[1 , 1] with i = 1 , j= 1 , w_i = 2 and v_i =3

As j< w_i we will obtain F[1,1] as

F[1,1] = F[i – 1, j]

= F[0, 1]

F[1,1] = **0**

Compute F[1 , 2] with i = 1 , j= 2 , w_i = 2 and v_i =3

As j≥ w_i we will obtain F[1,2] as

F[1,2]= Max{ F[i -1,j],v_i + F [i-1, j-w_i]}

= Max {F[0,2]) , (3 + F[0,0])}

= Max{ 0, 3 + 0}

F[1,2] = **3**

Compute F[1 , 3] with i = 1 , j= 3 , w_i = 2 and v_i =3

As j≥ w_i we will obtain F[1,3] as

F[1,3]= Max{ F[i -1,j],v_i + F [i-1, j-w_i]}

= Max {F[0,3]) , (3 + F[0,1])}

= Max{0, 3 + 0}

F[1,3] = **3**

Compute F[1 , 4] with i = 1 , j= 4 , w_i = 2 and v_i =3

As j≥ w_i we will obtain F[1,4] as

F[1,4] = Max{ F[i -1,j],v_i + F [i-1, j-w_i]}

= Max {F[0,4]) , (3 + F[0,2])}

= Max{0, 3 + 0}

F[1,4]= **3**

Compute F [1 , 5] with i = 1 , j= 5, w_i = 2 and v_i =3

As j≥ w_i we will obtain F[1,5] as

F[1,5]= Max{ F[i -1,j],v_i + F [i-1, j-w_i]}

= Max {F[0,5]) , (3 + F[0,3])}

= Max{0, 3 + 0}

F[1, 5]= **3**

The table with these values can be

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0					
3	0					
4	0					

Compute F[2 , 1] with i = 2 , j= 1, w_i = 3 and v_i =4

As j≤ w_i we will obtain F[2,1] as

F[2,1] = F[i – 1, j] = F[1, 1]

F[2,1] = **0**

Compute F [2 , 2] with i = 2 , j= 2, w_i = 3 and v_i =4

As j≤ w_i we will obtain F[2,2] as

F[2,2] = F[i – 1, j]

= F[1, 2]

F[2,2] = **3**

Compute F [2 , 3] with i = 2 , j= 3, w_i = 3 and v_i =4

As $j \geq w_i$ we will obtain F[2,3] as

$F[2,3] = \max\{F[i-1,j], v_i + F[i-1, j-w_i]\}$

$= \text{Max}\{F[1,3]), (4 + F[1,0])\}$

$= \text{Max}\{3, 4+0\}$

$F[2,3] = \mathbf{4}$

Compute F [2 , 4] with $i = 2$, $j = 4$, $w_i = 3$ and $v_i = 4$

As $j \geq w_i$, we will obtain F[2,4] as

$F[2,4] = \max\{F[i-1,j], v_i + F[i-1, j-w_i]\}$

$= \text{Max}\{F[1,4]), (4 + F[1,1])\}$

$= \text{Max}\{3, 4+0\}$

$F[2,4] = \mathbf{4}$

Compute F [2 , 5] with $i = 2$, $j = 5$, $w_i = 3$ and $v_i = 4$

As $j \geq w_i$, we will obtain F[2,5] as

$F[2,5] = \max\{F[i-1,j], v_i + F[i-1, j-w_i]\}$

$= \text{Max}\{F[1,5]), (4 + F[1,2])\}$

$= \text{Max}\{3, 4+3\}$

$F[2,5] = \mathbf{7}$

The table with these values can be

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0					
4	0					

Compute F [3 , 1] with $i = 3$, $j = 1$, $w_i = 4$ and $v_i = 5$

As $j < w_i$ we will obtain F[3,1] as

$F[3,1] = F[i-1, j]$

$= F[2, 1]$

$F[3,1] = \mathbf{0}$

Compute F [3 , 2] with $i = 3$, $j = 2$, $w_i = 4$ and $v_i = 5$

As $j < w_i$ we will obtain F[3,2] as

$F[3,2] = F[i-1, j]$

$= F[2, 2]$

$F[3,2] = \mathbf{3}$

Compute F [3 , 3] with $i = 3$, $j = 3$, $w_i = 4$ and $v_i = 5$

As $j < w_i$ we will obtain F[3,3] as

$F[3,3] = F[i-1, j]$

$= F[2, 3]$

F[3,3] = **4**

Compute F [3 , 4] with i = 3 , j=4, w_i = 4 and v_i =5

As j<w_i we will obtain F[3,4] as

F [3,4]= max{ F[i -1,j],v_i + F [i-1, j-w_i]}

= Max { F[2,4]) , (5 + F[2,0])}

= Max{ 4, 5+ 0}

F[3,4] = **5**

Compute F [3 , 5] with i = 3 , j=5 w_i = 4 and v_i =5

As j<w_i we will obtain F[3,5] as

F [3,5]= max{ F[i -1,j],v_i + F [i-1, j-w_i]}

= Max { F[2,5]) , (5 + F[2,1])}

= Max{ 7, 5+ 0}

F[3,5] = **7**

The table with these values can be

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0					

Compute F [4 , 1] with i = 4 , j=1, w_i = 5 and v_i =6

As j<w_i we will obtain F[4,1] as

F[4,1] = F[i – 1, j]

= F[3, 1]

F[4,1]= **0**

Compute F [4 , 2] with i = 4 , j=2, w_i = 5 and v_i =6

As j<w_i we will obtain F[4,2] as

F[4,2] = F[i – 1, j]

= F[3, 2]

F[4,2]= **3**

Compute F [4 , 3] with i = 4 , j=3 w_i = 5 and v_i =6

As j<w_i we will obtain F[4,3] as

F[4,3] = F[i – 1, j]

= F[3, 3]

F[4,3]= **4**

Compute F [4 , 4] with i = 4 , j=4 w_i = 5 and v_i =6

As j<w_i we will obtain F[4,4] as

F[4,4] = F[i – 1, j]

= F[3, 4]

F[4,4] = **5**

Compute F [4 , 5] with i = 4 , j=5 w_i = 5 and v_i =6

As $j<w_i$ we will obtain F[4,5] as

$F[4,5]= Max\{ F[i-1,j], v_i + F[i-1, j-w_i]\}$

$= Max\{ F[3,5]) , (6 + F[3,0])\}$

$= Max\{ 7, 6+ 0\}$

F[4,5]=**7**

The table with these values can be

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

Identification of Knapsack Items:

F[n, W] is the total value of selected items that can be placed in the knapsack. Following steps are used repeatedly.

```
Let i=n and k = W then
While ( i>0 and k>0)
{
        If(F[i,k] ≠ F[i-1,k]) then
            Mark ith item as in knapsack
            i= i-1 and k=k-wi        // selection of ith item
        else
            i= i-1 // do not select ith item
}
```

Let us apply these steps to the problem we have in final table.

0	1	2	3	4	5
0	0	0	0	0	0
0	0	3	3	3	3
0	0	3	4	4	7
0	0	3	4	5	7

0	0	0	3	4	5	7
1						
2						
3						
4						

Start : Let **i = 4** and k = 5

Capacity ->0 1 2 3 4 5

0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

As **F[i,k] = F[i – 1, k]**

F[4,5] = F[3,5] , Don't select i th item.

Now set i = i - 1, **i = 3**

Capacity -> 0 1 2 3 4 5

0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

F[3,5] = F[2,5] Don't select i th item i.e ., 3 rd item . now set i = i-1, **i=2**

Capacity -> 0 1 2 3 4 5

0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

F [i,k]≠ F[i – 1, k] F[2,5] ≠ F[1,5] Select i^{th} item, ie , select 2^{nd} item.Set i = i -1 and k = k- wi i.e . i = i -1, **i=1** and k = 5 - 3 =2

Capacity -> 0 1 2 3 4 5

0	0	0	0	0	0	0
1	0	0	3	3	3	**3**
2	0	0	3	4	4	**7**
3	0	0	3	4	5	7
4	0	0	3	4	5	7

As F [i,k]≠ F[i – 1, k]

i.e F[1,2] ≠ F[0,2]

select i^{th} item

That is , select 1^{st} item.

Set i = i - 1 and k = k – wi

i = 0 and k=2-2=0

Thus **item 1** and **item 2** are selected for the knapsack.

Algorithm

Algorithm Dynamic knapsack (n,, W,w[],v[])

// problem description : this algorithm is for obtaining knapsack

// solution using dynamic programming

//input : n is total number of items , W is the capacity of //knapsack, w [] stores weights of each item and v[] stores

//the values of each item

//output : returns the total value of selected items for yhe

// knapsack

For(i← 0 to n) do

{

for(i← 0 to W) do

{

```
                F[i,0] = 0 // table initialization
                F[0,j] = 0
        }
    }
    for(i← 0 to n ) do
    {
        for(i← 0 to W ) do
                {
                    If(j<w[i]) then
                    {
                        F[i,j] ←F[i – 1, j]
                        Else if (j>= w[i]) then
                        F[i,j] ←max(F[i -1,j],v[i] + F [i-1, j-w[i])
                    }
                }
    }
    Return F[n,W]
```

Analysis:

In this algorithm the basic operation is if ….else if statement within two nested for loops

Hence

$$C(n) = \sum_{i=0}^{n} \sum_{j=0}^{w} 1$$

$$C(n) = \sum_{i=0}^{n} W - 0 + 1$$

$$C(n) = \sum_{i=0}^{n} W + 1$$

$$C(n) = \sum_{i=0}^{n} W + \sum_{i=0}^{n} 1$$

$$C(n) = W. \sum_{i=0}^{n} 1 + \sum_{i=0}^{n} 1$$

$C(n) = W(n- 0 +) + (n-0+1)$

$C(n) = W_n + W + n + 1$

$C(n) = W_n$

The time complexity of this algorithm is **Θ(nW)**

7. Explain 0/1 knapsack Memory function in detail. Or Explain knapsack problem and memory function in detail. (APR/MAY 2018)

Memory Function:

- The memory function technique seeks to combine strengths of the top down and bottom up approaches to solving problems with overlapping subprograms of a given problem and recording their solutions in a table.

Memorization: is a way to deal with overlapping subproblems in dynamic programming .During memorization

1. After computing the solution to a subproblem , store it in a table
2. Make use of recursive calls.

The 0/1 knapsack memory function algorithm:

ALGORITHM MFKnapsack(i, j)

//Implements the memory function method for the knapsack //problem

//Input: A nonnegative integer i indicating the number of the //first items being considered and a nonnegative integer j //indicating the knapsack capacity

```
//Output: The value of an optimal feasible subset of the first i //items
//Note: Uses as global variables input arrays Weights[1..n],
// Values[1..n], and table F[0..n, 0..W ] whose entries are //initialized with −1's except for row 0 and column 0 initialized //with 0's
if F[i, j ]< 0
if j <Weights[i]
        value←MFKnapsack(i − 1, j)
else
        value←max(MFKnapsack(i − 1, j),
        Values[i]+ MFKnapsack(i − 1, j −Weights[i]))
        F[i, j ]←value
        return F[i, j ]
```

Example

Apply the dynamic programming following instance of the knapsack problems and solve. *Feb 2009*

The instance is

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

Capacity W=5

Solution :

Initially , F[0,j] = 0 and F[i,0] = 0.

There are 0 to n rows and 0 to W columns in the table.

	0	1	2	3	4	5
1	0	0	0	0	0	0
2	0					
3	0					
4	0					
5	0					

Now we will fill up the table either row by row or column by Colum. Let us start filling the table row by row using following formula:

$$F(i, j) = \begin{cases} \max\{F(i-1, j), v_i + F(i-1, j-w_i)\} & \text{if } j - w_i \geq 0, \\ F(i-1, j) & \text{if } j - w_i < 0. \end{cases}$$

Compute F [1 , 1] with i = 1 , j= 1 , w_i = 2 and v_i =12

As j< w_i we will obtain F[1,1] as

F[1,1] = F[i – 1, j]

=F[0, 1] = **0**

Compute F [1 , 2] with i = 1 , j= 2 , w_i = 2 and v_i =12

As j≥ w_i we will obtain F[1,2] as

F[1,2] = max{ F[i -1,j],v_i + F [i-1, j-w_i]}
= Max{0, 12 + 0}
F[1,2] = **12**

Compute F [1 , 3] with i = 1 , j= 3 , w_i = 2 and v_i =12
As j≥ w_i we will obtain F[1,3] as
F[1,3] = max{ F[i -1,j],v_i + F [i-1, j-w_i]}
= Max{ 0, 12 + 0}
F[1,3] = **12**

Compute F [1 , 4] with i = 1 , j= 4 , w_i = 2 and v_i =12
As j≥ w_i we will obtain F[1,4] as
F[1,4] = max{ F[i -1,j],v_i + F [i-1, j-w_i]}
= Max{ 0, 12 + 0}
F [1, 4] = **12**

Compute F [1 , 5] with i = 1 , j= 5, w_i = 2 and v_i =12
As j≥ w_i we will obtain F[1,5] as
F[1,5] = max{ F[i -1,j],v_i + F [i-1, j-w_i]}
= Max{ 0, 12 + 0}
F [1,5]= **12**

The table with these values can be

	0	1	2	3	4	5
1	0	0	0	0	0	0
2	0	0	12	12	12	12
3	0					
4	0					
5	0					

Compute F [2 , 1] with i = 2 , j= 1, w_i = 1 and v_i =10
As j≥ w_i we will obtain F[2,1] as
F[2,1] = max{ F[i -1,j],v_i + F [i-1, j-w_i]}
= Max{ 0, 10 + 0}
F[2,1]= **10**

Compute F [2 , 2] with i = 2 , j= 2, w_i = 1 and v_i =10
As j≥w_i we will obtain F[2,2] as
F[2,2] = max{ F[i -1,j],v_i + F [i-1, j-w_i]}
= Max { F[1,2) , (10 + F[1,1])}
= Max{ 12, 10 + 0}
F[2,2] = **12**

Compute F [2 , 3] with i = 2 , j= 3, w_i = 1and v_i =10
As j≥w_i we will obtain F[2,3] as
F [2 ,3] = max{ F[i -1,j],v_i + F [i-1, j-w_i]}
= Max { F[1,3) , (10 + F[1,2])}
= Max{ 12, 10+ 12}
F[2,3] = **22**

Compute F [2 , 4] with i = 2 , j= 4, w_i = 1 and v_i =10
As j≥w_i , we will obtain F[2,4] as
F [2,4]= max{ F[i -1,j],v_i + F [i-1, j-w_i]}
= Max { F[1,4]) , (10 + F[1,3])}

= Max{ 12, 10+ 12}

F[2,4] = **22**

Compute F [2 , 5] with i = 2 , j= 5, w_i = 1 and v_i =10

As $j \geq w_i$, we will obtain F[2,5] as

F [2,5] = max{ F[i -1,j],v_i + F [i-1, j-w_i]}

= Max { F[1,5]) , (10 + F[1,4])}

= Max{ 12, 10+ 12}

F[2,5] = **22**

The table with these values can be

	0	1	2	3	4	5
1	0	0	0	0	0	0
2	0	0	12	12	12	12
3	0	10	12	22	22	22
4	0					
5	0					

Compute F [3 , 1] with i = 3 , j= 1, w_i = 3 and v_i =20

As $j \geq w_i$ we will obtain F[3,1] as

F[3,1] = F[i – 1, j]

= F[2, 1]

F[3,1] = **10**

Compute F [3 , 2] with i = 3 , j=2, w_i = 3 and v_i =20

As $j < w_i$ we will obtain F[3,2] as

F[3,2] = F[i – 1, j]

= F[2, 2]

F[3,2] = **12**

Compute F [3 , 3] with i = 3 , j=3, w_i = 3 and v_i =20

As $j \geq w_i$ we will obtain F[3,3] as

F [3 , 3]= max{ F[i -1,j],v_i + F [i-1, j-w_i]}

= Max { F[2,3) , (20 + F[2,0])}

= Max{ 22, 20+ 0}

F[3,3] = **22**

Compute F [3 , 4] with i = 3 , j=4 w_i = 3 and v_i =20

As $j \geq w_i$ we will obtain F[3,4] as

F [3 , 4]= max{ F[i -1,j],v_i + F [i-1, j-w_i]}

= Max { F[2,4]) , (20 + F[2,1])}

= Max{ 22, 20+1 0}

F [3 , 4] = **30**

Compute F [3 , 5] with i = 3 , j=5, w_i = 3 and v_i =20

As $j \geq w_i$ we will obtain F[3,5] as

F [3,5] = max{ F[i -1,j],v_i + F [i-1, j-w_i]}

= Max { F[2,5]) , (20 + F[2,2])}

= Max{ 22, 20+ 12}

F [3 , 5] = **32**

The table with these computed values will be

	0	1	2	3	4	5
1	0	0	0	0	0	0
2	0	0	12	12	12	12
3	0	10	12	22	22	22
4	0	10	12	22	30	32
5	0					

Compute F [4 , 1] with i = 4 , j=1, w_i = 2 and v_i =15

As j<w_i we will obtain F[4,1] as

F[4,1] = F[i – 1, j]

= F[3, 1]

F[4,1] **=1 0**

Compute F [4 , 2] with i = 4 , j=2 w_i = 2 and v_i =15

As j≥w_i we will obtain F[4,2] as

F [4, 2]= max{ F[i -1,j],v_i + F [i-1, j-w_i]}

= Max { F[3,2]) , (15 + F[3,0])}

= Max{ 12, 15+ 0}

F[4,2] = **15**

Compute F [4 , 3] with i = 4 , j=3 w_i = 2 and v_i =15

As j≥w_i we will obtain F[4,3] as

F [4, 3]= max{ F[i -1,j],v_i + F [i-1, j-w_i]}

= Max { F[3,3]) , (15 + F[3,1])}

= Max{ 22, 15+1 0}

F [4,3] **= 25**

Compute F [4 , 4] with i = 4 , j=4 w_i = 2 and v_i =15

As j≥w_i we will obtain F[4,3] as

F [4, 4]= max{ F[i -1,j],v_i + F [i-1, j-w_i]}

= Max { F[3,4]) , (15 + F[3,2])}

= Max{ 30, 15+1 2}

F [4 , 4] **= 30**

Compute F [4 , 5] with i = 4 , j=5 w_i = w_i = 2 and v_i =15

As j<w_i we will obtain F[4,5] as

F [4 , 5] = max{ F[i -1,j],v_i + F [i-1, j-w_i]}

= Max { F[3,5]) , (15 + F[3,3])}

= Max{ 32,156+ 22}

F [4 , 5] **= 37**

The table with these computed values will be

0	1	2	3	4	5
0	0	0	0	0	0

1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37
5						

Therefore i = 4, k = 4

As F [i,k]= F[I – 1, k]

i.e F[4,5] = F[3,5]

Select i^{th} item i.e ., 4 th item .

Now set i = i-1 and k = k- wi

Therefore i=4-1= 3 and k = 5 – 2= 3

As F [i,k]= F[I – 1, k]

i.e F[3,3] = F[2,3]

do not Select i^{th} item i.e ., 3 rd item .

Now set i = i-1 and k = k- wi

Therefore i =3-1 = 2 and k = 5 – 2= 3

As F [i,k]≠ F[I – 1, k]

i.e F[2,3] ≠ F[1,3]

Select i^{th} item i.e ., 2 nd item .

Now set i = i-1 and k = k- wi

Therefore i =2-1 = 1 and k = 3 – 2= 2

As F [i,k]≠ F[I – 1, k]

i.e F[1,2] ≠ F[0,2]

Select i^{th} item i.e ., 1 st item .

Now set i = i-1 and k = k- wi

Therefore i =1-1 = 0 and k = 2 – 2= 0

Thus solution to this knapsack problem **is (item 1 , item 2 , item 4) with total profit = 37**

8. Explain in detail about Greedy Techniques:

The greedy method is a straightforward method.

This method is popular for obtaining the optimized solutions

In greedy technique, the solution is constructed through a sequence of steps , each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached .

- At each step the choice made should be, Feasible-It has to satisfy the problem's constraints.

  Locally optimal-It has to be the best local choice among all feasible choices available on that step.

  Irrevocable -Once made, it cannot be changed on subsequent steps of the algorithm.

General method

- The greedy method uses the subset paradigm or ordering paradigm to obtain the solution.
- In subset paradigm , at each stage the decision is made based on whether a particular input is in optimal solution or not .

For example

Knapsack problem

Applications of greedy method

1. Knapsack problem
2. Prim's algorithm for minimum spanning tree
3. Kruskal's algorithm for minimum spanning tree
4. Finding shortest path
5. Job sequence with deadlines
6. Optimal storage on tapes

For solving all above problems, a set of feasible solutions is obtained. From this solution, optimum solution is selected.

This optimum solution then becomes the final solution for given problem.

Divide and Conquer Vs greedy method:

Divide and Conquer	**Greedy method**
Divide and conquer is used to obtain a solution to given problem.	Greedy method is used to obtain optimum solution
In this technique, the problem is divided into small sub problems are solved independently. Finally all the solutions of sub problems are collected together to get the solution to the given problem	In greedy method a set of feasible solution is generated and optimum solution is picked up
In this method , duplications in sub solutions are neglected. The means duplicate solution may be obtained	In greedy method , the optimum selection is without revising previously generated solutions
Divide and conquer is less efficient because of rework on solutions	Greedy method is comparatively efficient but there is no as such guarantee of getting optimum solution.
Example : quick sort, binary search	Example : knapsack problem, finding mining spanning tree

Greedy method Vs Dynamic programming

Greedy Method	**Dynamic Programming**
Greedy method is used for obtaining **optimum solution**	Dynamic Programming is also for obtaining **optimum solution**
Greedy method a set of **feasible solutions** and the picks up the optimum solution	There is no special set of **feasible solutions** in this method

In Greedy method the optimum selection is without revising previously generated solutions	Dynamic Programming considers all **possible sequences** in order to obtain the optimum solution
In Greedy method there is **no** as such guarantee of getting **optimum solution**	It is guaranteed that the dynamic programming will generate optimal solution using principle of optimality

9. Define Spanning Tree. Discuss the design steps in prims and Kruskal's algorithm to construct minimum spanning tree. (APR/MAY 2018)

A spanning tree of a connected graph is its connected acyclic subgraph that contains all the vertices of the graph.

A tree T is a Spanning tree if a connected graph G(V,E) such that

- Every vertex of G belongs to an edge in T.
- The edge in T form a tree

Minimum Spanning Tree (MST)

A technique of building a spanning tree with minimum cost and weight is known as minimum spanning tree.

A Minimum Spanning tree of a weighted graph connected graph G is its spanning tree of the smallest weight, where the weight of a tree is defined as the sum of the weights on all its edges.

The total number of edges in minimum spanning tree (MST) is |V|-1 where V is the number of vertices.

Characteristics of Minimum Spanning Tree (MST)

A minimum spanning tree connects all nodes in a given graph

- A MST must be a connected and undirected graph
- A MST can have weighted edges
- Multiple MSTs can exist within a given undirected graph

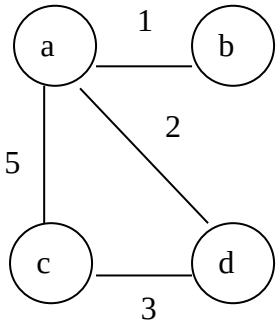
There are two types of spanning trees based on the traversal.

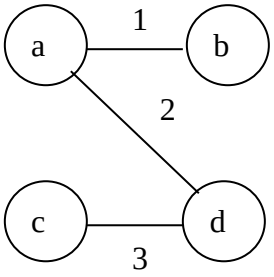
If the spanning tree resulting from a cell to DFS is known as a Depth First Spanning Tree.

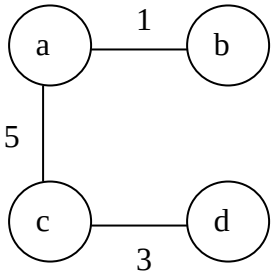
When BFS is used, the resulting spanning tree is called a Breadth First Spanning tree.

Example:

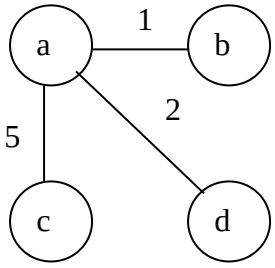
Shows the graph and its spanning tree







W(T1)=6 W(T2)=9



W(T3)=8

Spanning tree of graph T1 is the Minimum Spanning Tree

Types of algorithm to find MST

To find the MST, We have two algorithms as follows,

- Prim's algorithm
- Kruskal's algorithm

Characteristics of Kruskal and Prim algorithm

- Both Prim's and Kruskal's Algorithms work with undirected graphs
- Both work with weighted and un-weighted graphs but are more interesting when edges are weighted
- Both are greedy algorithms that produce optimal solutions

Applications of MST

- Network design- telephone, electrical, hydraulic, TV cable, computer, road
- Approximation algorithms for NP-hard problems, traveling salesperson problem

10. Explain Prim's Algorithm for constructing minimum cost spanning tree. Or Explain the working of Prim's Algorithm. Nov/Dec 2017

Prim's algorithm is a greedy algorithm for constructing a minimum spanning tree of a weight connected graph.

It works by attaching to a previously constructed subtree a vertex closest to the vertices already in the tree.

Procedure

- Prim's algorithm constructs a minimum spanning tree through a sequence of expanding subtrees.
- The initial subtree contains an arbitrarily selected vertex.
- On each iteration, the current tree is expanded in greedy manner by attaching the nearest vertex ti it.
- The nearest vertex will have the smallest weight.
- The algorithm stops, when all vertices have been included.
- If n is the number of vertices in a graphs then the total numbers of iterations is n-1.
- The tree generated by the algorithm is obtained as the set of edges used for the tree expansion.

Pseudo code of the algorithm

Algorithm Prim(G)

//Prim's algorithm for constructing a minimum spanning tree

//Input : A weight connected graph G = {V,E}

//Output: E_T , the set of edges composing a MST of G

$V_T \leftarrow \{v_0\}$// set of tree vertices can be initialize with any vertex.
$E_T \leftarrow \phi$ // empty set
for $i \leftarrow 1$ **to** $|V|- 1$ **do**
find the minimum weight edge, $e^* = (v^*, u^*)$ among all the edges(v,u) such that v is in V_T and u is in $V- V_T$
$V_T \leftarrow V_T$ union $\{u^*\}$
$E_T \leftarrow E_T$ union $\{e^*\}$
return E_T

In prim's algorithm,

The information about the shortest edge of each vertex can be provided.

The information can provided by attaching two labels to a vertex.

Name of the nearest vertex

Weight of the corresponding edge.

Vertices that does not have adjacent to the any vertex can be given.

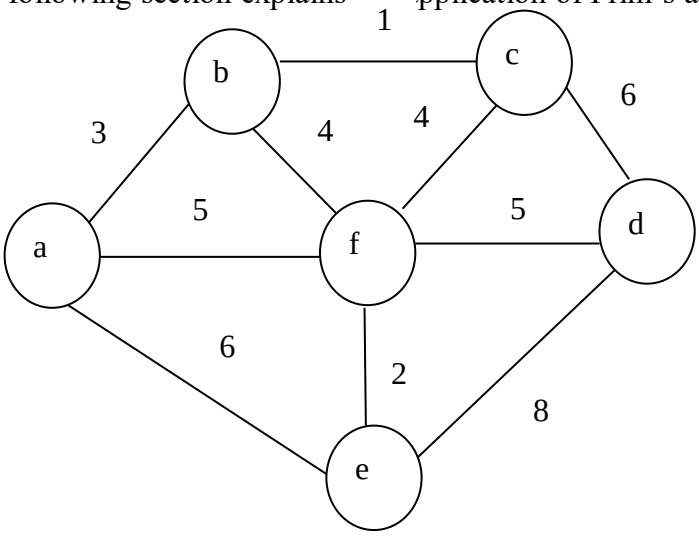
The label ∞ indicates infinite distance to the tree vertices

The null label is used to indicate the name of the nearest tree vertex

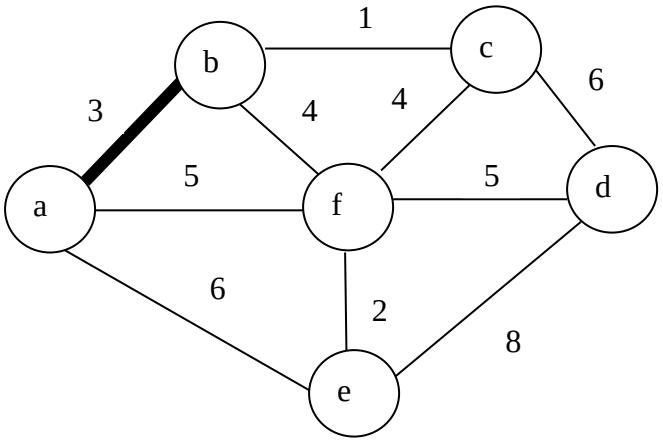
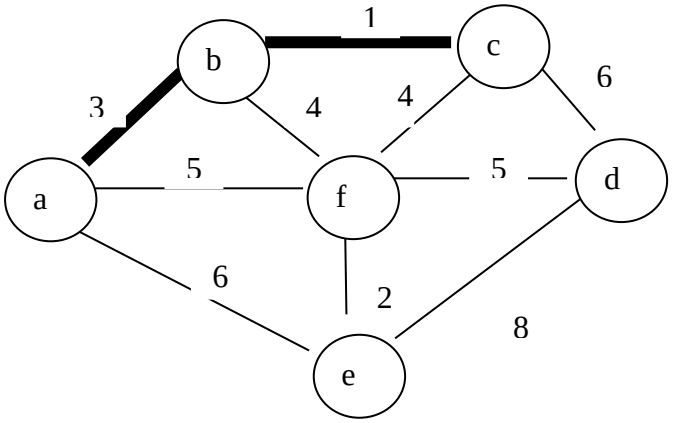
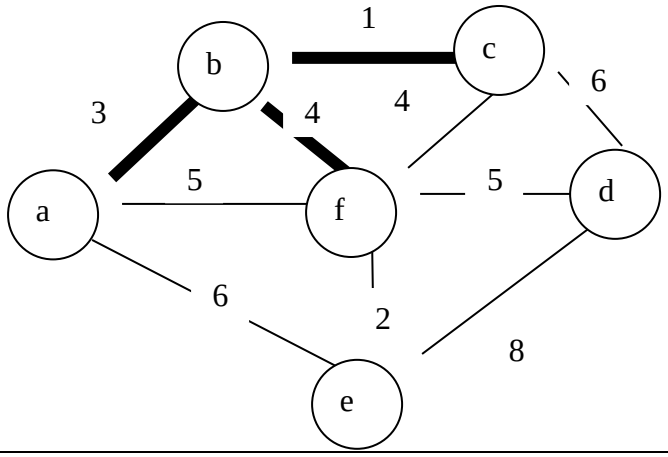
With the help of an above information it is easy to find the vertex with smallest distance.

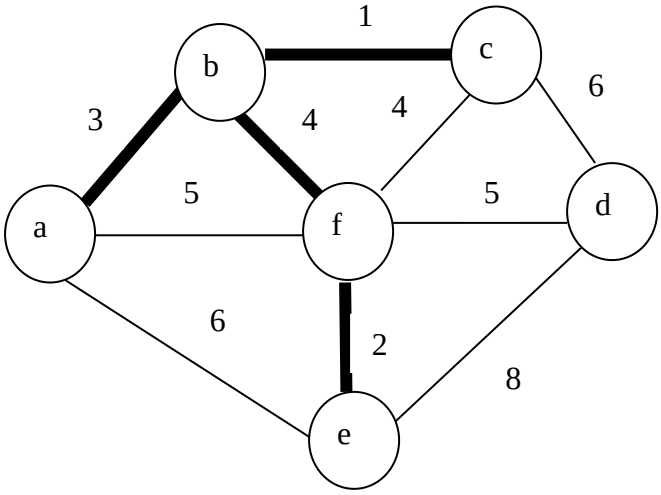
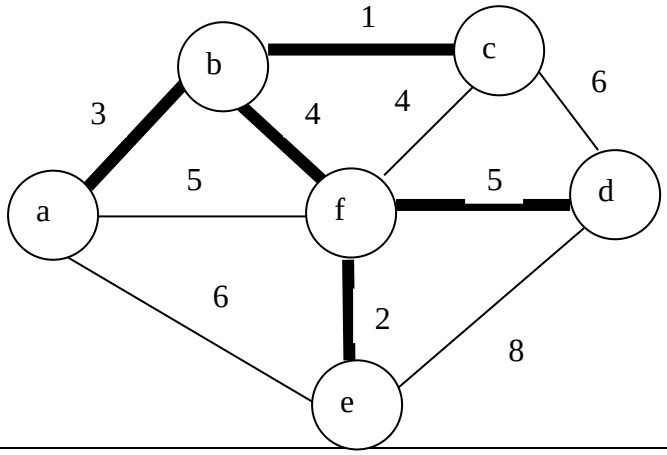
Prim's Example

The following section explains the application of Prim's algorithm to a specific graph.The graph is

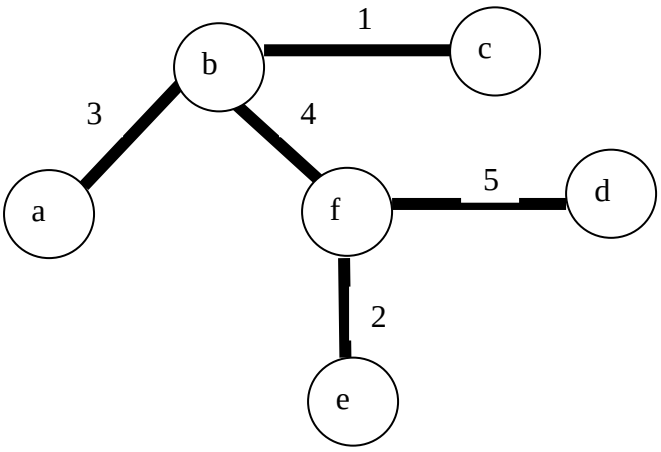


Tree Vertices	Remaining Vertices	Illustration

a(_,_)	**b(a,3),**c(_,∞), d(_,∞),e(a,6), f(a,5)	
b(a,3)	**c(b,1),**d(_,∞), e(a,6),f(b,4)	
c(b,1)	d(c,6),e(a,6), **f(b,5)**	

f(b,5)	d(f,5),**e(f,2)**	
e(f,2)	d(f,5)	
d(f,5)	-	-

Hence, the prim's algorithm finds the minimum spanning tree
The minimum spanning tree is



Analysis Of The Prims Algorithm

The algorithm spends most of the time in selecting the edge with minimum length .

Hence the **basic operation** of this algorithm is to find the edge with **minimum path length**.

This can be given by following formula

$$T(n) = \sum_{k=1}^{nodes-1} \sum_{i=0}^{nodes-1} + \sum_{j=0}^{nodes-1}$$

Time taken by For k=1 to nodes-1 loop

Time taken by For i=0 to nodes-1 loop

Time taken by For j=0 to nodes-1 loop

We take variable n for nodes for the sake of simplicity of solving the equation then

$$T(n) = \sum_{k=1}^{n-1} \sum_{i=0}^{n-1} + \sum_{j=0}^{n-1}$$

$$T(n) = \sum_{k=1}^{n-1} [((n-1)+0+1)+((n-1)+0+1)]$$

$$T(n) = \sum_{k=1}^{n-1} (2n)$$

$$T(n) = 2n \sum_{k=1}^{n-1} 1$$

$T(n) = 2n [(n-1) – 1 + 1]$

$T(n) = 2n(n-1)$

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

$T(n) = n^2$

$\mathbf{T(n) = \Theta(n^2)}$

But n stands for total number of nodes or vertices in the tree . hence we can also state

Time complexity of prim's algorithm is $\mathbf{\Theta(\|V^2\|)}$

11. Explain Kruskal's Algorithm for constructing minimum cost spanning tree.(AU april/may 2015) EXTRA

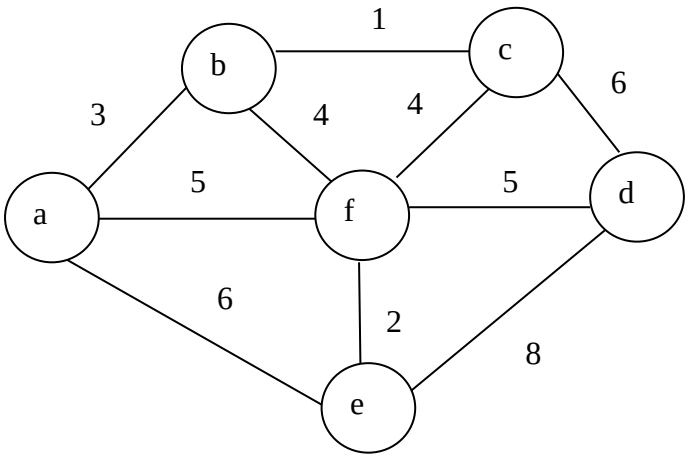
- Kruskal's algorithm is another greedy algorithm for the minimum spanning tree problem.
- It constructs a minimum spanning tree by selecting edges in increasing order of their weights provided that the inclusion does not create a cycle.
- Kruskal's algorithm provides a optimal solution.
- Kruskal's algorithm looks at a minimum spanning tree as an weighted connected graph G = {V,E} edges for which the sum of the edge weights is the smallest.
- The algorithm constructs a minimum spanning tree as an expanding sequence of subgraph, which are always acyclic but are not necessary connected on the intermediate stages of the algorithm.
- The algorithm begins by sorting the graph edges in non decreasing order of their weights.
- Then starting with the empty subgraph , it scans this sorted list adding the next edge on the list to the current subgraph if such an inclusion does not create a cycle and simply skipping the edge otherwise.

ALGORITHM *Kruskal(G)*

```
//Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph G = ⟨V, E⟩
//Output: E_T, the set of edges composing a minimum spanning tree of G
sort E in nondecreasing order of the edge weights w(e_i1) ≤ ··· ≤ w(e_i|E|)
E_T ← ∅;  ecounter ← 0        //initialize the set of tree edges and its size
k ← 0                         //initialize the number of processed edges
while ecounter < |V| − 1 do
    k ← k + 1
    if E_T ∪ {e_ik} is acyclic
        E_T ← E_T ∪ {e_ik};  ecounter ← ecounter + 1
return E_T
```

Example

The application of Kruskal's algorithm is explained for the graph is



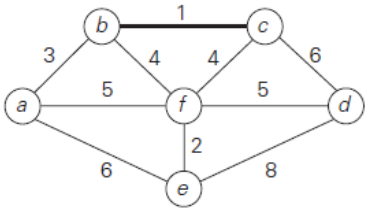
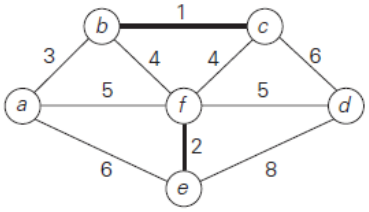
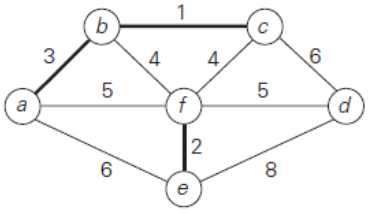
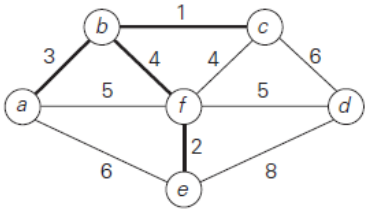
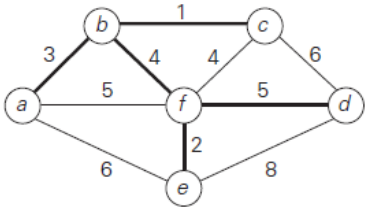
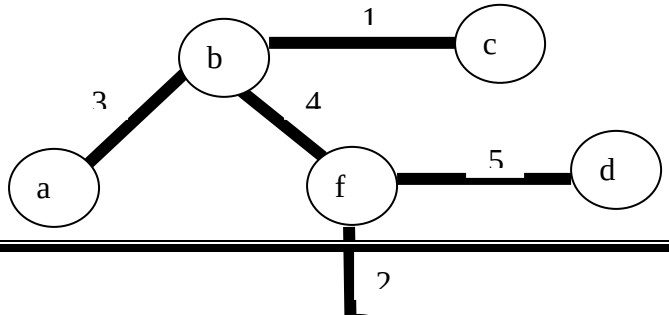
Tree edges	Sorted list of edges	Illustration
	bc ef ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bc 1	bc **ef** ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ef 2	bc ef **ab** bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ab 3	bc ef ab **bf** cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bf 4	bc ef ab bf cf af **df** ae cd de 1 2 3 4 4 5 5 6 6 8	
df 5		

FIGURE 9.5 Application of Kruskal's algorithm. Selected edges are shown in bold.

Hence, the Kruskal's algorithm finds the minimum spanning tree

The minimum spanning tree is



Analysis of the kruskal's algorithm

The worst –case running time of this algorithm is O (|E| log |E|), which is dominated by heap operation.

Prim's Vs Kruskal's algorithm

Prim's algorithm	**Kruskal's algorithm**
This algorithm is for obtaining minimum spanning tree selecting the adjacent vertices of already selected vertices	This algorithm is for obtaining minimum spanning tree but it is not necessary to choose adjacent vertices of already selected vertices

12. How will you find the shortest path between two given vertices using Dijkstra 's algorithm ? or Explain the Dijikstra's shortest path algorithm and its efficiency. Nov/Dec 2017 EXTRA

Shortest path: the path whose total weight (i.e., sum of edge weights) is minimum

Shortest-Path problems:

- **Single-source (single-destination).** Find a shortest path from a given source (vertex *s*) to each of the vertices. The topic of this lecture.
- **Single-pair.** Given two vertices, find a shortest path between them. Solution to single-source problem solves this problem efficiently, too.
- **All-pairs.** Find shortest-paths for every pair of vertices. Dynamic programming algorithm.
- **Unweighted shortest-paths** – BFS.

Single-source shortest paths problem

For a weighted graph G = (V,E,w), the single-source shortest paths problem is to find the shortest paths from a vertex source vertex v to all other vertices in the graph.

Algorithms used to resolve Single-source shortest-path problem

Common algorithms: *Dijkstra's* algorithm, *Bellman-Ford* algorithm

- **Dijkstra's algorithm** solves the single-source shortest path problem.
- **Bellman–Ford algorithm** solves the single-source problem if edge weights may be negative.
- **Floyd–Warshall algorithm** solves all pairs shortest paths.

BFS can be used to solve the shortest graph problem when the graph is **weightless or all the weights are the same**.

Shortest – Path Algorithm:

The shortest path algorithm determines the minimum cost of the path from source to every other vertex.

- The cost of the path V1, V2, ---Vn is $\sum_{i=1}^{N-1} C_{i,i+1}$.

This is referred as weight path length.

- The unweighted path length is merely the number of the edges on the path, namely N-1.

Two type of shortest path problems, exit namely,

- The single source shortest path problem.
- The all pairs shortest path problem.

Single source shortest path algorithm

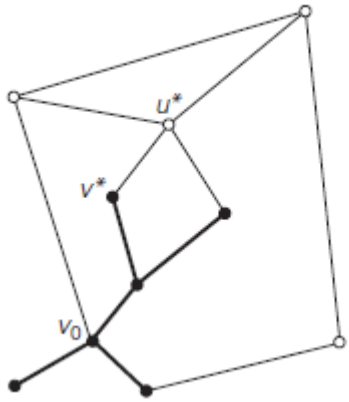
- The single source shortest path algorithm finds the minimum cost from single source vertex to all other vertices.
- Dijkstra's algorithm is used to solve this problem which follows the greedy technique.

All pairs shortest path algorithm

- All pairs shortest path problem finds the shortest distance from each vertex to all other vertices.
- To solve this problem dynamic programming technique known as floyd's algorithm is used.

Dijkstra's Algorithm:

- Dijkstra's algorithm solves the single source shortest path problem of finding **shortest paths** from a given vertex (the source), to all other vertices of a weight graph or digraph.
- It works as prim's algorithm but compares path lengths rather than edge lengths.
- Dijkstra's algorithm always provides a correct solution for a graph with non negative weight.
- Dijkstra's algorithm is applicable to graph with non negative weight only.
- It finds the shortest path from the source to vertex nearest ti it, then to a second nearest and so on.
- Before its ith iteration the algorithm would have identified the shortest paths to i-1 other vertices nearest to the source.
- These vertices, the source and the edges of the shortest paths leading to them from a subtree Ti of the given graph as hown in figure below



- The subtree of the shortest paths is found in hold.
- The next vertex nearest to the source can be found among the vertices adjacency to the vertices Ti, since all edge weights are non negative.
- The set of vertices adjacent to the vertices is Ti can be referred as " fringe vertices" from this vertex nearest to the source is chosen.

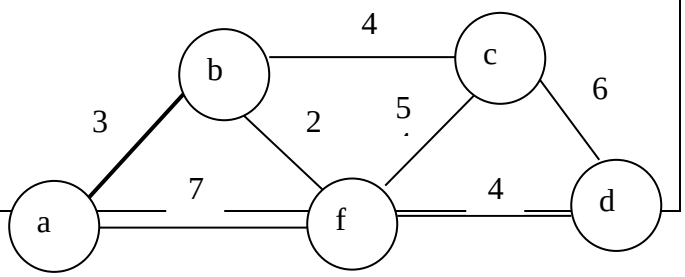
Identification of i th nearest vertex

- For every fringe vertex u, sum of the distance to the nearest tree vertex v and the length d_v of the shortest path from the source to v is computed.
- Then select the vertex with the smallest sum.
- The comparison of the lengths of special paths is the central insight of Dijkstra's algorithm.

Labels

- To perform the algorithm's operations two labels are used.
  - The numbers label d indicates the length of the shortest path from the source to vertex found.
  - The other label indicates the name of the next to last vertex on shortest path.(ie) the parent of the vertex in the tree being constructed.

- Using these labels , the next nearest vertex u* is identified.

Tree Vertices	Remaining Vertices	Illustration
a(_,0)	**b(a,3),c(_,∞), d(a,7),e(_,∞)**	

Operation

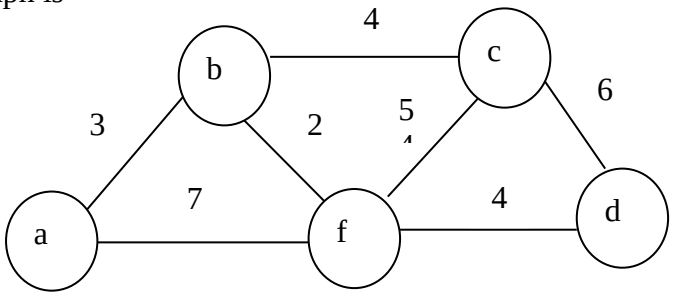
Two operations are performed to add u* to the tree. They are

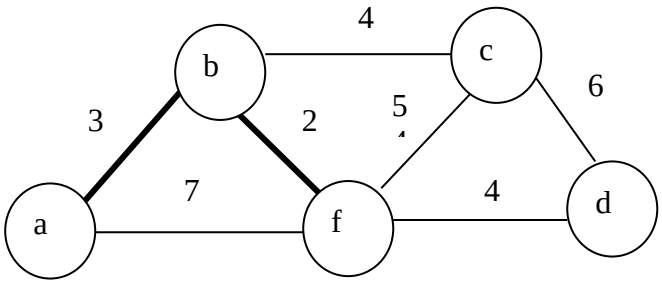
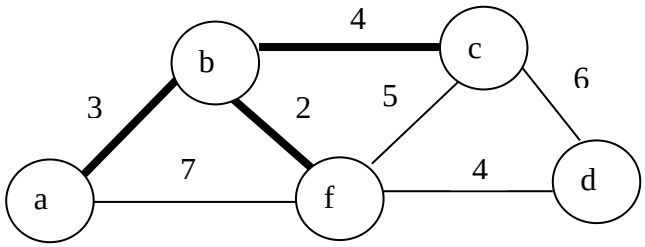
1. Move u* from the fringe to set of tree vertices.
2. For each remaining fringe vertex, u, that is connected to u* by an edge of weight w(u*,u) such that du*+ w(u*,u) < d_u , updates the labels of u by u* and du*+ w(u*,u) respectively.

Example

The application of Dijkstra's algorithm to the graph is explained in details.

The graph is



b(a,3)	**c(b,3+4),d(b,3+2), e(_,∞)**	
d(b,5)	**c(b,7),e(d,5+4)**	
e(d,9)		

The shortest paths and their lengths are

From a to b : a – b of length 3
From a to d : a – b - d of length 5
From a to c : a – b - c of length 5
From a to e : a – b – d - e of length 9

Dijkstra's algorithm compares path lengths and therefore add edge Weights.

Pseudocode of the algorithm

AlgorithmDijkstra(G.S)
// Dijkstra's algorithm for single source shortest paths
// Input: A weight connected graph G = <V,E > and its vertex S.
// Output: The length d_v of a shortest path from S to V.

Initialize (Q) //initialize vertex priority queue to empty
for every vertex v in V do
$d_v \leftarrow \infty$
$P_v \leftarrow$null
insert (Q,V, d_v)
$d_s \leftarrow 0$
decrease(Q,V,d_s)
$V_T \leftarrow \phi$
for i ← 0 to |V| - 1 do
u*←0Deletemin(Q)
$V_T \leftarrow V_T U \{u^*\}$
for every vertex u in V- V_T that is adjacent to u* do
if du*+w(u*,u)<du
du ← du*+w(u*,u)
pu ← u*
decrease (Q,V, d_u)

- The time efficiency of Dijkstra's algorithm depends on the structure used for implementing the priority queue and for representing as input graph.
- The efficiency is $\Theta(|V|^2)$ for graphs represented by their weight matrix and the priority queue implemented as an unordered array.
- The efficiency is $\Theta(|E|\log|V|)$ for graphs represented by the adjacency linked list and the priority queue implemented as a min heap.
- Better efficiency can be achieved if priority queue is implemented using a sophisticated data structure called the **Fibonacci Heap.**

13. Explain in detail about Huffman Trees. Or Write the Huffmans' algorithm. Construct. The Huffmans' tree for the following data and obtain its Huffmans' code Nov/Dec 2017 or

(i)write the Huffman code algorithm and derive its time complexity(5+2)

(ii)generate the Huffman code for the following data comprising of alphabet and their frequency.(6)

a:1, b :1 ,c :2, d :3, e :5, f: 8,g : 13,h : 21 Apr/May 2019

- A Huffman tree is a binary tree that minimizes the weighted path length from the root to the leaves containing a set of predefined weights.
- The most important application of Huffman trees are Huffman codes.
- A Huffman code is a optimal prefix tree variable length encoding scheme that assigns bit strings to characters based on their frequencies in a given text.
- This is accomplished by a greedy construction of a binary tree whose leaves represent the alphabet characters and whose edges are labeled with 0's and 1's.
- To encode a text that comprises n characters from some alphabet by assigning to each of the text's characters some sequence of bit called the **code word.**

Types of encoding

1. Fixed length encoding
2. Variable length encoding

Fixed length encoding

It assigns to each character a bit string of the some length m ($m>=\log_2 n$)

Variable length encoding

It assigns code words of different lengths to different characters.

Prefix free code or Prefix code

- In a prefix code no codeword is a prefix of a codeword of another characters.
- To construct a tree that would assign shorter bit strings to high frequency characters and longer ones to low frequency characters can be done by greedy algorithm invented by David Huffman.

Huffman's algorithm

Step 1

Initialize n one node trees and label them with the characters of the alphabet.
Record the frequency of each character in its tree's root to indicate the tree's weight.
The weight of a tree will be equal to the sum of the frequencies in the tree's leaves.

Step 2

Repeat the following operation until a single tree is obtained.
Find two trees with the smaller weight.
Make them, the left and right sub tree of a new tree and record the sum of their weights in the root of the new tree as its weight.

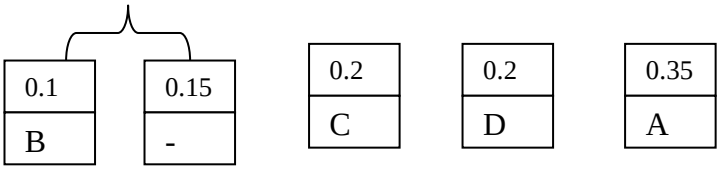
Example

Consider the five character alphabet {A,B,C,D,-} with the following occurrence probabilities.

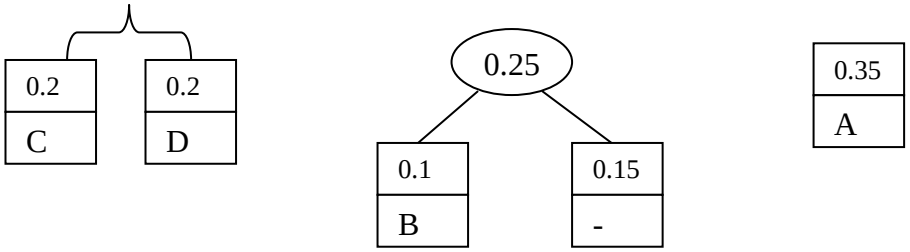
Character	A	B	C	D	-
Probability	0.35	0.1	0.2	0.2	0.15

The huffman tree construction for the input is

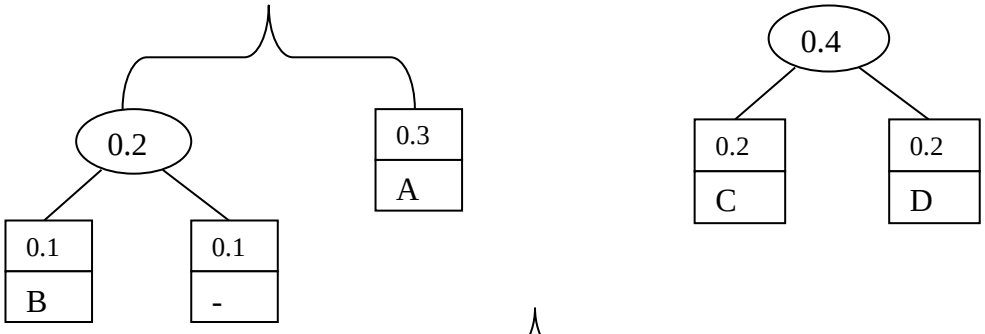
Step 1



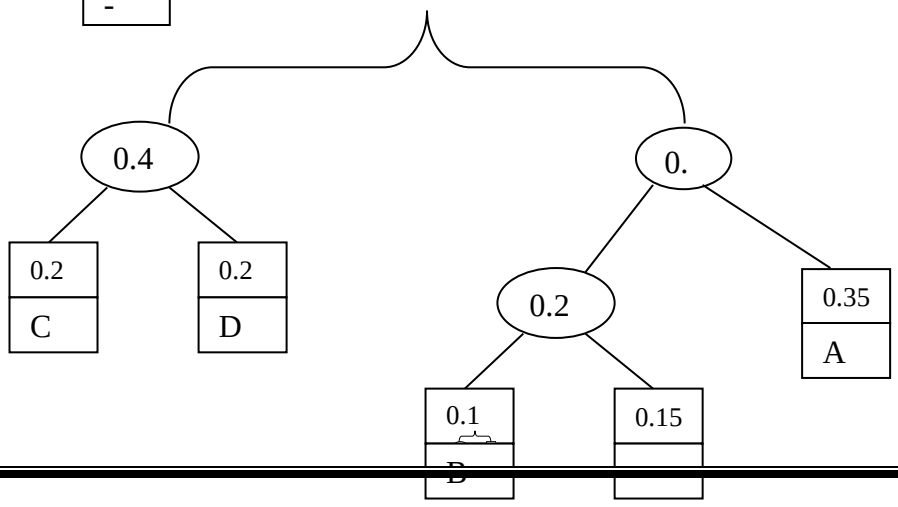
Step2



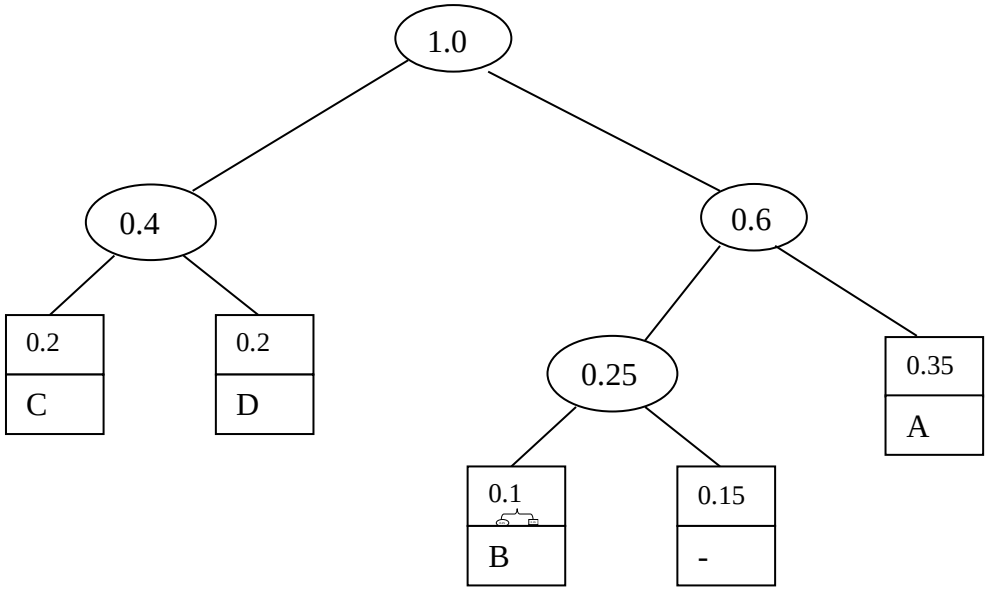
Step 3



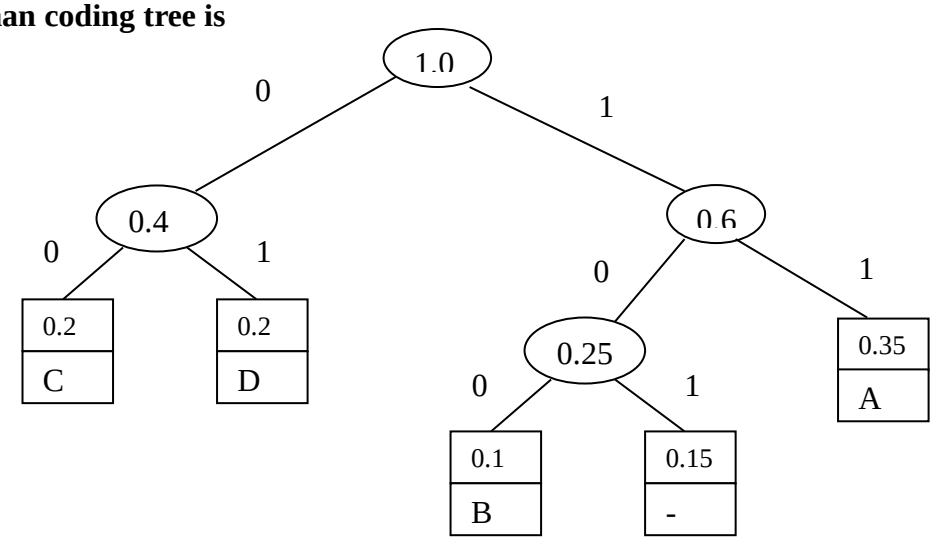
Step 4



Step 5



The Huffman coding tree is



Hence the resulting codeword for the characters are

Character	**A**	**B**	**C**	**D**	**-**
Probability	0.35	0.1	0.2	0.2	0.15
Codeword	11	100	00	01	101
Bits	2	3	2	2	3

Hence the string DAD is encoded as

D	A	D		DAD
01	11	01	⇨	011101

And BAD_AD is encoded as

B	A	D	_	A	D	BAD_AD
100	11	01	101	11	01	⇨ 100011011011101

With the occurrence probabilities given and the codeword lengths obtained, the expected number of bits per character in this code is calculated as,

Sum of the multiplications of probability of characters and number of bits in the code word.

(i.e) =0.35*2+0.1*3+0.2*2+0.2*2+0.15*3
= 0.7+0.3+0.4+0.4+0.45
=2.25

Therefore the expected number of bit per character is 2.25

In fixed length encoding, minimum three bits are used per characters.

Compression Ratio

Huffman's code achieves the compression ratio, which is a standard measure of compression algorithms effectiveness of

(3-2.25)/3*100=0.75/3*100
= 0.25*100
=25%

So, Huffman encoding of a text will use 25%less memory than its fixed length encoding.

Advantage of huffman's encoding

1) Huffman's encoding is one of the most important file compression methods.
2) It is simple
3) It is versatility
4) It provides optimal and minimum length encoding

Simple Version

- The simple version of Huffman compression calls for a preliminary scanning of a given text to count the frequencies of character occurrences in it.
- The frequencies are used to construct a Huffman coding tree and to encode the text.

Drawback of simplest version

- The information about the coding tree has to be included in to the encoded text to make the decoding possible.

Dynamic Huffman encoding

- Dynamic Huffman encoding is used to overcome the drawback of simplest version.
- In dynamic Huffman encoding, the coding tree is updated each time a new character is read from the source text.

Weighted path length

The weighted path length is defined as the sum $\sum_{I=1}^{n} l_i w_i$

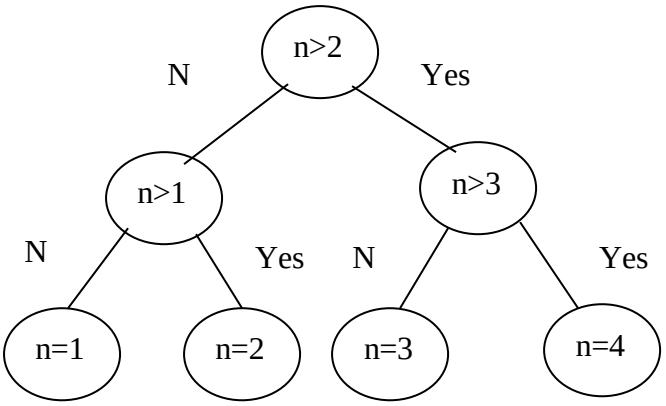
Where, l_i is the length of the simple path from the root to the i^{th} leaf.
w_i is the length of the frequency.

- In coding application,
  - l_i is the length of the codeword.
  - w_i is the length of the frequency.

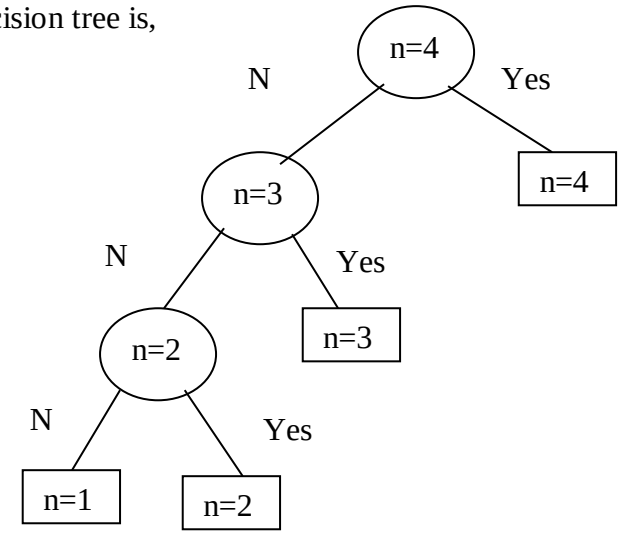
Huffman algorithm is used to construct a binary tree with a minimum weighted path length.

Example

Consider the game of guessing a chosen object from n possibilities.
When n=4, the decision tree is



The another decision tree is,



The length of simple path from the root to a leaf in a decision tree = number of questions needed to get to the chosen number represented by the leaf.

- If number I is chosen with probability p_i, the the sum is $\sum_{I=1}^{n} l_i P_i$

Where, l_i is the length of the path from the root P_i is probability.
The sum indicates the average number of question needed to guess the chosen number.

Application of Huffman trees:

- Huffman encoding is used in file compression algorithm
- Huffman's code is used in transmission of data in an encoded form
- This encoding is used in game playing method in which decision trees need to be formed

14. Using dynamic approach programming, solve the following graph using the backward approach. (APRIL/MAY 2011)

What is multistage graph ? List any three applications of multistage graph.(April/May 2021)

Multistage Graph:-

Concept:

The multistage graph problem is to find a minimum cost path from S to t.

Problem description:

- A multistage graph G=(V,E) is a directed graph in which the vertices are portioned into K> 2 disjoint sets Vi, 1<i<=K.
- if (u,v) is an edge in E, then u E Vi and VEVi+1 for some i., 1< = i< =K.
- The sets V1 and Vk are such that (V1)= VK/=1, Let S and t respectively the vertex in b1 and bk.
- The vertex S is the source, and t is the sin R. Let C (i, j) be the cost of edge (i,j)
- The cost of a path from S to t is the sun of the cost of edges on the path.
- Each set vi defines a stage in the graph Because of the constraints on E.
- Every path from S to t starts in stage 1, goes to stage 2, then to stage 3, then to stage 4, etc., and finally terminates in stage K.

Procedure for multistage problems:

Multistage graph using Backward Approach:-

- Find path from s to t, stage by stage.
- Every s to t path is the result of a sequence of K-2 decisions.
- The ith decision involves determining which vertex in V_{i+1}, 1<i< K-2, is to be on the path.
- P (i,j) be a minimum cost path from vertex j in v_i to vertex t.
- cost (i,j) be the cost of the path.
- Find cost of path using the formula.

Cost (i,j) = min {C(f,l) + Cost (i+1, l)}

l€ Vi+1

(j,l) €E

- Cost (K-1,j) =if c (j,t) €E
- Cost (K-1, j) = α if (j,t)) €E

- The shortest distance between source S and sink t using following formula.

Cost (K-2, j) for all j €V_{k-2}

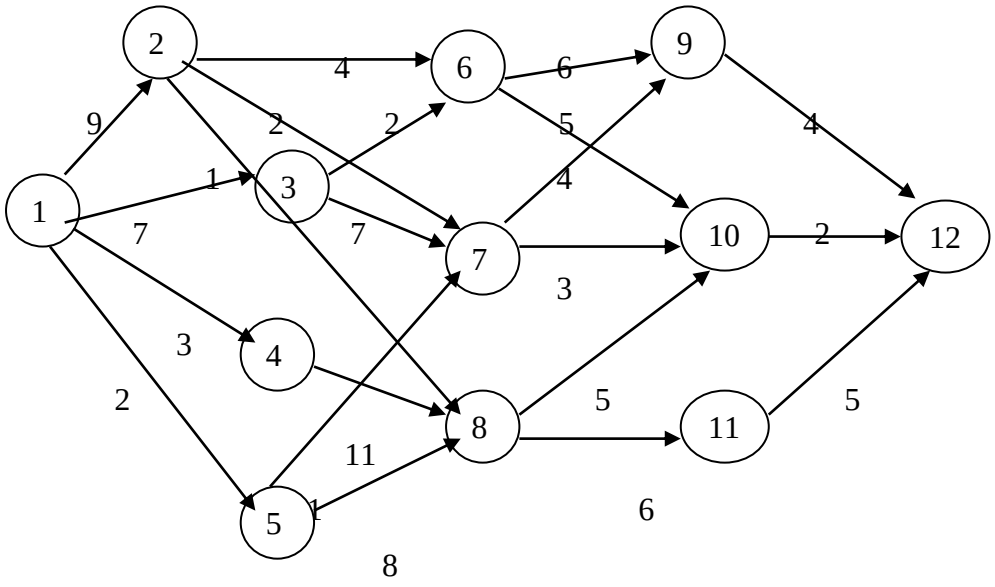
Cost (K-3,j) for all j € V_{K-3}

Cost (1,s)

Example:

Find the shortest distance between source ‘s’ and sink ‘t’.

Using 5 stage graph



1. Compute cost (k-2,j) for all j€V_{k-2}

K=5,because it is 5 stage graph.

III stage contains 6,7&8 ie., 3nodes.

i)Cost (i,j) = Min{c(j,l) + cost (i+1, l)}

ii)Cost (3,6) = Min{6+ Cost (4,9), 5+ cost (4,10)}

= Min{(6+4), 5+2)}

= Min {10,7}

Cost (3,6) =7

iii)Cost (3,7) = Min{c(j,l) + cost (i+1, l)}

=Min{(4+ cost (4,9), 3+ cost (4,10)}

= Min {(4+4, 3+2)}

= Min {(8,5)}

Cost (3,7) =5

iV)Cost (3,8) = Min{c(j,l) + cost (i+1, l)}

=Min{5+cost (4,10) 6+cost (4,11)}

= Min { 5+2, 6+5}

= Min (7,11)

Cost (3,8) = 7

2. Compute cost (k-3,j) for all j€V_{k-3}

II stage contains 2,3,4 & 5nodes.

i) Cost (2,2) = Min (4+cost(3,6) 2+cost (3,7), 1+ Cost (3,8))

= Min (4+7, 2+5,1+7)

= Min (11,7,8)

Cost (2,2) =7

ii) Cost (2,3) = Min (2+ Cost (3,6) 7+ Cost (3,7)

= Min (2+7, 7+5)

= Min (9,12)

Cost (2,3) = 9

iii) Cost (2,4) = Min (11+ Cost (3,8)

= Min (11+7)

= Min (18)

Cost (2,4) = 18

iv) Cost (2,5) = Min (11+ Cost (3,7) 8+Cost (3,8)

= Min (11+5, 8+7)

= Min (16, 15)

Cost (2,5) = 15

3. Compute cost (1,s)

I stage contains 1 node

i) Cost (1,1) = Min (9+ Cost (2,2) 7+ Cot (2,3),3+ Cost (2,4) 2+ Cost (2,5))

= Min (9+7, 7+9, 3+18, 2+15)

= Min (16,16,21,17)

Cost (1,1) = 16

Conclusion:- (Forward approach)

A Minimum cost S to t path has a cost of 16.

Program:-

Multistage graph using forward approach.

```
Void  F Graph (graphG, int K, int n, int P ( ) )
// The input is a K-stage graph G=(V,E) with n Vertices
// indexed in order of stages
//E is a set of edges and c(I,j) is the cost of (i,j)
// P(i:K)  is a minimum cost path vertex
{
Cost [n]=0.0;//cost of vertex n is zero
for (j= n-1; j>=1; j --)
{// compute cost (j)
 //Let r be the vertexsuch that (j,r) is an edge of G and
//c[j,r] + cost[r] is minimum
Cost[j] = C[j,r]+ Cost[r];
D[j] = r;
//find a minimum cost path
P[1] = 1
P[k] = n;
for(j=2,j<K-1; j+1)
P[j]=d(P[j-1]];
}
}
```

Let the minimum cost path be s=1, v_2, v_3 , V_{k-1}, t

for the above figure

v_2 =d(1,1)=2

v_3 =d(2,D(1,1))

v_3 =d(2,2)=7

=d(3,d(2,d(1,1)))

=d(3,7)

=10

Multistage graph using Backward Approach:-

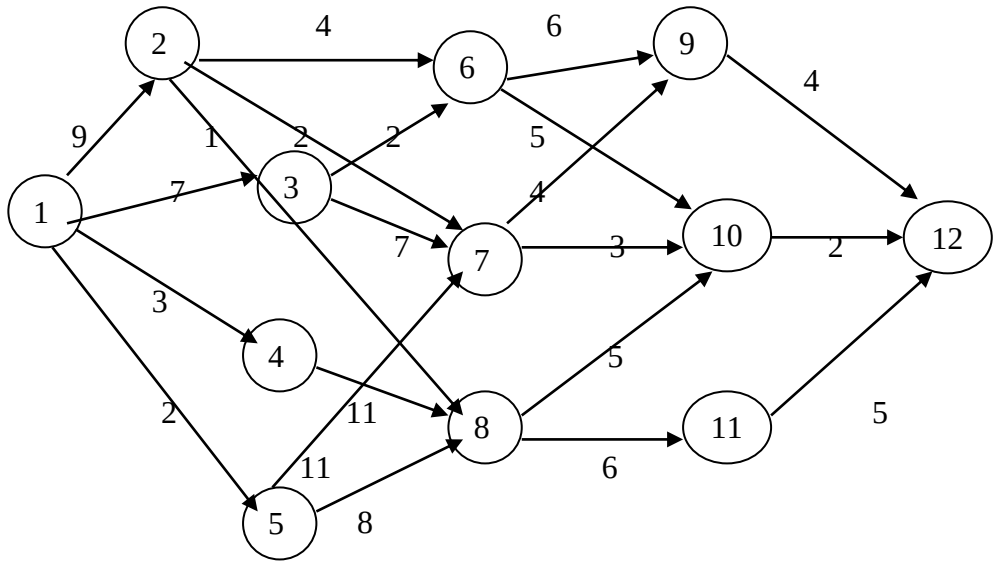
- The multistage graph can be solved using the backward approach.
- Let, bp(i,j) be a minimum cost path from vertex S to a vertex j in Vi
- bcost(i,j) be the cost of bp(i,j)

- Shortest path from source‘s’ to sink ‘t’ using backward
- bcost(i,j) = min {bcost (i-1, l) + c (l, j)}

  l€V_{i-1}

  (l,j)€E
- bcost(2,j)=c(1,j) if(1,j) €E
- bcost(2,j)=α if(1,j) €E

Find shortest path from source ‘s’ to sink ‘t’ for the following graph using backward approach.



i) Compute bcost for i=2

bcost(2,2)=min{c(1,2)}

= 9

bcost(2,3)=min{c(1,3)}

= 7

bcost(2,4)=min{c(1,4)}

= 3

bcost(2,5)=min{c(1,5)}

= 2

ii) Compute bcost for i=3

bcost (3,6) = min { bcost(2,2) +c(2,6),bcost(2,3) +c(3,6)}

= min {(9+2), (7+2)}

= min {13,9}

=9

bcost (3,7) = min (bcost (2,2) +c(2,7),bcost(2,3) +c(3,7),bcost(2,5)+c(5,7)}
= min {(9+2),(7+2),(2+11)}
= min {11,14,13)
=11

bcost (3,8) = min (bcost (2,2) +c(2,8),bcost(2,4) +c(4,8),bcost(2,5)+c(5,8)}
= min {(9+1),(3+11),(2+8)}
= min {10,14,10)
=10

iii) Compute bcost for i=4

bcost (4,9) = min { bcost(3,6) +c(6,9),bcost(3,7) +c(7,9)}
= min {(9+6), (11+5)}
= min {15,16}
=15

bcost (4,10) = min { bcost(3,6) +c(6,10),bcost(3,7) +c(7,10),
bcost(3,8)+c(8,10)}}
= min {(9+5),(11+3), (10+5)}
= min {14,14,15}
=14

bcost (4,11) = min { bcost(3,8) +c(8,11)}
= min {(10+6)}
= min {16}
=16

iv) Compute bcost for i=5

bcost (5,12) = min { bcost(4,9) +c(9,12),bcost(4,10) +c(10,12),
bcost(4,11)+c(11,12)}
= min {(15+4), (14+2),(16+5)}
= min {19,16,21}
=16

Conclusion: (Backward approach)

A minimum cost s to t path has a cost of 16.

Algorithm: (Backward approach)

Void Bgraph (graph G, int K, int n, int p[])

```
        {
        bcost [1] = 0.0;
//cost of vertex 1 is zero
for (j=2; j<n, j++)
{
// compute bcost [j]
//Let r be such that (r,j) is an edge of G and bcost[r]+C[r,j] is
// minimum
Bcost[j] = bcost[r]+C[r,j];
    D[j]=r;
}
//find a minimum  cost path
        P[1] =1;
        P[k]=n;
        for(j=k-1,j>=2,j--)
        P[j]=d[P[j+1]];
}
```

Complexity of multistage graph for both forward and backward approach.

Time complexity:

Finding the minimum cost for each and every stage – $\theta(|V|+|E|)$

Shortest path from source s to sink t-> $\theta(k)$

Space complexity:

Storage space for cost array cost[] - n location
Storage space for minimum cost path array p[] - n location
Storage space for decision array d[] - n location
Storage space for stage 'K' variable - 1
Storage space for variable 'n' - 1
Storage variable 'j' - 1
Total storage space -3n+3 =3(n+1)

Application of multistage problem:
Resource allocation problem

- n units of resource are to be allocated to 'r' projects
- The problem is to allocate the resource to r projects in such a way to maximize total net profit.

15.**Write the container loading greedy algorithm and explain. Prove that this algorithm is optimal. (NOV/DEC 2010)**
CONTAINER LOADING
Concept:

- A large ship is to be loaded with cargo. The cargo is containerized, and all containers are the same size. Different containers may have different weights.
- Let w_i be the **weight** of the i^{th} container, $1 \le i \le n$. The **cargo capacity** of the ship is **c**. we wish to load the ship with the **maximum number of containers.**
- Let x_i be a **variable** whose value can be either **0 or 1.**
- If we set x_i **to 0**, then container i is **not to be loaded**.
- If x_i **is 1,** then the **container is to be loaded.**
- We wish to assign values to the x_i**'s** that satisfy the constraints

$$\sum_{i=1} w_i x_i \le c \text{ and } x_i \in \{0, 1\}, 1 \le i \le n$$

- The **optimization function** is $\sum_{i=1} x_i$
- Every set of x_i**'s that** satisfies the constraints is a feasible solution. Every **feasible solution** that maximizes $\sum_{i=1}$ x_i **is an optimal solution.**
- The ship may be loaded in stages; one container per stage. At each stage we need to select a container to load. For this decision we may use the greedy criterion: **From the remaining containers, select the one with least weight.** This order of selection will keep the total weight of the selected containers **minimum** and hence leave **maximum capacity** for loading more containers.
- Using the greedy algorithm, we **first select** the container that has **least weight**, then the one with the **next smallest weight**, and so on until either all containers have been loaded or there isn't enough capacity for the next one.

```
Algorithm ContainerLoading (c, capacity, numberOfContainers, x)
//Greedy algorithm for container loading
//Set x[i]=1 iff container c[i], i ≥ 1 isloaded.
{
        //sort into increasing order of weight
        Sort (c, numberOfContainers);
        n:=numberOfContainers;
        // Initialize x
        for i:=1 to n do
        x[i]:= 0;
        //select containers in order of weight
        i:=1;
        while (i ≤ n && c[i].weight ≤ capacity)
        {
                //enough capacity for container c[i].id
                x[c[i].id]:=1;
                capacity =capacity - c[i].weight; //remaining capacity
                i++;
         }
}
```

Explanation: The greedy method loads containers in increasing order of their weight. Each element of the array c has two components - **id,** which is an identifier in the range 1 through number of containers (id is a container identifier) and **weight**, which is the weight of the container.

Example:

- Suppose that n=8, $[w_1,\cdots\cdots\cdots,w_8]$ = [100,200,50,90,150,50,20,80] and **c=400**.
- When the greedy algorithm is used, the containers are considered for loading in the

order 7, 3, 6, 8, 4, 1, 5, 2.

Step 1:

Arrange the containers in ascending order of their weights.
{7, 3, 6, 8, 4, 1, 5, 2} = {20, 50, 50, 80, 90, 100, 150, 200}
Initially, the solution set is {0, 0, 0, 0, 0, 0, 0, 0}

Step 2:

In stage 1, the container 7 is selected, whose weight is 20

- ⇨ **while** ($i \leq n$ && $c[i].weight \leq capacity$)
- ⇨ **while (1** ≤ 8 && $20 \leq 400$) condition **True**
- ⇨ The container 7 is loaded.
- ⇨ X_7 value is assigned to 1.
- ⇨ The solution set is **{0, 0, 0, 0, 0, 0, 1, 0}**
- ⇨ Capacity = capacity – c[i].weight ;
- ⇨ 380= 400-20
- ⇨ i value is increased by one.

Step 3:

In stage 2, the container 3 is selected, whose weight is 50

- ⇨ **while (2** ≤ 8 && $50 \leq 380$) condition **True**
- ⇨ The container 3 is loaded.
- ⇨ X_3 value is assigned to 1.
- ⇨ The solution set is **{0, 0, 1, 0, 0, 0, 1, 0}**
- ⇨ 380-50=330
- ⇨ i value is increased by one.

Step 4:

In stage 3, the container 6 is selected, whose weight is 50

- ⇨ **while (3** ≤ 8 && $50 \leq 330$) condition **True**
- ⇨ The container 6 is loaded.
- ⇨ X_6 value is assigned to 1.
- ⇨ The solution set is **{0, 0, 1, 0, 0, 1, 1, 0}**
- ⇨ 330-50=280
- ⇨ i value is increased by one.

Step 5:

In stage 4, the container 8 is selected, whose weight is 80

- ⇨ **while (4** ≤ 8 && $80 \leq 280$) condition **True**
- ⇨ The container is loaded.
- ⇨ X_8 value is assigned to 1.
- ⇨ The solution set is **{0, 0, 1, 0, 0, 1, 1, 1}**
- ⇨ 280-80=200
- ⇨ i value is increased by one.

Step 6:

In stage 5, the container 4 is selected, whose weight is 90

- ⇨ **while (5** ≤ 8 && $90 \leq 200$) condition **True**
- ⇨ The container 4 is loaded.
- ⇨ X_4 value is assigned to 1.
- ⇨ The solution set is **{0, 0, 1, 1, 0, 1, 1, 1}**
- ⇨ 200-90=110
- ⇨ i value is increased by one.

Step 7:

In stage 6, the container 1 is selected, whose weight is 100

- ⇨ **while (6** ≤ 8 && $100 \leq 110$) condition **True**
- ⇨ The container 1 is loaded.
- ⇨ X_1 value is assigned to 1.

- ⇨ The solution set is **{1, 0, 1, 1, 0, 1, 1, 1}**
- ⇨ 110-100=10
- ⇨ i value is increased by one.

- Containers 7, 3, 6, 8, 4 and 1 together weight **390 units are loaded**. The available capacity is now 10 units, which is inadequate for any of the remaining containers.

Conclusion:

- In the greedy solution we have

  [x_1, ····· , x_8] = [1, 0, 1, 1, 0, 1, 1, 1] and $\sum_i x_i = 6$

Time complexity:

- Sorting the containers is loaded in increasing order of weight. The sort takes **O(n log n)** time, where n is the number of containers.

16. Explain in detail about Optimal merge pattern .

Optimal merge pattern is a pattern that relates to the merging of two or more sorted files in a single sorted file. This type of merging can be done by the two-way merging method.

If we have two sorted files containing n and m records respectively then they could be merged together, to obtain one sorted file in time **O (n+m)**.

There are many ways in which pairwise merge can be done to get a single sorted file. Different pairings require a different amount of computing time.The main thing is to pairwise merge the n sorted files so that the number of comparisons will be less.

The formula of external merging cost is:

$$\sum_{i=1}^{n} f(i)d(i)$$

Where, f (i) represents the number of records in each file and d (i) represents the depth

Algorithm for optimal merge pattern

```
struct treenode {
        struct treenode *lchild, *rchild;
        int weight;
};
typedef struct treenode Type;

Type *Tree(int n)
//      list is a global list of n single node
//      binary trees as described above.
{
        for (int i=1; i<n; i++) {
            Type *pt = new Type;
            // Get a new tree node.
            pt -> lchild = Least(list); // Merge two trees with
            pt -> rchild = Least(list); // smallest lengths.
            pt -> weight = (pt->lchild)->weight
                    + (pt->rchild)->weight;
            Insert(list, *pt);
        }
        return (Least(list)); // Tree left in l is the merge tree.
}
```

An optimal merge pattern corresponds to a binary merge tree with minimum weighted external path length. The function tree algorithm uses the greedy rule to get a two- way merge tree for n files. The algorithm contains an input list of n trees. There are three field child, rchild, and weight in each node of the tree. Initially, each tree in a list contains just one node. This external node has lchildand rchild field zero whereas weight is the length of one of the n files to be merged. For any tree in the list with root node t, t = it represents the weight that gives the length of the merged file. There are two functions least (list) and insert (list, t) in a function tree. Least (list) obtains a tree in lists whose root has the least weight and return a pointer to this tree. This tree is deleted from the list. Function insert (list, t) inserts the tree with root t into the list.

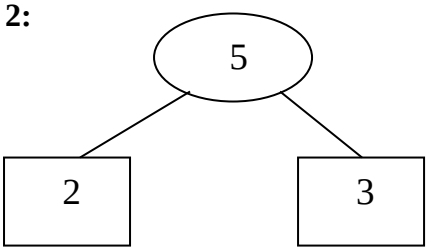
The main for loop in this algorithm is executed in n-1 times. If the list is kept in increasing order according to the weight value in the roots, then least (list) needs only O(1) time and insert (list, t) can be performed in O(n) time. Hence, the total time taken is O (n2). If the list is represented as a min heap in which the root value is less than or equal to the values of its children, then least (list) and insert (list, t) can be done in O (log n) time. In this condition, the computing time for the tree is O (n log n).

Example:
Given a set of unsorted files: 5, 3, 2, 7, 9, 13
Now, arrange these elements in ascending order: 2, 3, 5, 7, 9, 13
After this, pick two smallest numbers and repeat this until we left with only one number.

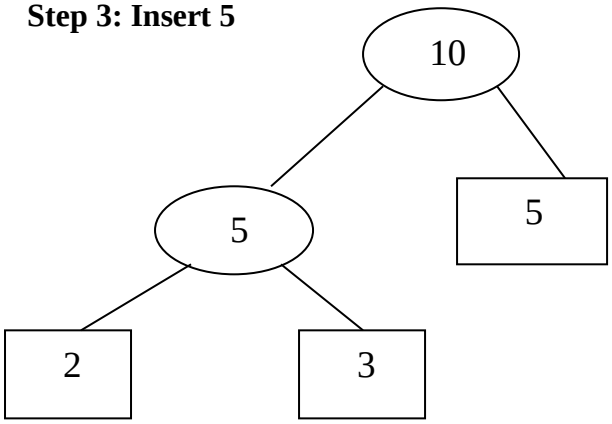
Now follow following steps:

Step 1: Insert 2, 3

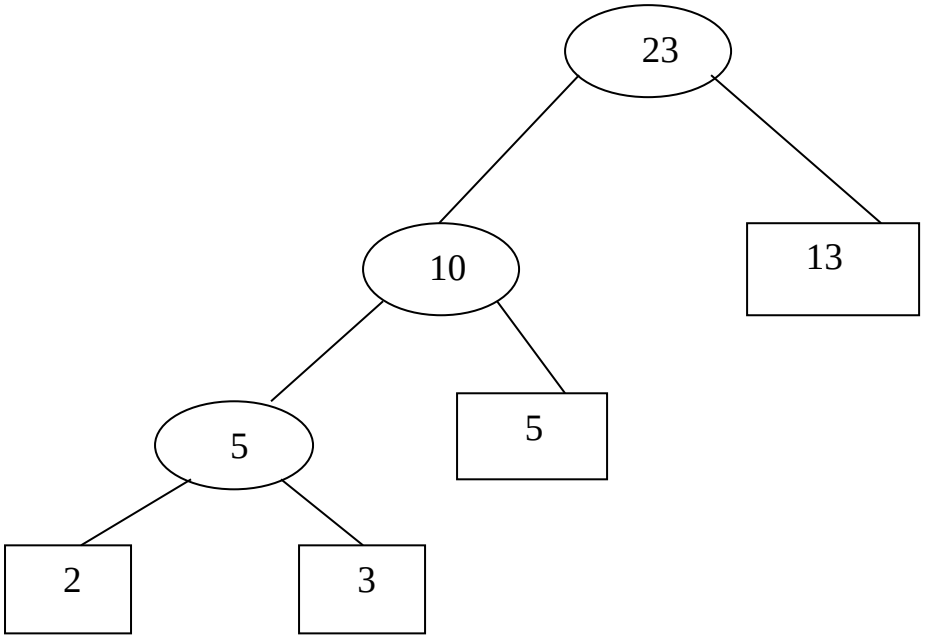
Step 2:



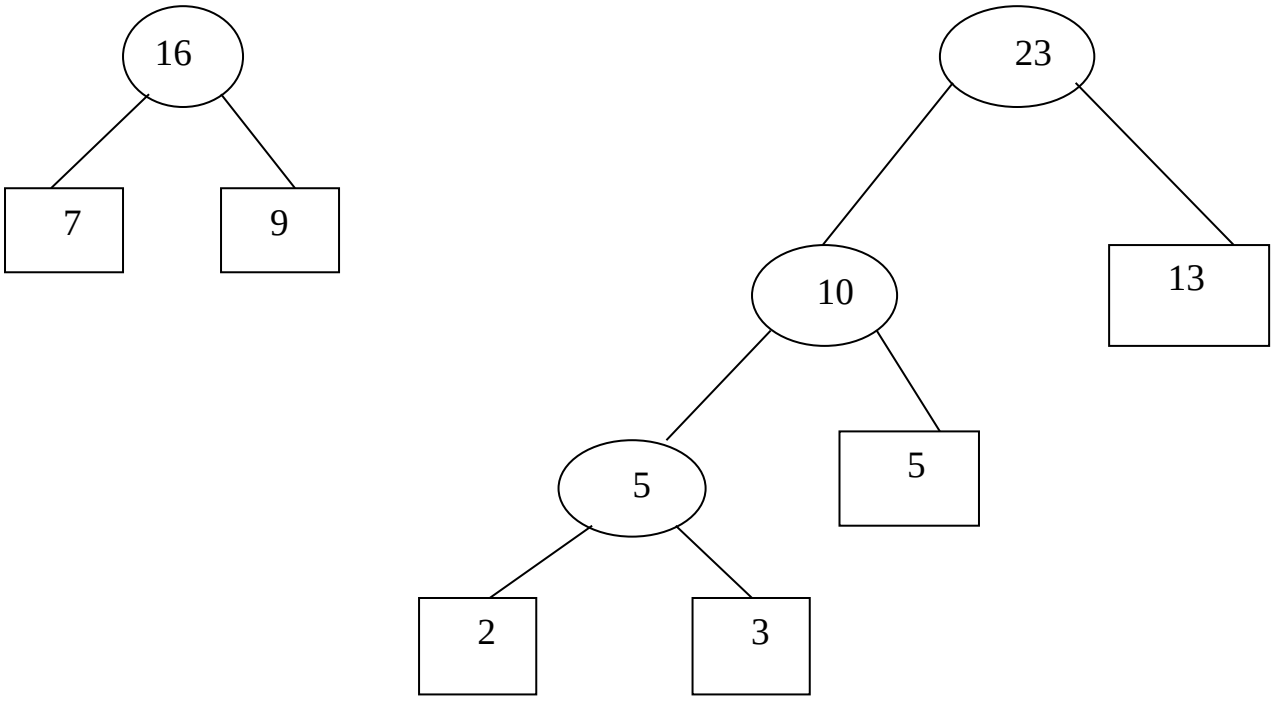
Step 3: Insert 5



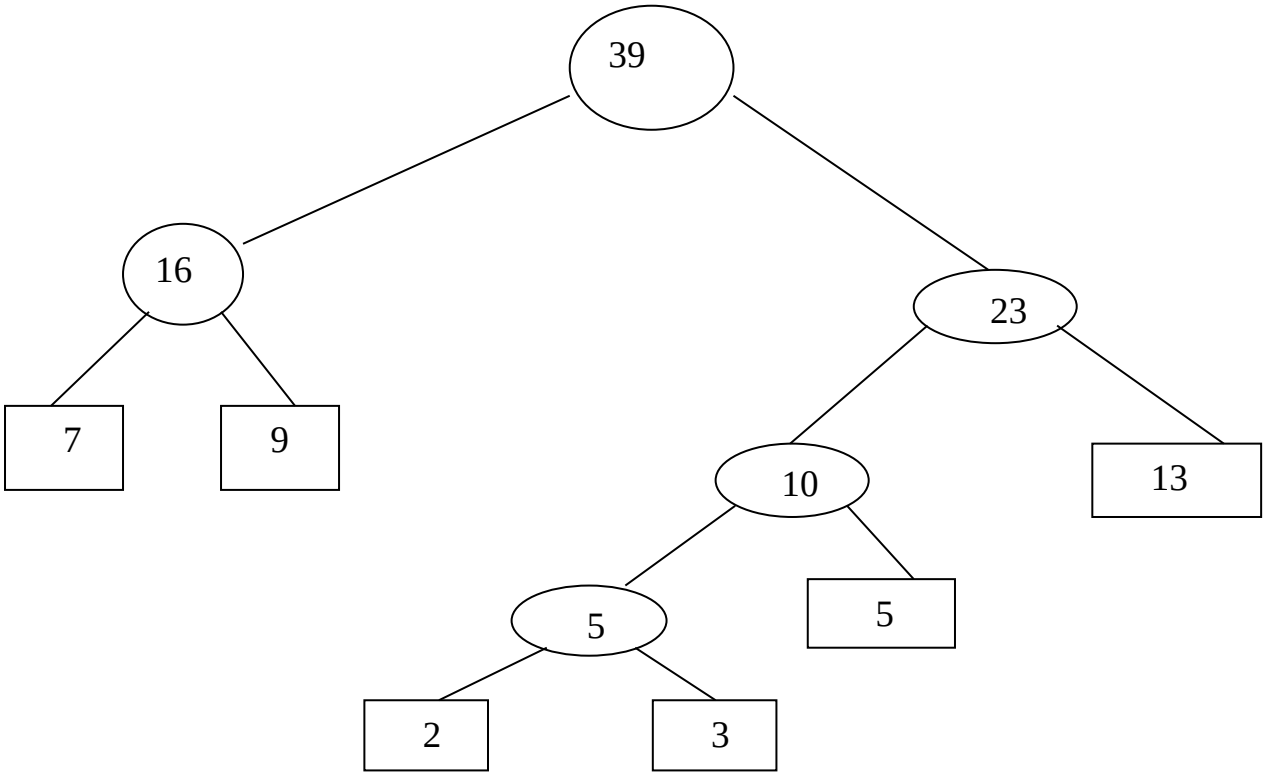
Step 4: Insert 13



Step 5: Insert 7 and 9



Step 6:



so, the merging cost = 5+10+16+23+39=93

17. Explain in detail about Coin-change Problem with example.

- To find the minimum number of Canadian coins to make any amount, the greedy method always works.
  - At each step select the largest denomination not going over the desired amount.
- The greedy method doesn't work if we didn't have 5¢ coin.
  - For 31¢, the greedy solution is 25 +1+1+1+1+1+1
  - But we can do it with 10+10+10+1
- The greedy method also wouldn't work if we had a 21¢ coin
  - For 63¢, the greedy solution is 25+25+10+1+1+1
  - But we can do it with 21+21+21

Coin set for examples

- For the following examples, we will assume coins in the following denominations:
  1¢ 5¢ 10¢ 21¢ 25¢
- We'll use 63¢ as our goal

A solution

- We can reduce the problem recursively by choosing the first coin, and solving for the amount that is left
- For 63¢:
  - One 1¢ coin plus the best solution for 62¢
  - One 5¢ coin plus the best solution for 58¢
  - One 10¢ coin plus the best solution for 53¢
  - One 21¢ coin plus the best solution for 42¢
  - One 25¢ coin plus the best solution for 38¢
- Choose the best solution from among the 5 given above
- We solve 5 recursive problems.

This is a very expensive algorithm

A dynamic programming solution

- **Idea:** Solve first for one cent, then two cents, then three cents, etc., up to the desired amount
  - *Save each answer in an array !*
- For each new amount N, combine a selected pairs of previous answers which sum to N
  - For example, to find the solution for 13¢,
    - First, solve for all of 1¢, 2¢, 3¢, ..., 12¢
    - Next, choose the best solution among:
      - Solution for 1¢ + solution for 12¢
      - Solution for 5¢ + solution for 8¢
      - Solution for 10¢ + solution for 3¢
- Let T(n) be the number of coins taken to dispense n¢.
- The recurrence relation
  - $T(n) = \min \{T(n-1), T(n-5), T(n-10), T(n-25)\} + 1, n \geq 26$
  - $T(c)$ is known for $n \leq 25$
- It is exponential if we are not careful.
- The bottom-up approach is the best.
- Memorization idea also can be used.
- The dynamic programming algorithm is O(N*K) where N is the desired amount and K is the number of different kind of coins.

15. Discuss the algorithm for finding a minimum cost binary search trees. Explain with suitable example.

16. Write down and explain the algorithm to solve all pairs shortest path problem. *Apr 2010*

17. Describe all pairs shortest path problem and write procedure to compute length of the shortest paths. *Jun 2013*Refer Part B – Q. No. 4

18. Explain the 0/1 knapsack with an algorithm *Jun 2014*Refer Part B – Q. No. 6

20. Solve All pairs shortest path problem for the digraph with the weight matrix given below. *Jun 2014*

	A	**B**	**C**	**D**
A	0	∞	3	∞
B	2	0	∞	∞
C	∞	7	0	1
D	6	∞	∞	0

Refer Part B – Q. No. 24

21. Explain an algorithm used to calculate binomial coefficient. *Mar 2014*Refer Part B – Q. No. 1

22. Discuss various memory functions and its properties. *Mar 2014* Part B – Q. No. 39

23. Write and explain the algorithm to compute the all pairs source shortest path using dynamic

programming and prove that it is optimal. *Nov 2010*Refer Part B – Q. No. 4

24. For the following graph having four nodes represented by the matrix given below determine the all pairs source shortest path. *Nov 2010*

0	∞	3	∞
2	0	∞	∞
∞	7	0	1
6	∞	∞	0

Step 1

	A	B	C	D
A	0	∞	3	∞
B	2	0	∞	∞
C	∞	7	0	1
D	6	∞	∞	0

Calculate the new shortest path between the vertices B-C and D-C

Shortest path B-C = BA + AC = 2 + 3 = 5

D-C = DA + AC = 6 + 3 = 9

Step 2

	A	B	C	D
	0	∞	3	∞
A				
B	2	0	**5**	∞
C	∞	7	0	1
D	6	∞	**9**	0

Calculate the new shortest path between the vertices C-A

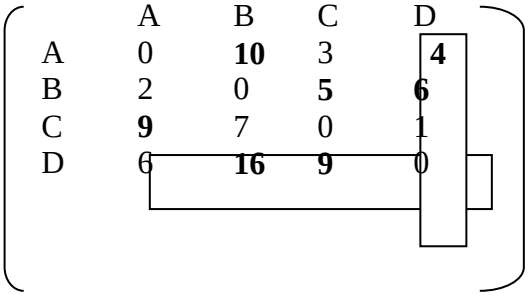
Shortest path C-A = BA + CB = 2 + 7 = 9

Step 3

	A	B	C	D
A	0	∞	3	∞
B	2	0	**5**	∞
C	**9**	7	0	1
D	6	∞	**9**	0

Calculate the new shortest path between the vertices A-B,
D-B, A-D, B-D
Shortest path A-B = AC + CB = 3 + 7 = 10
D-B = DC + CB = 9 + 7 = 16
A-D = AC + CD = 3 + 1 = 4
B-D = BC + CD = 5 + 1 = 6

Step 4



	A	B	C	D
A	0	**10**	3	**4**
B	2	0	**5**	**6**
C	**9**	7	0	1
D	6	**16**	**9**	0

Calculate the new shortest path between the vertices C-A
Shortest path C-A = CD + DA = 1 + 6 = 7
The shortest path between all pairs of algorithm is

	A	B	C	D
A	0	**10**	3	**4**
B	2	0	**5**	**6**
C	**7**	7	0	1
D	6	**16**	**9**	0

25. Write the algorithm to compute the 0/1 knapsack problem using dynamic programming and explain it.*Nov 2010*Refer Part B – Q. No. 6

26. Given the mobile numeric keypad.you can only press buttons that are up,left,right or doun to the first number pressed to obtain the subsequent numbers.you are not to press bottom row corner buttons .given a number N how many key strokes will be involved to press the given number.what is the length of it?which dynamic programming technique could be used to find solution for this ?explain each step with help of pseudocode and derive its time complexity.

Given the mobile numeric keypad. You can only press buttons that are up, left, right or down to the current button. You are not allowed to press bottom row corner buttons(i.e.*and#). Given a number N, find out the number of possible numbers of given length.
Examples:
For N=1,number of possible numbers would be 10 (0, 1, 2, 3, …., 9)

ForN=2,number of possible numbers would be 36

Possible numbers:00,08 11,12,14 22,21,23,25 and so on.

If we start with 0, valid numbers will be 00, 08 (count: 2)

If we start with 1, valid numbers will be 11, 12, 14 (count: 3)

If we start with 2, valid numbers will be 22, 21, 23,25 (count: 4)

If we start with 3, valid numbers will be 33, 32, 36 (count: 3)

If we start with 4, valid numbers will be 44,41,45,47 (count: 4)

If we start with 5, valid numbers will be 55,54,52,56,58 (count: 5)

……
……
We need to print the count of possible numbers.

Recursive Solution:

Mobile Keypad is a rectangular grid of 4X3 (4 rows and 3 columns)
Lets say Count(i, j, N) represents the count of N length numbers starting from position (i, j)
If N = 1
Count(i, j, N) = 10
Else
Count(i, j, N) = Sum of all Count(r, c, N-1) where (r, c) is new position after valid move of length 1 from current position (i, j)

Dynamic Programming

There are many repeated traversal on smaller paths (traversal for smaller N) to find all possible longer paths (traversal for bigger N). See following two diagrams for example. In this traversal, for N = 4 from two starting positions (buttons '4' and '8'), we can see there are few repeated traversals for N = 2 (e.g. 4 -> 1, 6 -> 3, 8 -> 9, 8 -> 7 etc).
aversals for N = 2 (e.g. 4 -> 1, 6 -> 3, 8 -> 9, 8 -> 7 etc

Few traversals starting for button 8 for N = 4
e.g. 8 -> 7 -> 4 -> 1, 8 -> 9 -> 6 -> 3
8 -> 5 -> 4 -> 1, 8 -> 5 -> 6 -> 3
8 -> 5 -> 2 -> 2, 8 -> 5 -> 2 -> 3

Few traversals starting from button 5 for N= 4

e.g. 5 -> 8 -> 7 -> 4, 5 -> 8 -> 9 -> 6

5 -> 4 -> 1 -> 2, 5 -> 6 -> 3 -> 2

5 -> 2 -> 1 -> 4, 5 -> 2 -> 3 -> 6

Since the problem has both properties: Optimal Substructure and Overlapping Subproblems, it can be efficiently solved using dynamic programming

IMPORTANT QUESTIONS

Part A

1.Write thedifference between Greedy method and Dynamic programming. *May 2011* – Q. No.

2. Write an algorithm to find shortest path between all pairs of nodes. *May 2011*

3.What is an optimal solution? *May 2010*

4.What is Knapsack problem? *Dec 2011*

5.What is greedy algorithms? *Dec 2011*

6.State the general principle of greedy algorithm? *Dec 2010*

7.What is the limitation of Greedy algorithm? *May 2010*

8.State the principle of optimality. *Dec 2010*

9.Compare feasible and optimal solution *May 2008*

10.What are optimal binary search trees OBST? *May 2010*

11.What is a Feasible solution ? *Dec 2013 / May 2014*

12.Differentiate between subset paradigm and ordering paradigm*Dec 2012*

13.What is the drawback of greedy algorithm ? *May 2012*

14.Write control abstraction for the ordering paradigm. *May 2012*

15.What is minimum Spanning tree? *Dec 2010*

16.Compare Greedy technique with dynamic programming method.*Dec 2012*

17.What is 0 / 1 knapsack problem *Dec 2012*

18.What is an optimal solution? *May 2010*

19.Define optimal binary search tree. *May 2010*

20.Define feasible and optimal solution. *un 2014*

21.State the principle of optimality.*Jun 2014 / Dec 2010*

22.List out the advantages of dynamic programming. *Jun 2014*

23.What is knapsack problem? *Jun 2013*

24.Write a note on Greedy approach. *Mar 2014*

25.Define dynamic programming. *Mar 2014*

26.What is memory function? *Mar 2014*

27.State the general principle of greedy algorithm. *Dec 2010*

28.Compare divide and conquer with dynamic programming and Dynamic programming with greedy technique. *Dec 2010*

Part B

1. Discuss the algorithm for finding a minimum cost binary search trees. Explain with suitable example. *Dec 2012*
2. Write down and explain the algorithm to solve all pairs shortest path problem.*Apr 2010* – Q. No. 4
3. Describe all pairs shortest path problem and write procedure to compute length of the shortest paths. *Jun 2013*
4. Explain the 0/1 knapsack with an algorithm. *Jun 2014*
5. Solve All pairs shortest path problem for the digraph with theweight atrix given below. *Jun 2014*

	A	B	C	D
A	0	∞	3	∞
B	2	0	∞	∞
C	∞	7	0	1
D	6	∞	∞	0

6. Explain an algorithm used to calculate binomial coefficient.*Mar 2014* – Q. No. 1
7. Discuss various memory functions and its properties. *Mar 2014*
8. Write and explain the algorithm to compute the all pairs source shortest path using dynamic programming and prove that it is optimal. *Nov 2010*
9. For the following graph having four nodes represented by the matrix given below determine the all pairs source shortest path. *Nov 2010*

0	∞	3	∞
2	0	∞	∞
∞	7	0	1
6	∞	∞	0

Refer Part B – Q. No. 22

10. Write the algorithm to compute the 0/1 knapsack problem using dynamic programming and explain it. *Nov 2010*
11. Explain Warshall's and Floyd' algorithm
12. .Explain Optimal Binary Search Trees with example
13. Explain Greedy Technique– Prim's algorithm- Kruskal's Algorithm with example

ANNA UNIVERSITY APRIL/MAY 2015

Part-A

1. Write down the optimization techniques used for warshall's algorithm.state the rules and assumption which are implied behind that (AU april/may 2015)**Part A – Refer Q. No. 42**
2. List out memory function used under dynamic programming.. (AU april/may 2015) **Refer Q. No. 39**

Part-B

1.(i)Given the mobile numeric keypad. You can only press buttons that are up,left,right or down to the first number pressed to obtain the subsequent numbers. You are not top bottom row corner buttons .given a number N how many key strokes will be involved to press the given number. What is the length of it? Which dynamic programming technique could be used to find solution for this ?explain each step with help of pseudo code and derive its time complexity.**Refer – Q. No. 24**

(ii) How do you construct minimum spanning tree using kruskal algorithm**Refer Part B– Q. No. 11**

2.(i) Let A={l/119,m/96,c/247,g/283,h/72,f/77,k/92,j/19} be the letters and its frequency of distribution in a text file.compute a suitable huffman coding to compress the data effectively.(8)
(ii)write algorithm to construct OBST given root (r(I,j) 0<=i<=j<=n.also proof that this code could be performed in time o(n). (AU april/may 2015) **Refer Part B – Q. No. 5**

ANNA UNIVERSITY NOV/DEC 2015

PART-A

1.State how binomial co-efficient is computes. **Refer Q. No. 45**
2.What is best algorithm suited to identity the topology for a graph.mention its efficiency factors. **Refer Q. No. 46**

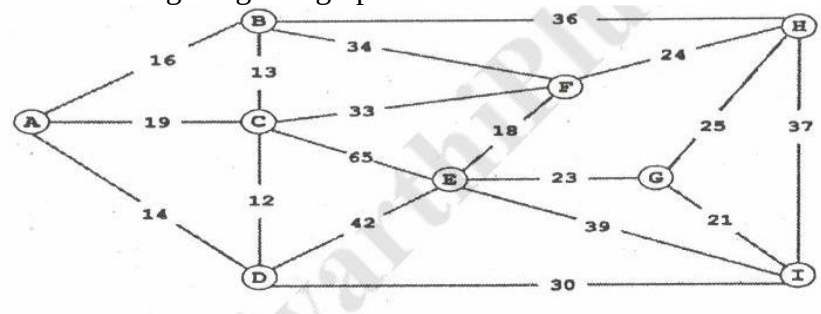
PART-B

12.a.(i) The binary string below is the title of song encoded using huffman codes
00110001011111011001110111011000001001110100101 01
Given the letter frequencies listed in the table below,build the huffman codes and use them to decode the title.in cases where there ar multiple "greedy"choices,the code are assembled by combinning the first letters (or groups of letter)from left to right,in the order given in table.also,code are assigned by labelling the left and right branches of the prefixcode treee with '0' and '1',respectively.

Letter:	a	h	v	w	''	e	t	l	o
Frequencies	1	1	1	1	2	2	2	3	3

(ii)Write the procedure to compute huffman code **Refer Q. No. 13**

b (i) Write and analysze the prim"s algorithm(8)
(ii)Consider the following weighted graph



Give the list of edges in the MST in the order that prim"s algorithm insert them.start prims algorithm from vertex A.(10) **Refer Q. No. 9**

ANNA UNIVERSITY APRIL/MAY 2016

PART-A

1. Define the single source shortest path problem. **Refer Q. No. 43**

2. State Assignment problem. **Refer Q. No. 44**

PART-B

1. (a) Discuss about the algorithm and pseudocode to find the Minimum Spanning Tree using Prim's Algorithm. Find the Minimum Spanning Tree for the graph shown below. Ans Discuss about the efficiency of the algorithm.(16) **Refer Q. No. 9 & 10**

OR

(b) Find all the solution to the traveling salesman problem (cities and distance shownbelow) by exhaustive search. Give the optimal solutions.(16) **Refer Q. No. 45**

[----------UNIT II Question asked in Unit III----------]

ANNA UNIVERSITY NOV/DEC 2016

PART-A

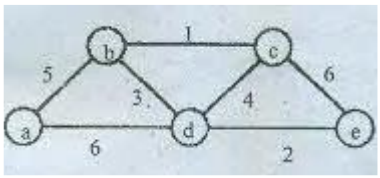
1.How to calculate the efficiency of dijkstra's Algorithm? **Refer Q. No. 45**

2.What is mean by principal of optimality? **Refer Q. No. 11**

PART-B

1.(a) Solve the all-pair shortest-path problem for the digraph with the following weight matrix(16) **Refer Q. No. 22**

b Apply kruskal's algorithm to find a minimum spanning tree of the following graph.(16) **Refer Q. No. 11**



ANNA UNIVERSITY APRIL/MAY 2016

PART-A

1. State the general principle of greedy algorithm? **Refer Q. No. 9**

2. What do you mean by dynamic programming? **Refer Q. No. 12**

PART-B

3. Solve the following instance of the 0/1, Knapsack problem given the knapsack capacity in W = 5 using dynamic programming and explain it. **Refer Q. No. 6**

Item	Weight	Value
1	4	10
2	3	20
3	2	**15**
4	5	**25**

4. Write the Huffmans' algorithm. Construct. The Huffmans' tree for the following data and obtain its Huffmans' code **Refer Q. No. 13**

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

ANNA UNIVERSITY NOV/DEC 2017

PART-A

1. What does Floyd's algorithm do ? **Refer Q. No. 11**
2. Define principle of Optimality . **Refer Q. No. 16**

PART-B

1. Explain the working of Prim's Algorithm.**Refer Q. No. 10**
2. Explain the Dijikstra's shortest path algorithm and the efficiency. **Refer Q. No. 12**

PART-C

1.Explain the steps in Building a Huffman Tree. Find the codes for the alphabets given below according to the frequency. **Refer Q. No. 13**

_ (space) 4
A 2
E 5
H 1
I 2
L 2
M 2
P 2
R 1
S 2
X 1

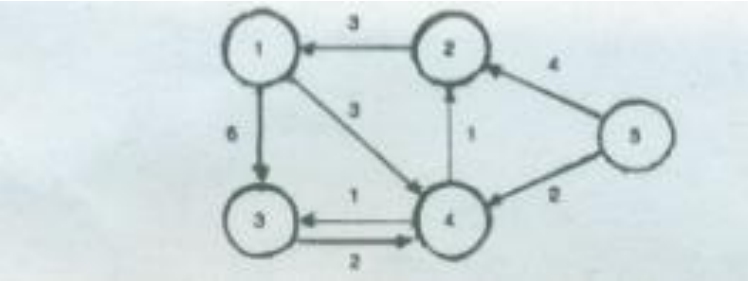
ANNA UNIVERSITY NOV/DEC 2018

PART-A

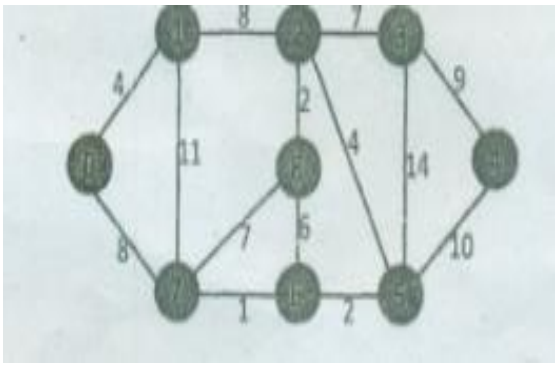
1. Define multistage graphs. Give an example. **Refer Q. No. 48**
2. How dynamic programming is used to solve Knapsack problem **Refer Q. No.49**

PART-B

11. (a) Explain Floyds – Warshall algorithm using dynamic programming. Trace the algorithm for the given example. **Refer Q. No.3 & 4** (13)

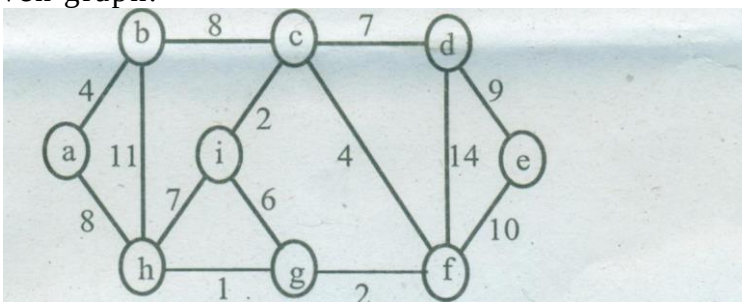


b) Explain how greedy approach is used in Dijkstra's algorithm for finding the single–source shortest paths for the given graph. (13) **Refer Q. No.11**



PART-C

1. Apply the greedy technique to find the minimum spaning tree using Prim's algorithm for the given graph. (15)



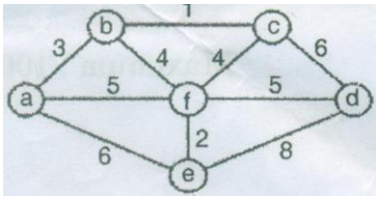
ANNA UNIVERSITY APR/MAY 2018

PART-A

1. Define transitive closure of a directed graph. **Refer Q. No.50**
2. Define the minimum spanning tree problem. **Refer Q. No.23**

PART-B

1. a) Give the Pseudo code for Prim's algorithm and apply the same to find the minimum spanning tree of the graph shown below : **Refer Q. No.9**



2. Explain the memory function method for the knapsack problem and give the algorithm. **Refer Q. No.7**

PART-C

1. Apply Warshall's algorithm to find the transitive closure of the digraph defined by the following adjacency matrix

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

i) Prove that the time efficiency of Warshall's algorithm is cubic. **(7)**

ii) Explain why the time efficiency of Warshall's algorithm is inferior to that of the traversal-based algorithm for sparse graphs represented by their adjacency lists. **Refer Q. No.3 & 4**

ANNA UNIVERSITY APRIL/MAY 2019

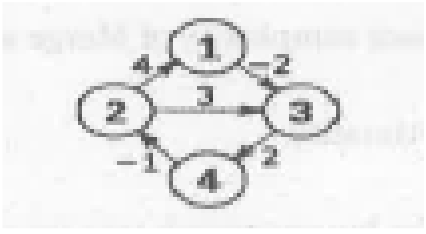
PART-A

1. State the principal of optimality. **Refer Q. No.11**

2. What is the constraint of for binary search tree insertion? **Refer Q. No.51**

PART-B

1. (i) write the Floyd algorithm to find all pair shortest path and derive its time complexity(4+3)
(ii) solve the following using floyd's algorithm.(6) **Refer Q. No.4**



2. (i)write the Huffman code algorithm and derive its time complexity(5+2)
(ii)generate the Huffman code for the following data comprising of alphabet and their frequency.(6)
a:1, b :1 ,c :2, d :3, e :5, f: 8,g : 13,h : 21 **Refer Q. No.13**

PART-C

1. (i) Given a matrix of order M*N and two coordinators (p,q) and (r,s) which represents the top left and bottom right of a sub-matrix*N,calculate the sum of elements present in the in the sub-matrix in O(1) time using dynamic programming. Determine the optimal sub-structure and write an algorithm
(ii)Prove that any algorithm that sorts by comparison, require Ω (n log n) time

2.(i)The longest common subsequence (LCS) is the problem of finding the longest subsequence that is present in the given two sequences in the same order but not necessarily contiguously. Write an algorithm using dynamic programming that determines the LCS of two string 'x' and 'y' and returns the string 'z'
(ii)Prove that any algorithm that searches need to necessarily do Ω (n log n)comparisons

ANNA UNIVERSITY NOVEMBER/DECEMBER 2019

PART-A

1. What is brute force method? **Refer Q. No.52**
2. Define a binary search tree. **Refer Q. No.53**

PART-B

1.(i) outline dynamic programming approach to solve the optimal binary search tree problem and analyse its time complexity
(ii) construct the optimal binary search tree for the following 5 keys with probabilities as indicated. **Refer Q. No.5**

i	0	1	2	3	4	5
Pi		0.15	0.10	0.05	0.10	0.20
pj	0.05	0.10	0.05	0.05	0.05	0.10

4. write a greedy algorithm to solve the 0/1 knapsack problem. Analyse its time complexity. Show that this algorithm is not optimal with an example.(5+2+6) **Refer Q. No.6**

PART-C

1.(i)The longest increasing subsequence(LIS)problem is to find the length of the longest subsequence of a given sequence such that all element of subsequence are sorted in increasing order. write an algorithm using dynamic programming that determines the lius of a string 'X'.For example the length of LIS for {10,22,9,33,21,50,41,60,80}is 60 and LIS is {10,22,33,50,60,80}.
(ii)Determine the time and space complexity of the above algorithm.

ANNA UNIVERSITY NOV/DEC 2021

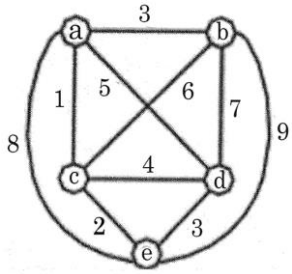
PART-A

1. What is meant by optimal substructure property of a dynamic programming problem. **Refer Q.No.54**

2. Write the control abstraction for greedy method. **Refer Q.No.55**

PART-B

1.a) i) **C**ompare and contrast dynamic programming and greedy method. **Refer Q.No.1**
(ii) Write the algorithm for optimal binary search tree and solve the following problem instance to construct the optimal binary search tree. Keys are a, b, c, d and the probabilities are 0.1, 0.2, 0.4 and 0.3. **Refer Q.No.5**
b) (i) Write the prim's algorithm to find the minimum spanning tree and illustrate the algorithm for the following graph. Where a, b, c, d and e are nodes and each edges are weighted by numbers. **Refer Q.No.10**



(ii) What is multistage graph? List any three applications of multistage graph. **Refer Q.No.14**

ANNA UNIVERSITY NOV/DEC 2021

PART-A

1. State the principle of optimality.
2. What is the container loading problem.

PART-B

1. a) State the knapsack problem and outline the steps to solve the knapsack problem using dynamic programming with an example.

   b) What is a minimum spanning tree? Outline the steps in the Kruskal's algorithm to find a minimum spanning tree with an example

CS8451- DESIGN AND ANALYSIS OF ALGORITHMS

SYLLABUS

UNIT I INTRODUCTION 9

Notion of an Algorithm – Fundamentals of Algorithmic Problem Solving – Important Problem Types – Fundamentals of the Analysis of Algorithmic Efficiency –Asymptotic Notations and their properties. Analysis Framework – Empirical analysis - Mathematical analysis for Recursive and Non-recursive algorithms – Visualization

UNIT II FORCE AND DIVIDE-AND-CONQUER 9

Brute Force – Computing an – String Matching - Closest-Pair and Convex-Hull Problems - Exhaustive Search - Travelling Salesman Problem - Knapsack Problem - Assignment problem. Divide and Conquer Methodology – Binary Search – Merge sort – Quick sort – Heap Sort - Multiplication of Large Integers – Closest-Pair and Convex - Hull Problems.

UNIT III DYNAMIC PROGRAMMING AND GREEDY TECHNIQUE 9

Dynamic programming – Principle of optimality - Coin changing problem, Computing a Binomial Coefficient – Floyd‘s algorithm – Multi stage graph - Optimal Binary Search Trees – Knapsack Problem and Memory functions. Greedy Technique – Container loading problem - Prim‘s algorithm and Kruskal's Algorithm – 0/1 Knapsack problem, Optimal Merge pattern - Huffman Trees.

UNIT IV ITERATIVE IMPROVEMENT 9

The Simplex Method - The Maximum-Flow Problem – Maximum Matching in Bipartite Graphs, Stable marriage Problem.

UNIT V COPING WITH THE LIMITATIONS OF ALGORITHM POWER 9

Lower - Bound Arguments - P, NP NP- Complete and NP Hard Problems. Backtracking – n-Queen problem - Hamiltonian Circuit Problem – Subset Sum Problem. Branch and Bound – LIFO Search and FIFO search - Assignment problem – Knapsack Problem – Travelling Salesman Problem - Approximation Algorithms for NP-Hard Problems – Travelling Salesman problem – Knapsack problem.

TOTAL: 45 PERIODS

TEXT BOOKS:

1. Anany Levitin, ―Introduction to the Design and Analysis of Algorithms‖, Third Edition, Pearson Education, 2012.
2. Ellis Horowitz, Sartaj Sahni and Sanguthevar Rajasekaran, Computer Algorithms/ C++, Second Edition, Universities Press, 2007.

REFERENCES:

1. Thomas H.Cormen, Charles E.Leiserson, Ronald L. Rivest and Clifford Stein, ―Introduction to Algorithms‖, Third Edition, PHI Learning Private Limited, 2012.
2. Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman, ―Data Structures and Algorithms‖, Pearson Education, Reprint 2006.
3. Harsh Bhasin, ―Algorithms Design and Analysis‖, Oxford university press, 2016.
4. S. Sridhar, ―Design and Analysis of Algorithms‖, Oxford university press, 2014.
5. http://nptel.ac.in/

UNIT I

PART A

1. **What is an algorithm? Or Define an algorithm. (Apr\May- 2017) Or Define algorithm with its properties.(April/May 2021)**
   - An algorithm is a finite set of instructions that, if followed, accomplishes a particular task.
   - In addition, all algorithms must satisfy the following criteria:
     - input
     - Output
     - Definiteness
     - Finiteness
     - Effectiveness.

2. **Define Program.**
   A program is the expression of an algorithm in a programming language.

3. **What is performance measurement?**
   Performance measurement is concerned with obtaining the space and the time requirements of a particular algorithm.

4. **Write the For LOOP general format.**
   The general form of a for Loop is

```
For variable : = value 1 to value 2
Step do
 {
        <statement 1>
        <statement n >
 }
```

5. **What is recursive algorithm?**
   - ✓ Recursive algorithm makes more than a single call to itself is known as recursive call.
   - ✓ An algorithm that calls itself is Direct recursive.
   - ✓ Algorithm A is said to be indeed recursive if it calls another algorithm,which in turn calls A

6. **What is space complexity?**
   The space complexity of an algorithm is the amount of memory it needs to run to completion.

7. **What is time complexity?**
   The time complexity of an algorithm is the amount of time it needs to run to completion.

8. **Give the two major phases of performance evaluation.**
   Performance evaluation can be loosely divided into two major phases:
   - a prior estimates (performance analysis)
   - a posterior testing (performance measurement)

9. **Define input size.**
   The input size of any instance of a problem is defined to be the number of elements needed to describe that instance.

10. **Define best-case step count.**
    The best-case step count is the minimum number of steps that can be executed for the given parameters.

11. Define worst-case step count.

The worst-case step count is the maximum number of steps that can be executed for the given parameters.

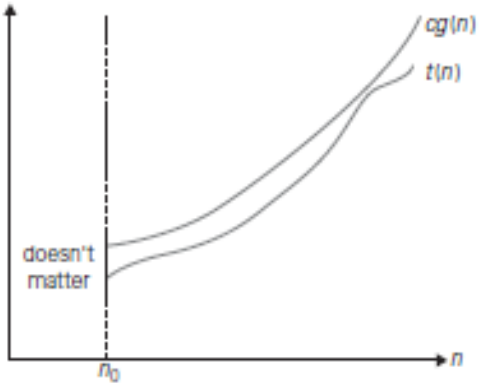
12. Define average step count.

The average step count is the average number of steps executed an instances with the given parameters.

13. Define the asymptotic notation "Big oh" (0)

A function t(n) is said to be in O(g(n)) **(t(n) Є O(g(n))),** if t(n) is bounded above by constant multiple of g(n) for all values of n, and if there exist a positive constant c and non negative integer n0 such that

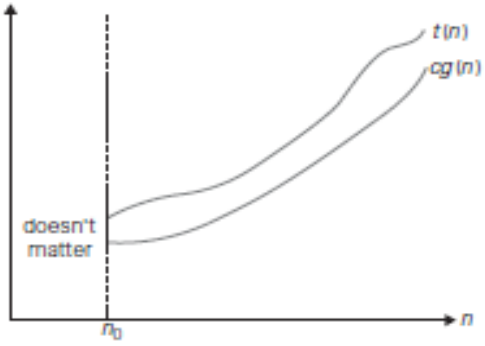
t(n) ≤ c*g(n) for all n ≥ n0.



1 Big-oh notation: $t(n) \in O(g(n))$.

14. Define the asymptotic notation "Omega" (Ω). NOV/DEC 2021

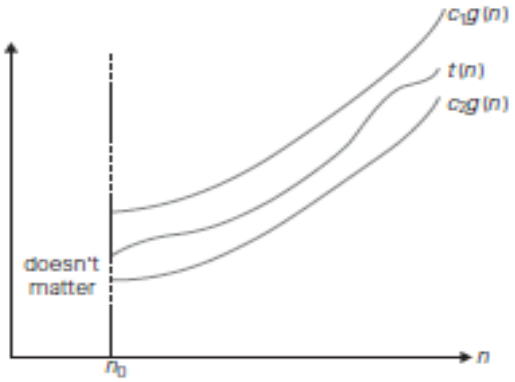
A function t(n) is said to be in Ω(g(n)) **(t(n) Є Ω(g(n))),** if t(n) is bounded below by constant multiple of g(n) for all values of n, and if there exist a positive constant c and non negative integer n0 such that **t(n) ≥ c*g(n)** for all n ≥ n0.



Big-omega notation: $t(n) \in \Omega(g(n))$.

15. Define the asymptotic notation "theta" (Θ)

A function t(n) is said to be in Θ(g(n)) **(t(n) Є Θ(g(n))),** if t(n) is bounded both above and below by constant multiple of g(n) for all values of n, and if there exist a positive constant c1 and c2 and non negative integer n0 such that **C2*g(n) ≤ t(n) ≤ c1*g(n)** for all n ≥ n0.



3 Big-theta notation: $t(n) \in \Theta(g(n))$.

16. What is a Computer Algorithm?

An algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

17. What are the features of an algorithm?

More precisely, an algorithm is a method or process to solve a problem satisfying the following properties:

Finiteness-Terminates after a finite number of steps

Definiteness-Each step must be rigorously and unambiguously specified.

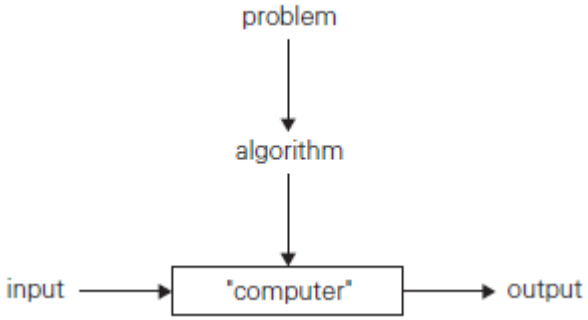
Input-Valid inputs must be clearly specified.

Output-Can be proved to produce the correct output given a valid input.

Effectiveness-Steps must be sufficiently simple and basic.

18. Show the notion of an algorithm. *Dec 2009 / May 2013*

An algorithm is a sequence of unambiguous instructions for solving a problem in a finite amount of time.



19. What are different problem types?

- Sorting
- Searching
- String Processing
- Graph problems
- Combinatorial Problems
- Geometric problems
- Numerical problems

20. What are different algorithm design techniques/strategies?

- Brute force
- Divide and conquer
- Decrease and conquer
- Transform and conquer

- Space and time tradeoffs
- Greedy approach
- Dynamic programming
- Backtracking
- Branch and bound

21. How to measure an algorithm's running time? Nov/Dec 2017

Unit for measuring the running time is the algorithms basic operation. The running time is measured by the count of no. of times the basic operations is executed.

Basic operation: the operation that contributes the most to the total running time.

Example: the basic operation is usually the most time-consuming operation in the algorithm's innermost loop.

22. How time efficiency is analyzed?

Let c_{op} – execution time of algorithms basic operation on a particular computer.

c(n) – no. of times this operation need to be executed.

T(n) – running time.

Running time is calculated using the formula

$$T(n) \approx c_{op}\, c(n)$$

23. What are orders of growth?

Orders of Growth

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

24. What are basic efficiency classes?

Basic Efficiency classes

1	Constant
log *n*	Logarithmic
n	Linear
n log *n*	*Linearithmic*
n^2	Quadratic
n^3	Cubic
2^n	Exponential
n!	Factorial

25. Give an example for basic operations.

Input size and basic operation examples

Problem	***Input size measure***	***Basic operation***
Searching for key in a list of *n* items	Number of list's items, i.e. *n*	Key comparison
Multiplication of two matrices	Matrix dimensions or total number of elements	Multiplication of two numbers

Checking primality of a given integer *n*	size = number of digits (in binary representation)	Division
Typical graph problem	Number of vertices and/or edges	Visiting a vertex or traversing an edge

26. What are six steps processes in algorithmic problem solving? ***Dec 2009***

1. Understanding the problem.
2. Ascertaining the capabilities of a computational device.
3. Choosing between exact and approximate problem solving.
4. Deciding on appropriate data structures.
5. Algorithm Design Techniques.
6. Methods of specifying an algorithm
7. Proving an algorithm's correctness.
8. Analysing an algorithm.
9. Coding an algorithm.

27. What do you mean by Amortized Analysis?

- ✓ Amortized analysis finds the average running time per operation over a worst case sequence of operations.
- ✓ Amortized analysis differs from average-case performance in that probability is not involved; amortized analysis guarantees the time per operation over worst-case performance.

28. Define order of an algorithm.

Measuring the performance of an algorithm in relation with the input size n is known as order of growth.

29. How is the efficiency of the algorithm defined? Or . How do you measure the efficiency of an algorithm? May/June 2019

The efficiency of an algorithm is defined with the components.

(i) Time efficiency -indicates how fast the algorithm runs
(ii) Space efficiency -indicates how much extra memory the algorithm needs

30. What are the characteristics of an algorithm?

Every algorithm should have the following five characteristics

(i) Input
(ii) Output
(iii) Definiteness
(iv) Effectiveness
(v) Termination

31. What are the different criteria used to improve the effectiveness of algorithm?

(i) The effectiveness of algorithm is improved, when the design, satisfies the following constraints to be minimum.

Time efficiency - how fast an algorithm in question runs.
Space efficiency – an extra space the algorithm requires.

(ii) The algorithm has to provide result for all valid inputs.

32. Analyse the time complexity of the following segment:

```
for(i=0;i<N;i++)
for(j=N/2;j>0;j--)
sum++;
```

Time Complexity= $N * N/2 = N^2/2 \in O(N^2)$

33. Write general plan for analysing non-recursive algorithms.

i. Decide on parameter indicating an input's size.

ii. Identify the algorithm's basic operation

iii. Check the no. of times basic operation executed depends on size of input. if it depends on some additional property, then best, worst, average cases need to be investigated

iv. Set up sum expressing the no. of times the basic operation is executed. (establishing order of growth)

34. How will you measure input size of algorithms?

The time taken by an algorithm grows with the size of the input. So the running time of the program depends on the size of its input. The input size is measured as the number of items in the input that is a parameter n is indicating the algorithm's input size.

35. Write general plan for analysing recursive algorithms.

i. Decide on parameter indicating an input's size.

ii. Identify the algorithm's basic operation

iii. Checking the no. of times basic operation executed depends on size of input. if it depends on some additional property, then best, worst, average cases need to be investigated

iv. Set up the recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed

v. Solve recurrence (establishing order of growth)

36. What do you mean by Combinatorial Problem?

Combinatorial Problems are problems that ask to find a combinatorial object-such as permutation, a combination, or a subset-that satisfies certain constraints and has some desired properties.

37. Define Little "oh".

The function $f(n) = 0(g(n))$ if and only if

$$\lim_{n \to \infty} f(n) / g(n) = 0$$

38. Define Little Omega.

The function $f(n) = \omega(g(n)))$ if and only if

$$\lim_{n \to \infty} f(n) / g(n) = 0$$

39. Write algorithm using iterative function to fine sum of n numbers.

Algorithm

```
sum(a, n)
{
    S := 0.0
    For i=1 to n
      do
        S : - S + a[i];
        Return S;
}
```

40. Write an algorithm using Recursive function to fine sum of n numbers.

Algorithm

```
Rsum (a, n)
{
        If(n≤0) then
                Return 0.0;
        Else
                Return Rsum(a, n- 1) + a(n);
}
```

41. Describe the recurrence relation for merge sort?

If the time for the merging operation is proportional to n, then the computing time of merge sort is described by the recurrence relation

$$T(n) = \begin{cases} a & n = 1, \quad a \text{ a constant} \\ 2T(n/2) + n & n > 1, \quad c \text{ a constant} \end{cases}$$

42. What is time and space complexity? *Dec 2012* Part A – Refer Q. No. 6 & 7

43. Define Algorithm validation. *Dec 2012*

The process of measuring the effectiveness of an algorithm before it is coded to know whether the algorithm is correct for every possible input. This process is called validation.

44. Differentiate time complexity from space complexity. *May 2010*

Part A – Refer Q. No. 6 & 7

45. What is a recurrence equation? *May 2010*

A recurrence [relation] is an equation or inequality that describes a function in terms of its values on smaller inputs.

Examples:

Factorial: multiply n by (n –1)!

$T(n) = T(n – 1) + O(1)$ -> $O(n)$

Fibonacci: add fibonacci(n – 1) and fibonacci(n – 2)

$T(n) = T(n – 1) + T(n – 2)$ -> $O(2^n)$

46. What do you mean by algorithm? *May 2013* Part A – Refer Q. No. 1, 16 & 18

47. Define Big Oh Notation. *May 2013* Part A – Refer Q. No. 13

48. What is average case analysis? *May 2014*

The average case analysis of an algorithm is analysing the algorithm for the average input of size n, for which the algorithm runs at an average between the longest and the fastest time.

49. Define program proving and program verification. *May 2014*

- ✓ Given a program and a formal specification, use formal proof techniques (e.g. induction) to prove that the program behaviour fits the specification.
- ✓ Testing to determine whether a program works as specified.

50. Define asymptotic notation. *May 2014*

Asymptotic notations are mathematical tools to represent time complexity of algorithms for measuring their efficiency.

Types :

- Big Oh notation - 'O'
- Omega notation - 'Ω'
- Theta notation - 'Θ'
- Little Oh notation - 'o '
- Little Omega notation - 'ω'

51. What do you mean by recursive algorithm? *May 2014* Part A – Refer Q. No. 5

52. Establish the relation between O and Ω *Dec 2010*

$f(n) \in \Omega(g(n)) \Leftrightarrow g(n) \in O(f(n))$

Proof:

$O(f(n))=\{g:N\to N \mid \exists c,n0\in N \ \forall n\geq n0: g(n)\leq c\cdot f(n)\}$

$\Omega(g(n))=\{f:N\to N \mid \exists c,n0\in N \ \forall n\geq n0: f(n)\geq c\cdot g(n)\}$

Step 1/2: $f(n) \in \Omega(g(n)) \Leftrightarrow g(n) \in O(f(n))$

$\exists c,n0 \in N\ \forall n \geq n0$: $f(n) \geq c \cdot g(n) \Rightarrow f(n)g(n) \geq c \Rightarrow 1g(n) \geq cf(n) \Rightarrow g(n) \leq 1c \cdot f(n)$

And this is exactly the definition of O(f(n)).

Step 2/2: $f(n) \in \Omega(g(n)) \Leftarrow g(n) \in O(f(n))$

$\exists c,n0 \in N\ \forall n \geq n0$: $g(n) \leq c \cdot f(n) \Rightarrow ... \Rightarrow f(n) \geq 1c \cdot g(n)$

Hence proved.

53. If $f(n) = a_m n^m + ... + a_1 n + a_0$. Prove that $f(n)=O(n^m)$. *Dec 2010* Refer Class note.

54. What is best case analysis? Or Best case efficiency.

The best case analysis of an algorithm is analysing the algorithm for the best case input of size n, for which the algorithm runs the fastest among all the possible inputs of that size.

55. what do you mean worst case efficiency of algorithm.Nov/Dec 2017

The worst case analysis of an algorithm is analysing the algorithm for the worst case input of size n, for which the algorithm runs the longest among all the possible inputs of that size.

56.Consider an algorithm that finds the number of binary digits in the binary representation ofa positive decimal integer. (AU april/may 2015)

Number of major comparisons=$\lfloor \log 2n \rfloor + 1 \in \log 2n$.

Algorithm 3: Finding the number of binary digits in the binary representation of a positive decimal integer.

```
Algorithm Binary(n)
count:=1;
whilen >1
do
count:=count+ 1;
n:=⌊n/2⌋;
end
return count;
```

57.write doun the properties of asymptotic notations.(AU april/may 2015)

The following property is useful in analyzing algorithms that comprise two consecutively executed parts.

Theorem

If $t_1(n)$ $O(g_1(n))$ and $t_2(n) \in O(g_2(n))$ then,

$t_1(n) + t_2(n) \in (\max \{g_1(n), g_2(n)\})$

Proof

Since $t_1(n) \in O(g_1(n))$, there exist some constant C_1 and some non negative integer n_1 such that

$t_1(n) \leq C_1 (g_1(n))$ for all $n \geq n_1$

Since

$t_2(n)$ $O(g_2(n))$

$t_2(n) \leq C_2 (g_2(n))$ for all $n \geq n_2$

Let us denote,

$C_3 = \max \{C_1, C_2\}$ and

Consider $n \geq \max \{n_1, n_2\}$, so that both the inequalities can be used.

The addition of two inequalities becomes,

$$t_1(n) + t_2(n) \leq C_1 (g_1(n)) + C_2 (g_2(n))$$
$$\leq C_3 (g_1(n)) + C_3 (g_2(n))$$
$$\leq C_3\, 2 \max\{g_1(n), (g_2(n))\}$$

Hence,

$t_1(n) + t_2(n) \in O(\max \{g_1(n), g_2(n)\})$,

with the constants C and n_0 required by the definition being $2C_3 = 2 \max (C_1, C_2)$ and $\max \{n_1, n_2\}$ respectively.

The property implies that the algorithms overall efficiency will be determined by the part with a larger order of growth.

(i.e.) its least efficient part is

$t_1(n) \in O(g_1(n))$
$t_2(n) \in O(g_2(n))$
} $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$

58. Give the Euclid's algorithm for computing *gcd(m, n)* (AU nov 2016) or write an algorithm to compute the greatest common divisor of two numbers (Apr/ May-2017)(or)

Give the Euclid's algorithm for computing gcd of two numbers. (May/June 2018)

	2	
Polynomial	**Quadratic**	**Quadratic**
1	0	1
2	1	4
4	6	16
8	28	64
10	45	10^2
10^2	4950	10^4
Complexity	Low	High
Growth	Low	high

ALGORITHM *Euclid_gcd(m, n)*

//Computes gcd*(m, n)* by Euclid's algorithm
//Input: Two nonnegative, not-both-zero integers *m* and *n*
//Output: Greatest common divisor of *m* and *n*
while $n \neq 0$ **do**
    $r \leftarrow m \bmod n$
    $m \leftarrow n$
    $n \leftarrow r$
return *m*

Example: gcd*(*60, 24*)* = gcd*(*24, 12*)* = gcd*(*12, 0*)* = 12.

59.Compare the order of growth n(n-1)/2 and n^2. (AU nov 2016)

n(n-1)/2 is lesser than the half of n^2

60.The *(log n)*th smallest number of n unsorted numbers can be determined in O(n) average-case time

Ans: True

61.Fibonacci algorithm and its recurrence relation

Algorithm for computing Fibonacci numbers

First method

Algorithm F(n)

//Computes the nth Fibonacci number recursively by using its definition.

//Input: A nonnegative integer n

//Output: The nth Fibonacci number

```
if n<1
        return n
Else
        return F(n-1)+(n-2)
```

the algorithm's basic operation is addition.

Let A(n) is the number of additions performed by the algorithm to compute F(n).

The number of additions needed to compute F(n-1) is A(n-1) and the number of additions needed to compute F(n-2) is A(n-2).

62. Design an algorithm to compute the area and circumference of a circle

1. Find Area, Diameter and Circumference of a Circle.

ALGORITHM

Step 1: Start
Step 2: Initialize PI to 0
Step 3: Read radius of the circle
Step 4: Calculate the product of radius with itself and PI value
Step 5: Store the result in variable area
Step 6: Calculate the product of radius with 2
Step 7: Store the result in variable diameter
Step 8: Calculate the product of PI and diameter value
Step 9: Store the result in variable circumference
Step 10: Print the value of area, diameter and circumference
Step 11: Stop

PSEUDO CODE

Step 1: Begin
Step 2: Set PI to 3.14
Step 3: Read radius
Step 4: Compute the product of radius with itself and PI value
Step 5: Set the result to area
Step 6: Compute the product of radius with 2
Step 7: Set the result to diameter
Step 8: Calculate the product of PI and diameter value
Step 9: Set the result to circumference
Step 10: Display area, diameter, circumference
Step 11: End

63. What is a basic operation?

A basic operation could be: An assignment. A comparison between two variables. An arithmetic operation between two variables. The worst-case input is that input assignment for which the most basic operations are performed.

Basic Operations on Sets. The set is the basic structure underlying all of mathematics. In algorithm design, sets are used as the basis of many important abstract data types, and many techniques have been developed for implementing set-based abstract data types.

64. Define algorithm. List the desirable properties of an algorithm.

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

An algorithm must satisfy the following properties: Input: The algorithm must have input valuesfrom a specified set. ... The output values are the solution to a problem. Finiteness: For any input, the algorithm must terminate after a finite number of steps. Definiteness: All steps of the algorithm must be precisely defined.

65. Define best, worst, average case time complexity.

- The *worst-case complexity* of the algorithm is the function defined by the maximum number of steps taken on any instance of size *n*. It represents the curve passing through the highest point of each column.
- The *best-case complexity* of the algorithm is the function defined by the minimum number of steps taken on any instance of size *n*. It represents the curve passing through the lowest point of each column.
- Finally, the *average-case complexity* of the algorithm is the function defined by the average number of steps taken on any instance of size *n*.

66.Prove that the of f(n)=o(g(n)) and g(n)=o(f(n)),then f(n)=θ g(n). OR state the transpose symmetry property of O and Ω **April/May 2019,Nov/Dec 2019**

Given function:

f(n) and g(n)

f(n)= O(g(n)) when f(n) $\leq C_1 g(n)$ for all $n \geq n_0$--------(1)

f(n)= Ω(g(n)) when f(n) $\geq C_2 g(n)$ for all $n \geq n_0$--------(2)

from (1) and (2)

$C_2\, g(n) \leq f(n) \leq C_{1g(n)}$ for all $n \geq n_0$ -------(3)

(i.e) Θ(g(n)) = O(g(n))Ω(g(n))

From (3) f(n) = Θ(g(n)) hence proved

67. Define recursion

A function may be recursively defined in terms of itself. A familiar example is the Fibonacci number sequence: F(n) = F(n − 1) + F(n − 2).

For such a definition to be useful, it must be reducible to non-recursively defined values: in this case F(0) = 0 and F(1) = 1. ccurs when a thing is defined in terms of itself or of its type. Recursion is used in a variety of disciplines ranging from linguistics to logic.

The most common application of recursion is in mathematics and computer science, where a function being defined is applied within its own definition.

While this apparently defines an infinite number of instances (function values), it is often done in such a way that no loop or infinite chain of references can occur.

68. List the reasons for choosing an approximate algorithm.

Approximation algorithms are typically used **when finding an optimal solution is intractable**, but can also be used in some situations where a near-optimal solution can be found quickly and an exact solution is not needed. Many problems that are NP-hard are also non-approximable assuming P≠NP.

PART – B

1. Explain the notion of an algorithm with diagram. *May2014*

Synopsis:

- Introduction
- Definition
- Diagram
- Characteristics of an Algorithm / Features of an Algorithm
- Rules for writing an Algorithm
- Implementation of an Algorithm
- Order of an Algorithm
- Program
- Example : GCD

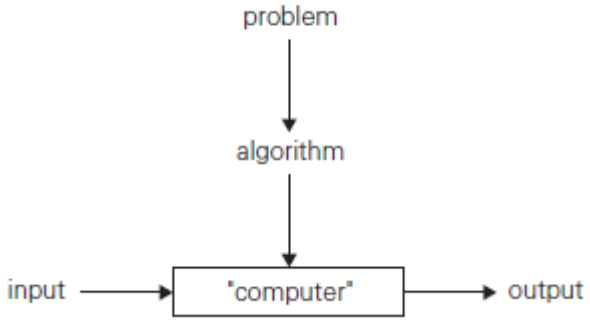
Introduction:

- An algorithm is a sequence of finite number of steps involved to solve a particular problem.
- An input to an algorithm specifies an instance of the problem the algorithm solves.
- An algorithm can be specified in a natural language or in a pseudo code.
- Algorithm can be implemented as computer programs.
- The same algorithm can be represented in several different ways.
- Several algorithms for solving the same problem may exist.
- Algorithms for the same problem can be based on different ideas and can solve the problem with dramatically different speeds.

Definition:

- An algorithm is a sequence of non ambiguous instructions for solving a problem in a finite amount of time.
- Each algorithm is a module, designed to handle specific problem.
- The non ambiguity requirement for each step of' an algorithm cannot be compromised.
- The range of inputs for which an algorithm works has to be specified carefully.

Diagram:



Characteristics of an algorithm / Features of an Algorithm

The important and prime characteristics of an algorithm are,

- ✓ **Input:**Zero or more quantities are externally supplied.
- ✓ **Output:**At least one quantity is produced.
- ✓ **Definiteness:**Each instruction is clear and unambiguous.
- ✓ **Finiteness:**For all cases the algorithm terminates after a finite number of steps.
- ✓ **Efficiency:**Every instruction must be very basic.
- ✓ An algorithm must be expressed in a fashion that is completely free of ambiguity.
- ✓ It should be efficient.
- ✓ Algorithms should be concise and compact to facilitate verification of their correctness.

Writing an algorithm

- Algorithm is basically a sequence of instructions written in simple English language.

- The algorithm is broadly divided into two sections

Algorithm heading
It consists of name of algorithm, problem description , input and output.
Algorithm Body
It consists of logical body of the algorithm by making use of various programming constructs and assignment statement.

Rules for writing an algorithm.

Algorithm is a product consisting of heading and body. The heading consists of keyword **algorithm** and name of the algorithm and parameter list. The syntax is

Algorithm name (p1, p2,.....pn)

1. Then in the heading section we should write following things :
   // Problem Description;
   // Input:
   //Output:
2. Then body of an algorithm is written, in which various programming constructs like if , for , while or some assignment statement may be written.
3. The compound statements should be enclosed within { and } brackets.
4. Single line comments are written using // as beginning of comment.
5. The **identifier** should begin by latter and not by digit. An identifier can be a combination of alphanumeric string.
   - It is not necessary to write data types explicitly for identifiers. It will be represented by the context itself.
   - Basic data types used are integer, float, and char, Boolean and so on.
   - The pointer type is also used to point memory locations.
   - The compound data type such as structure or record can also be used.
6. Using assignment operator ← an assignment statement can be given.
   For instance: **Variable ← expression**
7. There are other types of operators' such as Boolean operators such as true or false. Logical operators such as AND, OR, NOT. And relational operators such as < , <= , >, >=, = , !=.
8. The array indices are stored with in square brackets '[' ']'. The index of array usually starts at zero. The multidimensional arrays can also be used in algorithm.
9. The inputting and outputting can be done using read and write.
   For example:
   Write ("this message will be displayed on console ");
   Read (Val);
10. The conditional statements such as if –then – else are written in following form
    If (condition) then statement
    If (condition) then statement else statement
    If the **if – then** statement is of compound type then {and} should be used for enclosing block
11. While statement can be written as :

```
While (condition)do
{
        Statement 1
        Statement 2
          :
```

Statement n

}

While the condition is true the block enclosed with { } gets executed otherwise statement after} will be executed.

12. The general form for writing for loop is :

For variable ← value$_1$ to value$_n$ do

{

Statement 1

Statement 2

:

Statement n

}

Here value$_1$ is initialization condition and value$_n$ is a terminating condition the step indicates the increments or decrements in value$_1$ for executing the for loop.

Sometime a keyword step is used to denote increment or decrement the value of variable for example

For i ← 1 to n step 1 ←

Here variable i is incremented by 1 at each iteration

{

Write (i)

}

13. The **repeat – until** statement can be written as

Repeat

Statement 1

Statement 2

:

Statement n

Until (condition)

14. The **break** statement is used to exit from inner loop. The return statement is used to return control from one point to another. Generally used while exiting from function

Note: The statements in an algorithm executes in sequential order i.e. in the same order as they appear – one after the other

Example 1 : Write an algorithm to count the sum of n numbers

```
Algorithm  sum (1, n)
        //Problem description : this algorithm is for finding the
        //sum of given n numbers
        //Input: 1 to n numbers
        //Output: the sum of n numbers
    Result ← 0
     For i 1 to n do
          i  ← i+1
    Result ← result + i
 Return result
```

Example 2: Write an algorithm to check whether given number is even or odd.

```
Algorithm eventest ( val)
        //Problem description : this algorithm test whether given
        //number is even or odd
        //Input: the number to be tested i.e .val
        //Output: appropriate messages indicating even or odd
```

```
If (val % 2 = 0) then
        Write ("given number is even ")
Else
        Write ("given number is odd")
```

Example 3: Write an algorithm for sorting the elements.

```
Algorithm sort (a, n)
        //Problem description: sorting the elements in ascending
        //order
        //Input: an array in which the elements in ascending order
        //is total number of elements in the array
        //Output: the sorted array
        For i 1 to n do
                For j i + 1 to  n-1 do
                        If (a[i]>a[j]) then
                        {
                                temp ← a[i]
                                a[i] ←a[j]
                                a[j] ←temp
                        }
        Write ( " list is sorted ")
```

Example 4: Write an algorithm to find factorial of n number.

```
Algorithm fact (n)
        //Problem description: this algorithm finds the factorial.
        //for given number n
        //Input : the number n of which the factorial is to be
        //calculated.
        //Output : factorial value of given n number.
        If( n ← 1) then
                Return 1
        Else
                Return n * fact(n-1)
```

Example 5:

Write an algorithm to perform multiplication of two matrices

```
Algorithm mul (A, b, n)
        //Problem description: this algorithm is for computing
        //multiplication of two matrices
        //Input : the two matrices A, B and order of them as n
        //Output : The multiplication result will be in matrix c
        For i ← 1 to n do
                For j ← 1 to n do
                        C [i,j] ← 0

                For k ← 1 to n do
                        C[I ,j ] ←c[i, j] +A[i,k]B[k,j]
```

Implementation of algorithms

An algorithm describes what the program is going to perform. It states some of the actions to be executed and the order in which these actions are to be executed.

The various steps in developing algorithm are,

1. Finding a method for solving a problem. Every step of an algorithm should be in a precise and in a clear manner. Pseudo code is also used to describe the algorithm.
2. The next step is to validate the algorithm. This step includes, all the algorithm should be done manually by giving the required input, performs the required steps including in the algorithm and should get the required amount of output in an finite amount of time.
3. Finally, implement the algorithm in terms of programming language.

Order of an algorithm

The order of an algorithm is a standard notation of an algorithm that has been developed to represent function that bound the computing time for algorithms. It is an order notation. It is usually referred as O-notation.

Example

Problem size = 'n'
Algorithm = 'a' for problem size n
The document mechanism execution = Cn^2 times
where C – constant
Then the order of the algorithm 'a' = $O(n^2)$
where n^2 = Complexity of the algorithm 'a'.

Program

- A set of explicit and unambiguous instructions expressed using a programming languages constructs is called a program.
- An algorithm can be converted into a program, using any programming language. Pascal, Fortran, COBOL, C and C++ are some of the programminglanguages.

Difference between program and algorithm:

Sno	Algorithm	Program
1	Algorithm is finite.	Program need to be finite.
2	Algorithm is written using natural language or algorithmic language.	Programs are written using a specific programming language.

1.A. write an algorithm using recursion that determines the GCD of two numbers.Determine the time and space complexity Nov/Dec 2019

Example : Calculating Greatest common Divisor

The Greatest common Divisor (GCD) of two non zero numbers a and b is basically the largest integer that divides both a and b evenly i.e with a remainder of zero.
GCD using three methods

1. Euclid's algorithm
2. Consecutive integer checking algorithm
3. Finding Gusing repetitive factors

Euclid's algorithm to compute Greatest Common Divisor (GCD) of two non negative integers.

Euclid's algorithm is based on applying related equality

gcd (m, n) = gcd (n, m mod n) until the m and n is equal to 0

Where m mod n is the remainder of the division of m by n

Step 1: Start
Step 2: If n = 0, return the value of m as the answer and stop,
otherwise proceed to step 3.

Step 3: Divide m by n and assign the value of the remainder to r.
Step 4: Assign the value of n to m and the value of r to n. Goto step 2
Step 5: Stop

ALGORITHM *Euclid*(m, n)

//Computes gcd(m, n) by Euclid's algorithm
//Input: Two nonnegative, not-both-zero integers m and n
//Output: Greatest common divisor of m and n
while $n \neq 0$ **do**
$r \leftarrow m \bmod n$
$m \leftarrow n$
$n \leftarrow r$
return m

Example, gcd (60,24) can be computed as follows,

gcd (60,24)	gcd (m, n) m =60, n=24; m/n = 2 (remainder 12) n=m=24 r=n=12
gcd (24, 12)	m/2 = 2 (remainder 0) n=m=12 r=n=0
gcd (12, 0) =12	

Hence, gcd(60, 24) = gcd(24,l2)=gcd(12,0)=12

2. Consecutive integer checking algorithm

In this method while finding the GCD of a and b we first of all find the minimum value of them. Suppose if , value of b is minimum then we start checking the divisibility by each integer which is lesser than or equal to b.

Example:

a = 15 and b =10 then
t= min(15,10)
since 10 is minimum we will set value of t = 10 initially.

Consecutive integer checking algorithm for computing gcd(m, n)

Step 1: Start
Step 2: Assign the value of mini {m, n} to t
Step 3: Divide m by t. If the remainder of this division is 0, go to step 4,
Otherwise goto step 5.
Step 4: Divide n by t. If the remainder of this division is 0, return the value
of t as the answer and stop. Otherwise proceed to step 5.
Step 5: Decrease the value of t by I. Go to step 3.
Step 6: Stop

Algorithm GCD intcheck (a,b)

```
//Problem description : this algorithm computes the GCD of //two numbers a and b using consecutive integer checking
//method
//Input : two integers a and b
//Output: GCD value of a and b
 t ← min ( a, b)
while (t>=1) do
{
        If ( a mod t == 0 AND b mod t == 0) then
                Return t
```

```
        Else
                t ← t-1
    }
    Return 1
```

3. Finding GCD using repetitive factors

The third procedure for finding the greatest common divisor is middle school procedure.

Middle School Method

For the numbers 60 and 24

```
2 | 60          2 | 24
  | 30            | 12
3 | 15          2 |  6
  |  5            |  3
```

60=2x2x3 x5
24=2x2x3 x2
gcd (60,24) =2x2x3 =12

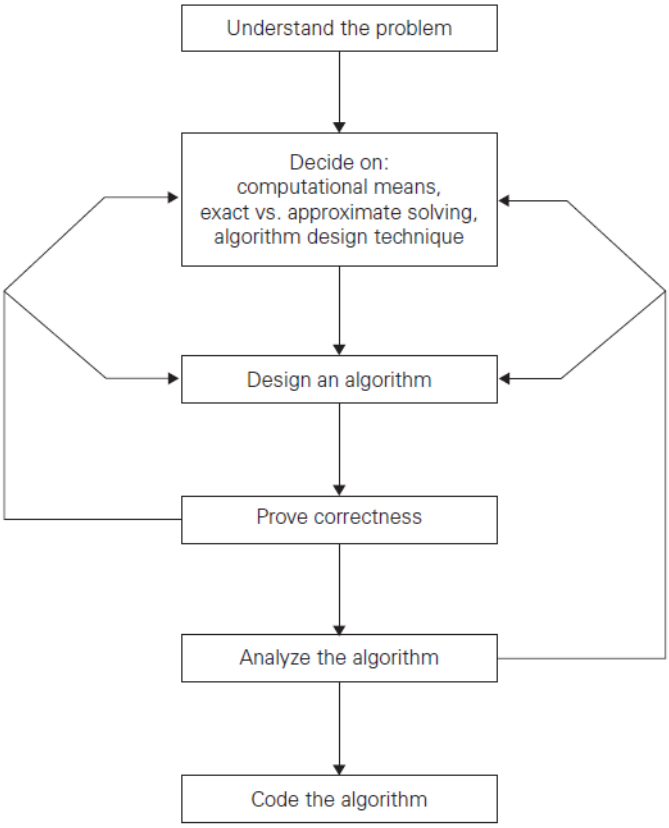
Algorithm:

Step 1: Start
Step 2: Find the prime Factor of m.
Step 3: Find the prime factors of n.
Step 4: Identify all the common factors in the two prime expressions Found in step 2 and step 3. If' P is a common factor occurring pm and pn times in m and n respectively. It should be repeated min (pm, and pn) times.
Step 5: Compute the product of the all the common factors and return it as the greatest common divisor of the numbers given.
Step 6: Stop.

2. Explain the Fundamentals of Algorithmic problem solving. Or explain the steps involved in problem solving *May 2014* ,April/May 2019

Sequential steps in designing and analysing an algorithm

1. Understanding the problem.
2. Ascertaining the capabilities of a computational device.
3. Choosing between exact and approximate problem solving.
4. Deciding on appropriate data structures.
5. Algorithm Design Techniques.
6. Methods of specifying an algorithm
7. Proving an algorithm's correctness.
8. Analysing an algorithm.
9. Codinganalgorithm.



1.Understanding the problem:

- ✓ To design an algorithm, understand the problem completely by reading the problem's description carefully.
- ✓ Read the problem description carefully and clear the doubts.
- ✓ Specify exactly the range of inputs the algorithm need to handle.
- ✓ Once the problem is clearly understandable, then determine the overall goals but it should be in a precise manner.
- ✓ Then divide the problem into smaller problems until they become manageable size.

2. Ascertaining the capabilities of a computational device

Sequential Algorithm:

- ✓ Instructions are executed one after another, one operation at a time.
- ✓ This is implemented in RAM model.

Parallel Algorithm:

- ✓ Instructions are executed in parallel or concurrently.

3. Choosing between exact and appropriate problem solving

- ✓ The next principal decision is to choose between solving the problem exactly or solving the problem approximately.
- ✓ The algorithm used to solve the problem exactly called **exact algorithm**.
- ✓ The algorithm used to solve the problem approximately is called **approximation algorithm.**

Reason to choose approximate algorithm

- There are important problems that simply cannot be solved exactly such as
  - Extracting square roots.
  - Solving non linear equations.
  - Evaluating definite integrals.
- ✓ Available algorithms for solving problem exactly can be unacceptably slow, because

of the problem's intrinsic complexity. Ex: Travelling salesman problem

4. Deciding on appropriate data structures

Data structure is important for both design and analysis of algorithms.

Algorithm + Data Structures = Programs.

In Object Oriented Programming, the data structure is important for both design and analysis of algorithms.

The variability in algorithm is due to the data structure in which the data of the program are stored such as

1. How the data are arranged in relation to each other.
2. Which data are kept in memory
3. Which data are kept in files and how the files are arranged.
4. Which data are calculated when needed?

5. Algorithm Design Techniques

An algorithm design techniques or strategy or paradigm is general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.

Uses

- ✓ They provide guidance for designing algorithms or new problems.
- ✓ They provide guidance to problem which has no known satisfied algorithms.
- ✓ Algorithm design technique is used to classify the algorithms based on the design idea.
- ✓ Algorithm design techniques can serve as a natural way to categorize and study the algorithms.

6. Methods of specifying an algorithm

There are two options, which are widely used to specify the algorithms.

They are

- o Pseudo code
- o Flowchart

Pseudo code

- o A pseudo code is a mixture of natural language and programming language constructs.
- o A pseudo code is more precise than a natural language
- o For simplicity, declaration of the variables is omitted.
- o For, if and while statements are used to show the scope of the variables.
- o "←" (Arrow) - used for the assignment operation.
- o "//" (two slashes) - used for comments.

Flow chart

- o It is a method of expressing an algorithm by a collection of connected geometric shapes containing description of the algorithms steps.
- o It is very simple algorithm.
- o This representation technique is inconvenient.

7. Proving an Algorithm's correctness

Once an algorithm has been specified, then its correctness must be proved.

- ✓ An algorithm must yield a required result for every legitimate input in a finite amount of time.
- ✓ A mathematical induction is a common technique used to prove the correctness of the algorithm.
- ✓ In mathematical induction, an algorithm's iterations provide a natural sequence of steps needed for proofs.
- ✓ If the algorithm is found incorrect, need to redesign it or reconsider other decisions.

8. Analysing an algorithm

- ✓ Efficiency of an algorithm is determined by measuring the time, space and amount of

resources, it uses for executing the program.

- ✓ The efficiency of the algorithm is determined with respect to central processing units time and internal memory.
- ✓ There are two types of algorithm efficiency.

They are

- Time efficiency (or) Time Complexity
- Space efficiency (or) Space Complexity

Time Efficiency / Time Complexity

- ✓ Time efficiency indicates how fast the algorithm runs.
- ✓ The time taken by a program to complete its task depends on the number of steps in an algorithm.
- ✓ The time required by a program to complete its task will not always be the same.
- ✓ It depends on the type of problem to be solved.

It can be of two types.

- Compilation Time
- Run Time (or) Execution Time

- ✓ The time (T) taken by an algorithm is the sum of the compile time and execution time.

Compilation Time

- ✓ The amount of time taken by the compiler to compile an algorithm is known as compilation time.
- ✓ During compilation time, it does not calculate the executable statements, it calculates only the declaration statements and check for any syntax and semantic errors.
- ✓ The different compilers can take different times to compile the same program.

Execution Time

- ✓ The execution time depends on the size of the algorithm.
- ✓ If the number of instructions in an algorithm is large then the run time is also large.
- ✓ If the number of instructions in an algorithm is small then the time need to execute the program is small.
- ✓ The execution time is calculated for executable statements and not for the declaration statements.
- ✓ The complexity is normally expressed as an order of magnitude.
- ✓ Example: O (n^ 2)
- ✓ The time complexity of a given algorithm is defined as computation of function f() as a total number of statements that are executed for computing the value f(n).
- ✓ The time complexity is a function which depends on the value of n.

The time complexity can be classified as 3 types.

They are

1. Worst Case analysis
2. Average Case analysis
3. Best Case analysis

Worst Case Analysis

- ✓ The worst case complexity for a given size corresponds to the maximum complexity encountered among all problem of the same size.
- ✓ Worst case complexity takes a longer time to produce a desired result.

This can be represented by a function f(n).

$f(n) = n^2$ or $n \log n$

Average Case Analysis

- ✓ The average case analysis is also known as the expected complexity which gives measure of the behaviour of an algorithm averaged over all possible problem of the same size.
- ✓ Average case is the average time taken by an algorithm for producing a desired output.

Best Case Analysis

- ✓ Best case is a shortest time taken by an algorithm to produce the desired result.

Space Complexity

- ✓ Space efficiency indicates how much extra memory the algorithm needs.
- ✓ The amount of storage space taken by the algorithm depends on the type of the problem to be solved.
- ✓ The space can be calculated as,
- ✓ A fixed amount of memory occupied by the space for the program code is space occupied by the variable used in the program.
- ✓ A variable amount of memory occupied by the component variable dependent on the problem is being solved.
- ✓ This space is more or less depending upon whether the program uses iterative or recursive procedures.

There are three different space considered for determining the amount of memory used by the algorithm.

They are

- o Instruction Space
- o Data Space
- o Environment Space

Instruction Space

- ✓ When the program gets compiled, then the space needed to store the compiled instruction in the memory is called instruction space.
- ✓ The instruction space independent of the size of the problem

Data Space

- ✓ The memory space used to hold the variables of data elements are called data space.
- ✓ The data space is related to the size of the Problem

Environment Space

- ✓ It is the space in memory used only on the execution time for each Function call.
- ✓ It maintains runtime stack in that it holds returning address of the previous functions.
- ✓ Every function on the stack has return value and a pointer on it.

Characteristics of an algorithms

- o Simplicity
- o Generality

Simplicity

- o Simpler algorithms are easier to understand.
- o Simpler algorithms are easier to program.
- o The resulting programs contains only few bugs.
- o Simpler algorithms are more efficient compared to the complicated alternatives.

Generality

- o The characteristic of an algorithm generality has two issues.
- o They are
  - ▪ Generality' of the problem the algorithm solves.
  - ▪ Range of inputs it accepts.

9. Coding an Algorithm

- ✓ Implementing an algorithm correctly is necessary but not sufficient to diminish the algorithm's power by an inefficient implementation.
- ✓ The standard tricks such as computing a loop's invariant (an expression that does not change its value) outside the loop, collecting common sub expressions, replacing expensive operations by cheaper ones and so on should be known to the programmers such factors can speed up a program only by a constant factor, where as a better algorithm can make a difference in running time by orders of magnitude.

- ✓ Once an algorithm has been selected, a 10-50% speed up may be worth an effort.
- ✓ An algorithm's optimality is not about the efficiency of an algorithm but about the complexity of the problem it solves.

3. Explain the important problem types.

Some of the most important problem types are

1. Sorting
2. Searching
3. String Matching (or) String processing
4. Graph Problems
5. Combinatorial problems
6. Geometric problems
7. Numerical Problems

1. Sorting

- ✓ Sorting means arranging the elements in increasing order or in decreasing order.
- ✓ The sorting can be done on numbers , characters (alphabets), string or employees record.
- ✓ Many algorithms are used to perform the task of sorting.
- ✓ Sorting is the operation of arranging the records of a table according to the key value of the each record.
- ✓ A table of a file is an ordered sequence of records r[l], r[2].. r[n] each containing a key k[l], k[2]....k[n]. The table is sorted based on the key.

Properties of Sorting Algorithms

The two properties of Sorting Algorithms are

1. Stable
2. In-place

Stable:

- ✓ A sorting algorithm is called **stable**, if it preserves the relative order of any two equal elements in its input.
- ✓ In other words, if an input list contain two equal elements in positions i and j, where i<j, then in the sorted list they have to be in position i' and j' respectively, such that i' < j'

In-place

- ✓ An algorithm is said to be **in-place** if it does not require extra memory, except, possibly for a few memory units.

The important **criteria for the selection of a sorting** method for the given set of data items are as follows.

1. Programming time of the sorting algorithm.
2. Execution time of the program
3. Memory space needed for the programming environment

The **main objectives** involved in the design of sorting algorithms are

1. Minimum number of exchanges.
2. Large volume of data blocks movement.

Types of Sorting

The two major classification of sorting methods are

1. Internal Sorting methods
2. External Sorting methods

Internal Sorting

- ✓ The key principle of internal sorting is that all the data items to be sorted are retained in the main memory and random access memory.
- ✓ This memory space can be effectively used to sort the data items.
- ✓ The various internal sorting methods are

1. Bubble sort
2. Selection sort
3. Shell sort

4. Insertion sort
5. Quick sort
6. Heap sort

External Sorting

- ✓ The idea behind the external sorting is to move data from secondary storage to mail memory in large blocks for ordering the data.
- ✓ The most commonly used external sorting method is merge sort.

2. Searching

- ✓ One of the important applications of array is searching
- ✓ Searching is an activity by which we can find out the desired element from the list. The element which is to be searched is called **search key**
- ✓ There are many searching algorithm such as sequential search , Fibonacci search and more.

Searching in dynamic set of elements

- ✓ There may be of elements in which repeated addition or deletion of elements occur.
- ✓ In such a situation searching an element is difficult.
- ✓ To handle such lists supporting data structures and algorithms are needed to make the list balanced (organized)

3. String processing

A **string is a collection** of characters from an alphabet.

Different type of strings are

- o Text string
- o Bit string

Text String It is a collection of letters, numbers and special characters.

Bit String It is collection of zeros and ones.

- Operations performed on a string are
  1. Reading and writing strings
  2. String concatenation
  3. Finding string length
  4. String copy
  5. String comparison
  6. Substring operations
  7. Insertions into a string
  8. Deletions from a string
  9. Pattern matching

Pattern Matching or String matching

The process of searching for an occurrence of word in a text is called Pattern matching.

Some of the algorithms used for pattern matching are

1. Simple pattern matching algorithm
2. Pattern matching using Morris Pratt algorithm
3. Pattern matching using Knuth-Morris-Pratt algorithm

4. Graph Problems

- ✓ Graph is a collection of vertices and edges.
- ✓ Formally, a graph G={ V, E } is defined by a pair of two sets.
- ✓ A finite set V of items called Vertices and a set E of pairs of these items called edges.
- ✓ If the pairs of vertices are ordered, then G is called a directed graph because every edge is directed.
- ✓ In a directed graph the direction between two nodes are not same G(V,W)!=G(W,V)
- ✓ If the pair of the vertices are unordered then G is called an undirected graph.
- ✓ In undirected graph, the edges has no specific direction.
- ✓ The graph problems involve graph traversal algorithms, shortest path algorithm and topological sorting and so on. Some graph problems are very hard to solve.
- ✓ For example travelling salesman problem, graph colouring problems

5. Combinatorial Problems

- ✓ The travelling salesman problem and the graph colouring problems are examples of combinatorial problems.
- ✓ A combinatorial object such as a permutation a combination or a subset that satisfies certain constraints and has some desired property such as maximizes a value or minimizes a cost should be find.
- ✓ Combinatorial problems are the **most difficult problems**.

The reason is,

1. As problem size grows the combinatorial objects grow rapidly and reach to huge value. size.
2. There is no algorithms available which can solve these problems in finite amount of time
3. Many of these problems fall in the category of unsolvable problem.

Some combinatorial problems can be solved by efficient algorithms.

6. Geometric Problems

- ✓ Geometric algorithms deal with geometric objects such as points ,lines and polygons.
- ✓ The procedure for solving a variety of geometric problems includes the problems of constructing simple geometric shapes such as triangles, circles and so on.

The two classic problems of computational geometry are the

1. Closest pair problem
2. Convex hull problem

- ✓ The closest pair problem is self explanatory. Given n points in the plane, find the closest pair among them.
- ✓ The convex hull problem is used to find the smallest convex polygon that would include all the points of a given set.
- ✓ The geometric problems are solved mainly in applications to computer graphics or in robotics

6.Numerical problems

- ✓ Numerical problems are problems that involve mathematical objects of continuous nature such as solving equations and systems of equations computing definite integrals evaluating functions and so on.
- ✓ Most of the mathematical problems can be solved approximate algorithms.
- ✓ These algorithms require manipulating of the real numbers; hence we may get wrong output many times.

3.Explain the fundamentals of the analysis framework. Or explain time-space trade off of the algorithm designed. April/May 2019

- Efficiency of an algorithm can be in terms of time or space.
- This systematic approach is modelled by a frame work called as analysis frame work.

Analysis framework

- The efficiency of an algorithm can be decided by measuring the performance of an algorithm.
- The performance of an algorithm is computed by two factors
  - amount of time required by an algorithm to execute
  - amount of storage required by an algorithm

Overview

- Space complexity
- Time complexity
- Measuring an Input's size
- Measuring Running Time
- Orders of Growth

Space complexity

- The space complexity can be defined as amount of memory required by an algorithm to run.

- To compute the space complexity we use two factors: constant and instance characteristics.

- The space requirement S(p) can be given as **S(p) = C+ S(p)**
  Where C is a constant i.e. fixed part and it denotes the space of inputs and outputs.

Time complexity

- The time complexity of an algorithm is the amount of computer time required by an algorithm to run to completion.
- For instance in multiuser system, executing time depends on many factors such as
  - System load
  - Number of other programs running
  - Instruction set used
  - Speed underlying hardware
- The time complexity is therefore given in term of frequency count
- Frequency count is a count denoting number of times of execution of statement

Example

```
For (i=0; i<n; i++)
{
        sum = sum + a[i];
}
```

Statement	**Frequency count**
i=0	1
i<n	This statement executes for (n+1) times. When conditions is true i.e. when i<n is true , the execution happens to be n times , and the statement execute once more when i<n is false
i++	n times
sum = sum + a[i]	n times
Total	**3n + 2**

Measuring an Input's size

- All algorithms run longer on larger inputs.
- Ex: Sorting larger arrays, multiply larger matrices etc.
- Investigates an algorithm efficiency as a function of some parameter n indicating the algorithm input size.
- Example:
  - In problem of evaluating a polynomial $p(x) = a_n x^n + \ldots + a_0$ of degree n, the parameter will be the polynomial's degree or the number of its coefficients which is larger by one than its degree.
- In spell checking algorithm,
  - If algorithm examines the individual character of its input, then the size of the input is the no. of characters.
  - If the algorithm processes the word, the size of the input is the no. of words.

Measuring Running Time

- Some units of time measurement such as a second, a millisecond and so on can be used to measure the running time of a program implementing the algorithm.
- **Drawbacks**
  1. Dependence on the speed of a particular computer
  2. Dependence on the quality of a program implementing the algorithm.
  3. The compiler used in generating the machine code.

4. The difficulty of clocking the actual running time of the program

- Since we are in need to measure an algorithm's efficiency, we should have a metric that does not depend on these factors.
- One possible approach is to count the number of times of the algorithm's operations is executed. But this approach is difficult and unnecessary.
- The main objective is to identify the most important operation of the algorithm, called the **Basic Operation -** the operation contributing the most to the total running time, and compute the number of times the basic operation is executed.
- It is not so difficult to identify the basic operation of an algorithm: it is usually the most time consuming operation in the algorithm's innermost loop.

Example

- Most sorting algorithms work by comparing the elements (keys) of a list being sorted with each other. For such algorithms the basic operation is a Key Comparison.

Problem statement	**Input Size**	**Basic operation**
Searching a key element from the list of n elements	List of n elements	Comparison of key with every element of list
Performing matrix multiplication	The two matrixes with order n×n	Actual multiplication of the elements in the matrices
Computing GCD of two numbers	Two numbers	Division

The formula to compute the execution time using basic operation is

. $\mathbf{T(n) \approx C_{op}\, C(n)}$

Where T(n) – running time

C(n) – no. of times this operation is executed.

Cop – execution time of algorithms basic operation.

Orders of Growth

- Measuring the performance of an algorithm in relation with the input size n is called order of growth.

Worst Case, Best Case and Average Case efficiencies

- It is reasonable to measure an algorithm's efficiency as a function of a parameter indicating the size of the algorithm's input.
- But for many algorithms the running time depends not only on an input size but also on the specifics of a particular input.

Example: **Sequential Search or Linear Search AU: Dec -11, Marks 10**

ALGORITHM *SequentialSearch(A*[0..*n* − 1], *K)*

```
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n − 1] and a search key K
//Output: The index of the first element in A that matches K
// or −1 if there are no matching elements
    i ←0
    while i < n and A[i] _= K do
        i ←i + 1
    if i < n return i
    else return -1
```

- This algorithm searches for a given item using some search key K in a list of 'n' elements by checking successive elements of the list until a match with the search key

is found or the list is exhausted.

- The algorithm makes the largest number of key comparisons among all possible inputs of size n: $C_{worst}(n)=n$

Worst case efficiency

- The worst case efficiency of an algorithm is its efficiency for the worst case input of size n, which is an input (or inputs) of size n. For which the algorithm runs the longest among all possible of that size.
- The way to determine the worst case efficiency of an algorithm is that:
  - Analyse the algorithm to see what Kind of inputs yield the largest value of the basic operations count C(n) among all possible inputs of size n and then compute is w value **$C_{worst} = (n)$.**

Best case efficiency

- The best case efficiency of an algorithm is its efficiency for the best case input of size n, which is an input (or inputs) of. size n for which the algorithm runs the fastest among all possible inputs of that size.
- The way to determine the best case efficiency of an algorithm is as follows.
  - First, determine the kind of inputs of size n.
  - Then ascertain the value of C(n) on these inputs.
- **Example:** For sequential search, the best case inputs will be lists of size 'n' with their first elements equal to a search key: **$C_{best}(n) = 1$.**

Average case efficiency

- It yields the necessary information about an algorithm's behaviour on a "typical" or "random" input.
- To determine the algorithm's average case efficiency some assumptions about possible inputs of size 'n'.
- The average number of key comparisons C_{avg} (n) can be computed as follows:
  - In case of a successful search the probability of the first match occurring in the position of the list is p/n for every i. and the number of comparisons made by the algorithm in such a situation is obviously 'i'.
  - In case of an unsuccessful search, the number of comparisons is 'n' with the probability of such a search being (1-p). Therefore,

$$C_{avg}(n)=[1.\frac{p}{n}+2.\frac{p}{n}+.....i.\frac{p}{n}+...n.\frac{p}{n}]+n.(1-p)$$

$$= \frac{p}{n}[1+2+3+....+i+...+n]+n(1-p)$$

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

$$\mathbf{C_{avg}(n)= \frac{p(n+1)}{2n} + n(1-p)}$$

There may be n elements at which chances of 'not getting element' are possible. Hence n . (1-p)

Example:

- If p = 1 (i.e.) if the search is successful, then the average number of key comparisons made by sequential search is (n+1)/2.
- If p = 0 (i.e.) if the search is unsuccessful, then the average number of key comparisons will be 'n' because the algorithm will inspect all n elements on all such inputs.

4. Explain the Asymptotic Notations and its properties? Or explain briefly Big oh notation , Omega notation and Theta notation give an example (Apr/May-2017) or what are the Rules of Manipulate Big-Oh Expression and about the typical growth rates of

algorithms? Nov/Dec 2017 Nov/Dec 2018

Define Big O notation, Big Omega and Big Theta Notation. Depict the same graphically and explain. May/June 2018 , Nov/Dec 2019

Explain the importance of asymptotic analysis for running time of an algorithm with an example. (April/May 2021)

Asymptotic notations are mathematical tools to represent time complexity of algorithms for measuring their efficiency. Types :

- Big Oh notation - 'O'
- Omega notation - 'Ω'
- Theta notation - 'Θ'
- Little Oh notation - 'o '

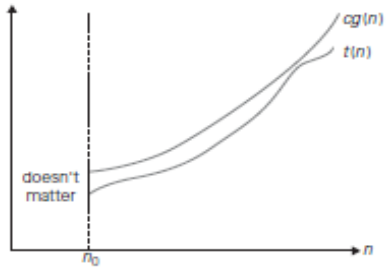
Big Oh notation (O)

- The big oh notation is denoted by **'O'.**
- It is a method of representing the **upper bound of algorithm's running time.**
- Using big oh notation we can give **longest amount of time taken** by the algorithm to complete.

Definition

A function t(n) is said to be in O(g(n)) **(t(n) Є O(g(n))),** if t(n) is bounded above by constant multiple of g(n) for all values of n, and if there exist a positive constant c and non negative integer n0 such that

- **t(n) ≤ c*g(n)** for all n ≥ n0.



- 1 Big-oh notation: $t(n) \in O(g(n))$.

Example 1:

Consider function t(n) = 2n + 2 and g(n) = $n^{2.}$ Then we have to find some constant c, so that f(n) ≤ c*g(n).

As t(n) = 2n + 2 and g(n) = $n^{2.}$ Then we find c for n=1 then

t(n) = 2n + 2
= 2(1) +2
t(n) = 4

And g(n) = n^2
= $(1)^2$
g(n) = 1

i.e t(n) > g(n)

if n = 2 then,

t(n) = 2n + 2
= 2(2) +2
t(n) = 6

And g(n) = n^2
= $(2)^2$
g(n) = 4

i.e t(n) > g(n)

if n = 3 then,

t(n) = 2n + 2

= 2(3) +2

t(n) = 8

And g(n) = n^2

= $(3)^2$

g(n) = 9

i.e t(n) < g(n) is true.

Hence we can conclude that for n> 2, we obtain

t(n) < g(n)

Thus always upper bound of existing time is obtained by big oh notation.

Omega Notation (Ω)

Omega notation is denoted by '**Ω**'.

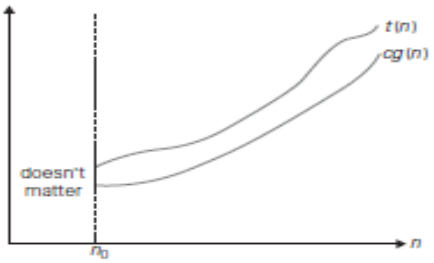
This notation is used is to represent the **lower bound of algorithm's** running time.

Using omega notation we can denote **shortest amount of time taken** by algorithm.

Definition

A function t(n) is said to be in Ω(g(n)) **(t(n) Є Ω(g(n))),** if t(n) is bounded below by constant multiple of g(n) for all values of n, and if there exist a positive constant c and non negative integer n0 such that

- **t(n) ≥ c*g(n)** for all n ≥ n0.



- Big-omega notation: $t(n) \in \Omega(g(n))$.

- **Example 1:**

Consider t(n)=$2n^2$ + 5 and g(n) = 7n

Then if n = 0

t(n) = 2 $(0)^2$ + 5

= 5

g(n) = 7(0)

= 0 i.e t(n) > g(n)

But if n = 1

t(n) = 2 $(1)^2$ + 5

= 7

g(n) = 7(1)

= 7 i.e t(n) = g(n)

But if n = 2

t(n) = 2 $(2)^2$ + 5

= 9

g(n) = 7(2)

= 12 i.e t(n) < g(n)

But if n = 3

t(n) = 2 $(3)^2$ + 5

= 18 + 5

= 23

g(n) = 7(3)

= 21 i.e t(n) > g(n)

Thus for n>3 we get t(n) > c * g(n).

It can be represented as

$2n^2 + 5 \in \Omega(n)$

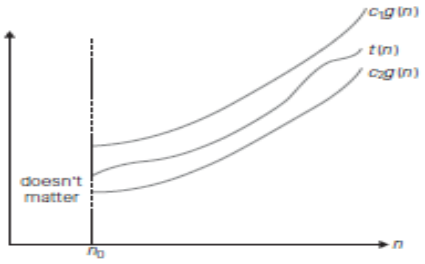
Theta Notation (Θ)

The theta notation is denoted by Θ. By this method the **running time is between upper bound and lower bound.**

Definition

A function t(n) is said to be in Θ(g(n)) **(t(n) Є Θ(g(n))),** if t(n) is bounded both above and below by constant multiple of g(n) for all values of n, and if there exist a positive constant c1 and c2 and non negative integer n0 such that

- **C2*g(n) ≤ t(n) ≤ c1*g(n)** for all n ≥ n0.



- 3 Big-theta notation: $t(n) \in \Theta(g(n))$.

Example 1:

If t(n) = 2n + 8 and g(n) = 7n, 5n

Where n ≥ 2

C2*g(n) ≤ t(n) ≤ c1*g(n) for all n ≥

Θ(g(n)) = O(g(n)) Ω(g(n))

(t(n) Є Θ(g(n)))

Similarly **t(n) = 2n + 8**

g(n) = 7n

g(n) = 5n

i.e **5n < 2n + 8 < 7n** for n ≥ 2

Here $c_2 = 5$ and $c_1 = 7$ with $n_0 = 2$

Little oh notation(o)

The function t(n) = o(g(n)), if O(*g*(*n*)) and t(n) <> Ω(*g*(n))

Example

t(n) = 3n+2

Where n>0, $3n+2 \leq 5\ n^2$

By definition of Big Oh

t(n) = Cg(n)

C = 5; $g(n) = n^2$

But t(n) = 3n+2 < > $\Omega(n^2)$

Therefore t(n) = 3n+2 = $o(n^2)$

Useful property involving the Asymptotic notation:

The following property is useful in analyzing algorithms that comprise two consecutively executed parts.

Theorem

If $t_1(n)$ Є $O(g_1(n))$ and $t_2(n)$ Є $O(g_2(n))$ then,

$t_1(n) + t_2(n)$ Є $(\max \{g_1(n), g_2(n)\})$

Proof

Since $t_1(n)$ Є $O(g_1(n))$, there exist some constant C_1 and some non negative integer n_1 such that

$t_1(n) \leq C_1 (g_1(n))$ for all $n \geq n_1$

Since

$t_2(n) \in O(g_2(n))$

$t_2(n) \leq C_2 (g_2(n))$ for all $n \geq n_2$

Let us denote,

$C_3 = \max \{C_1, C_2\}$ and

Consider $n \geq \max \{n_1, n_2\}$, so that both the inequalities can be used.

The addition of two inequalities becomes,

$$t_1(n) + t_2(n) \leq C_1 (g_1(n)) + C_2 (g_2(n))$$
$$\leq C_3 (g_1(n)) + C_3 (g_2(n))$$
$$\leq C_3\, 2 \max\{g_1(n), (g_2(n))\}$$

Hence,

$$t_1(n) + t_2(n) \in O (\max \{g_1(n), g_2(n)\}),$$

with the constants C and n_0 required by the definition being $2C_3 = 2 \max (C_1, C_2)$ and $\max \{n_1, n_2\}$ respectively.

The property implies that the algorithms overall efficiency will be determined by the part with a larger order of growth.

(i.e.) its least efficient part is

$t_1(n) \in O(g_1(n))$
$t_2(n) \in O(g_2(n))$
} $t_1(n) + t_2(n) \in O (\max \{g_1(n), g_2(n)\})$

Using limits for comparing orders of growth

There are 3 principal cases,

$\lim_{n \to \infty} t(n)/g(n)$

- 0, Implies that (n) has a smaller order of growth than g(n)
- C, Implies that (n) has a same order of growth than g(n)
- ∞, Implies that (n) has a larger order of growth than g(n)

L' Hospital's rule.

$$\lim_{n \to \infty} t(n)/g(n) = \lim_{n \to \infty} t(n)/g(n)$$

Stirling's formula

$n! \approx \sqrt{2\pi n} \left(\frac{n}{k}\right)^n$ for large values of n.

Asymptotic Growth Rate

Three notations used to compare orders of growth of an algorithm's basic operation count

- $O(g(n))$: class of functions $f(n)$ that grow *no faster* than $g(n)$
- $\Omega(g(n))$: class of functions $f(n)$ that grow *at least as fast* as $g(n)$
- $\Theta(g(n))$: class of functions $f(n)$ that grow *at same rate* as $g(n)$

Example

1. Compare orders of growth of $\frac{1}{2}n(n-1)$ and n^2

$$\lim_{n\to\infty}\frac{\frac{1}{2}n(n-1)}{n^2}=\frac{1}{2}\lim_{n\to\infty}\frac{n^2-n}{n}$$

$$=\frac{1}{2}\lim_{n\to\infty}\left(1-\frac{1}{n}\right)$$

$$=\frac{1}{2}$$

Since the limit is equal to a positive constant, the functions have the same order of growth or symbolically

$$\frac{1}{2}n(n-1)\in\theta(n^2)$$

2. Compare orders of growth of $\log_2 n$ and $\sqrt{n}$

$$\lim_{n\to\infty}\frac{\log_2 n}{\sqrt{n}}=\lim_{n\to\infty}\frac{(\log_2 n)'}{(\sqrt{n})'}$$

$$=\lim_{n\to\infty}\frac{(\log_2 e)\frac{1}{n}}{\frac{1}{2\sqrt{2}}}$$

$$=2\log_2 e\lim_{n\to\infty}\frac{\sqrt{n}}{n}$$

$$=0$$

Since the limit is equal to zero, $\log_2 e$ has a smaller order of growth than $\sqrt{n}$ or symbolically.

$$\log_2 e\in o(\sqrt{n}).$$

Little Oh notation is rarely used in analysis of algorithms.

3. Compare orders of growth n! and 2^n.

By using the Stirling's formula,

$$\lim_{n\to\infty}\frac{n!}{2^n}=\lim_{n\to\infty}\frac{\sqrt{2\pi n}\left(\frac{e}{n}\right)}{2^n}$$

$$=\lim_{n\to\infty}\sqrt{2\pi n}\frac{n^n}{2^n e^n}$$

$$=\lim_{n\to\infty}\sqrt{2\pi n}\left(\frac{n}{2e}\right)^n$$

$$=\infty$$

Basic Asymptotic Efficiency Classes

Class	Name	Comments
1	Constant	Short of best-case efficiencies
$\log_n$	Logarithmic	Cutting a problem size by a constant factor
n	Linear	Algorithms that scan a list of size n.(eg sequential search)
n $\log_n$	n-log-n	Many divide and conquer algorithm
n^2	Quadratic	Efficiency of algorithm with two embedded loops.
n^3	Cubic	Efficiency of algorithm with three embedded loops.
2^n	Exponential	Generate all the subsets of an n element set.

n!	Factorial	Algorithm that generate all permutations of an n element set

5. Explain the Mathematical analysis for non-recursive algorithm or write an algorithm for determining the uniqueness of an array. Determine the time complexity of your algorithm. (Apr/May-2017) April/May 2019

General plan for analyzing efficiency of non-recursive algorithm

1. Decide the **input size** based on parameter n.
2. Identify the algorithm **basic operation(s)**.
3. Check whether the number of times the **basic operation is executed** depends on only on the size of the input.
4. Set up a **sum** expressing the number of times the algorithm basic operation is excited
5. Simplify the sum using standard formula and rules

Example 1: Problem for finding the value of the largest element in a list of n numbers

The pseudo code to solving the problem is

```
ALGORITHM MaxElement(A[0..n-1])
//Problem Description : This algorithm is for finding the
//maximum value element from the array
//Input:An array A[0..n-1] of real numbers
//Output: Returns the largest element from array
Maxval ← A[0]
For i ← 1 to n-1 do ──→ Searching the maximum element from an array
{
        If ( A[i]>max_value)then
        Maxval ← A[i]  ──→ If any value is large than current Max_ Value then set new Max_value by obtained larger value
}
        Return Max_value
```

Mathematical Analysis

Step 1: The input size is the number of elements in the array(ie.),n

Step 2 : The basic operation is comparison in loop for finding larger value There are two operations in the for loop

- ✓ Comparison operation a[i]->maxval
- ✓ Assignment operation maxval->a[i]

Step 3: The comparison is executed on each repetition of the loop. As the comparison is made for each value of n there is no need to find best case worst case and average case analysis.

Step 4: Let C(n) be the number of times the comparison is executed. The algorithm makes comparison each time the loop executes. That means with each new value of I the comparison is made. Hence for i= 1 to n – 1 times the comparison is made . therefore we can formulate C(n) as

C(n) = one comparison made for each value of i

Step 5 : let us simplify the sum

Thus $C_{(n)} = \sum_{i=1}^{n-1} 1$

$= n-1 \in \theta$ **(n)**

Using the rule $\sum_{i=1}^{n} 1 = n \in \theta(n)$

The frequently used two basic rules of sum manipulation are,

$$\sum_{i=1}^{u} Ca_i = C\sum_{i=1}^{u} a_i \quad \text{——————} \quad R1$$

$$\sum_{i=1}^{u} (a_i + b_i) = \sum_{i=1}^{u} a_I + \sum_{i=1}^{u} b_i \quad \text{——————} \quad R2$$

The two summation formulas are

1. $\sum_{i=1}^{n} 1 = u - l + 1$

Where $l \leq u$ are some lower and upper integer limits ——————— S1

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

$= n(n+1)/2$

$= 1/2n^2 \in \Theta(n^2)$ ——————— S2

Example 2: Element uniqueness problem-check whether all the element in the list are distinct April/May 2019

ALGORITHM UniqueElements(A[0..n-1])
//Checks whether all the elements in a given array are distinct
//Input :An array A[0..n-1]
//Output Returns 'true' if all elements in A are distinct and 'false'
//otherwise
for i ← to n-2 do
for j ←i+1 to n-1 do
if a[i] = a[j] then
return false
else
return true

If any two elements in the array are similar then return .false indicating that the array elements

Mathematical analysis

Step 1: Input size is n i.e total number of elements in the array A

Step 2: The basic iteration will be comparison of two elements . this operation the innermost operation in the loop . Hence if a[i] = a[j] then comparison will be the basic operation .

Step 3 : The number of comparisons made will depend upon the input n . but the algorithm will have worst case complexity if the same element is located at the end of the list. Hence the basic operation depends upon the input n and worst case

Worst case investigation

Step 4: The worst case input is an array for which the number od elements comparison $c_{worst}(n)$ is the largest among the size of the array.

There are two kinds of worst case inputs, They are

1.Arrays with no equal elements.

2.Arrays in which the last two elements are pair of equal elements.

For the above inputs, one comparison is made for each repetition of the inter most loop (ie) for each value of the loop's variable 'j' between its limits i+1 and n-1 and this is repeated limit for each values of the outer loop (ie) for each value of the loop's variable `i' between 0 and n-2. Accordingly,

C worst (n) = Outer loop × Inner loop

$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$

Step 5: now we will simplify **C worst as follows**

$= \sum_{i=0}^{n-1} [(n-1) - (i+1) + 1] \quad \Theta \sum_{i=k}^{n} [1 = n - k + 1 \text{ is the rule}]$

$= \sum_{i=0}^{n-2}[(n-1-i)]$

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

Now taking (n-1) as a common factor, we can write

$$= (n-1)\sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$

This can be obtained using formula $\sum_{i=1}^{n} i = n(n+1)/2$

$$= (n-1)\sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$

This can be obtained using formula

$$\sum_{i=1}^{n} 1 = (n-1+1) = n \text{ i.e when}$$

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

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

Solving this equation we will get

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

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

$= ((n^2 - n)/2$

$= 1/2\, n^2$

$\in \Theta(n^2)$

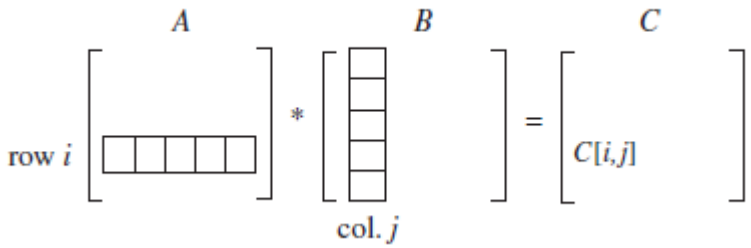
We can say that in the worst case the algorithm needs to compare all $n(n-1)/2$ distinct of its n elements.

Therefore $\mathbf{C_{worst}(n) = 1/2n^2 \in \Theta(n^2)}$

EXAMPLE 3 : Obtaining matrix multiplication

Given two *n* × *n* matrices *A* and *B*, find the time efficiency of the definition-based algorithm for computing their product *C* = *AB*, where A and B are n by n (n*n) matrices.

By definition, *C* is an *n* × *n* matrix whose elements are computed as the scalar (dot) products of the rows of matrix *A* and the columns of matrix *B*:



where $C[i, j] = A[i, 0]B[0, j] + \ldots + A[i, k]B[k, j] + \ldots + A[i, n-1]B[n-1, j]$ for every pair of indices $0 \le i, j \le n-1$.

$$C = \begin{pmatrix} a_{00} & a_{01} & a_{03} \\ 1 & 2 & 3 \\ a_{10} & a_{11} & a_{12} \\ 4 & 5 & 6 \end{pmatrix}_{2\times 3} \times \begin{pmatrix} b_{00} & b_{01} \\ 1 & 2 \\ b_{10} & b_{11} \\ 3 & 4 \\ b_{20} & b_{21} \\ 5 & 6 \end{pmatrix} \quad 3 \times 2$$

The formula for multiplication of the above two matrices is

$$\begin{pmatrix} a_{00}\times b_{00} + a_{01}\times b_{10} + a_{02}\times b_{20} & a_{00}\times b_{01} + a_{01}\times b_{11} + a_{02}\times b_{21} \end{pmatrix}$$

C =

$a_{10}\times b_{00} + a_{11}\times b_{10} + a_{12}\times b_{20}$ $a_{10}\times b_{01} + a_{11}\times b_{11} + a_{12}\times b_{21}$

$$C = \begin{bmatrix} 1\times 1 + 2\times 3 + 3\times 5 & 1\times 2 + 2\times 4 + 3\times 6 \\ 4\times 1 + 5\times 3 + 6\times 5 & 4\times 2 + 5\times 4 + 6\times 6 \end{bmatrix}$$

$$C = \begin{bmatrix} 22 & 28 \\ 49 & 64 \end{bmatrix}$$

Now the algorithm for matrix multiplication is

ALGORITHM *MatrixMultiplication*$(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$

```
//Multiplies two square matrices of order n by the definition-based algorithm
//Input: Two n × n matrices A and B
//Output: Matrix C = AB
for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
        C[i, j] ← 0.0
        for k ← 0 to n − 1 do
            C[i, j] ← C[i, j] + A[i, k] * B[k, j]
return C
```

Mathematical analysis

Step 1: The input's size of above algorithm is simply order of matrices i.e n.

Step 2: The basic operation is in the innermost loop and which is

$C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$

There are two arithmetical operations in the innermost loop here

1. Multiplication
2. Addition

Step 3: The basic operation depends only upon input size. There are no best case, worst case and average case efficiencies**.** Hence now we will go for computing **sum.** There is just one multiplication which is repeated no each execution of innermost loop**. (a for loop using variable k).** Hence we will compute the efficiency for innermost loops.

Step 4: The sum can be denoted by M (n).

M(n) = outermost × inner loop × innermost loop (1 execution)

=[for loop using i]×[for loop using j]×[for loop using k]× (1 execution)

$$M(n) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\sum_{k=0}^{n-1} 1.$$

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

$$\sum_{j=1}^{n-1} = n$$

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

$M(n)= n^3$

Thus the simplified sum is $n3$. Thus the time complexity of matrix multiplication Θ (n3)

Running time of the Algorithm T(n)

The estimation of running time of the algorithm on a particular machine is calculated by using the product.

$T(n) \approx c_m M(n) = c_m n^3$

Where- c_m is the time of one multiplication on the machine in question.

We would get a more accurate estimate if we took into account the time spent on the additions, too:

$$T(n) \approx c_m M(n) + c_a A(n) = c_m n^3 + c_a n^3 = (c_m + c_a) n^3$$

$$T(n) \approx c_m M(n) = c_m n^3$$

where c_m is the time of one multiplication on the machine in question. We would get a more accurate estimate if we took into account the time spent on the additions, too:

Time spend addition CA (n)

The time speed to perform the addition operation is given by

$$T(n) = c_a A(n) = c_a n^3$$

Where

c_a is the time taken to perform the one addition.

Hence the running time of the algorithm is given by

$$T(n) \approx c_m M(n) + c_a A(n) = c_m n^3 + c_a n^3 = (c_m + c_a) n^3$$

The estimation differs only by the multiplication constants and not by the order of growth.

EXAMPLE 4:The following algorithm finds the number of binary digits in the binary representation of a positive decimal integer.

ALGORITHM *Binary*(n)

```
//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
count ← 1
while n > 1 do
    count ← count + 1
    n ← ⌊n/2⌋
return count
```

Mathematical analysis

Step 1: The input size is n i.e . The positive integer whose binary digit in binary representation needs to be checked.

Step 2 : The basic operation is denoted by while loop. And it is each time checking whether n > 1. The while loop will be executed for the number of time at which n>1 is true . it will be executed once more when n>1 is false . but when n>1 is false the statements inside while loop wont get executed.

Step 3: The value of n is halved on each repetition of the loop. Hence efficiency algorithm is equal to **log2 n**

Step 4: hence total number of times the while loop gets executed is **$[\log_2 n] + 1$**

Hence time complexity for counting number of bits of given number is **$\Theta(\log_2 n)$.** this indicates floor value of **$\log_2 n$**

6. Explain the Mathematical analysis for recursive algorithm. (Apr/May-2017) or

Discuss the steps in Mathematical analysis for recursive algorithms. Do the same for finding factorial of a number. Nov/Dec 2017 or solve the following recurrence equations using iterative method or tree Nov/Dec 2019

Discuss various methods used for mathematical analysis of recursive algorithms.May/June 2018

General plan for analyzing efficiency of recursive algorithms

1. Decide the input size based on parameter n .
2. Identify algorithms basic operations
3. Check how many times the basic operation is executed.
   To find whether the **execution of basic operation** depends upon the input size n. **determine worst, average , and best case** for input of size n. if the basic operation depends upon worst case average case and best case then that has to be analyzed separately.
4. Set up the **recurrence relation** with some initial condition and expressing the basic operation.
5. Solve the recurrence or at least determine the order of growth. While solving the recurrence we will use the **forward and backward substitution method**. And then correctness of formula can be proved with the help of **mathematical induction** method.

Example 1:Computing factorial of some number n.

To compare the factorial F(n)=n! for an arbitrary non negative integer

N! =1.2.3……(n-1).n

= (n-1)!.n ,for n≥1

0! =1

By definition F(n)=F(n-1)!.n

ALGORITHM $F(n)$

//Computes $n!$ recursively
//Input: A nonnegative integer n
//Output: The value of $n!$
if $n = 0$ **return** 1
else return $F(n-1) * n$

Mathematical Analysis:

Step 1: The algorithm's input size is n.

Step 2: The algorithm's basic operation in computing factorial is multiplication .

Step 3 : The recursive function call can be formulated as

According to the formula, F(n) is computed as

F(n) = F(n-1) * n, for n>0

And the number of execution is denoted by M(n).

The number of multiplication M(n) is computed as

M(n) = M(n-1) + 1, for n>0

To compute F(n-1) To multiply F(n-1) by n

M(n-1) multiplication are spent to compute F(n-1).

One more multiplication is needed to multiply the result by n.

Step 4: in step 3 the recurrence relation is obtained.

The equation is

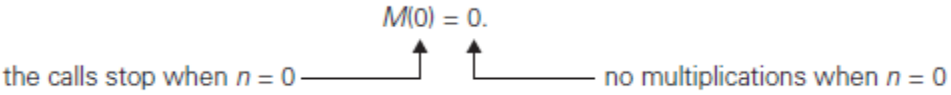
M(n)=M(n-1) +1, for n>0

Defines M(n)not explicitly(i.e.)as a function of n, but implicitly as function of its value at another point, namely n-1. These equations are called a**s recurrence relations or recurrences.**

- Recurrences relations play an important role in the analysis of algorithm and some area of applied mathematics.
- To solve a recurrence relation M(n)=M(n-1)+1 the formula for the sequence M(n) in terms of n only should be find.
- To determine the unique solution, an initial condition is needed that tells the value with which the sequence starts.
- The initial value is obtained from the condition if n=0 return 1 that makes the algorithm stops.

The condition, if n=0 return 1 tells 2 things

1. The recursive call stops when n=0 the smallest value for which the algorithm is executed. Hence M(n)=0.
2. When n=0 the algorithm performs no multiplication



Forward Substitution:

M(1) = M(0) +1

M(2) = M(1) + 1 = 1 + 1 =2

M(3) = M(2) + 1 = 2 + 1=3

The recurrence relation and the initial condition for the algorithm number of multiplication M(n) is

M(n)=M(n-1)+1,for n≤0, M(0)=0

Backward substitution:

M(n) = M(n-1) + 1

Substitute M(n-1) = M(n-2) + 1

Now M(n) becomes

M(n) = [M(n-2)+1]+1

= M(n-2) + 2

Substitute **M(n-2) =M(n-3)+1**

Now M(n) becomes

M(n)=[M(n-3)+1] + 2

= M(n-3) + 3

From the substitution method we can establish a general formula as :

M(n)= M(n-i) + i;

Since n=0, substitute i=n;

Now let us prove correctness of this formula using mathematical induction as follows

Proof

M(n) = n by using mathematical induction

Basis : let n = 0 then

M(n) =0

i.e M(0) = 0=n

Induction: if we assume M(n – 1) = n-1 then

M(n) = M(n-1) + 1

= n-1 + 1

= n

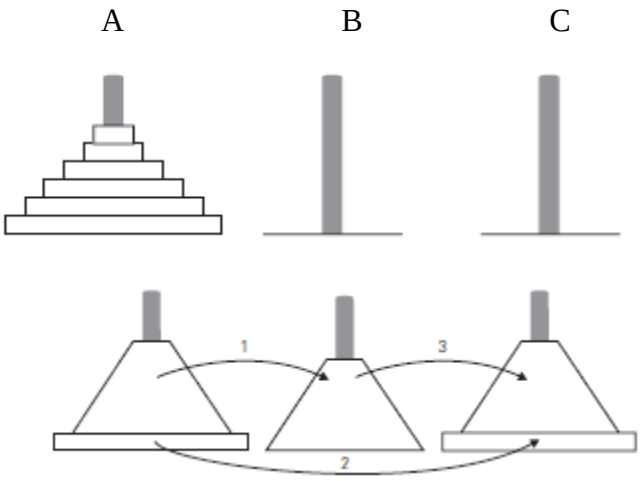
i.e M(n) = n Thus the time complexity of factorial function is Θ (n)

Give the general plan for Analyzing the time efficiency of Recursive Algorithms and use recurrence to find number of moves for Towers of Hanoi problem. May/June 2018

Example 2:Tower of Hanoi puzzle

- ✓ In this puzzle, there are n disks of different sizes, and three pegs.
- ✓ Initially all the disks are on the first peg in order if size, the largest on the bottom and the smallest on the top.
- ✓ The goal is to move all the disks from peg 1 to peg 3 using peg 2 as auxiliary.
- ✓ One disk should be moved at a time and do not place a larger disk on top of a smaller one.
- ✓ The following steps are used to move n>1 disks from peg 1 to peg 3, peg 2 as auxiliary.
    1. Move n-1 disks recursively from peg 1 to peg 3.(peg 2 as auxiliary).
    2. Move the largest disk directly from peg 1 to peg 3.
    3. Move n-1 disks recursively from peg 2 to peg 3.(peg 2 as auxiliary).

For example, if n=1 then the single disks is moved from source peg to destination peg directly.



General plan to tower of Hanoi problem

The input size is the number of disks “n”.

The algorithm basic operation is moving one disks at a time.

The number of moves M(n) depends only on n.

The recurrence equation is,

M(n)=M(n-1)+1+M(n-1),for n>1;

M(n)=2M(n-1)+1, for n>1;

The initial condition M(1)=1

Now the recurrence relation for number of moves is,

M(n)=2M(n-1)+1,for n>1

M(1)=1

The recurrence relation is solved by using backward substitution method

Backward substitution Method

M(n)=2M(n-1)+1

Substitute

M(n-1)=2M(n-2)+1

M(n)=2[2M(n-2)+1]+1

M(n)=2^2M(n-2)+2+1

Substitute

M(n-2)=2M(n-3)+1

Now, M(n) becomes

M(n)=2^2[2M(n-3)+1]+2+1

M(n)=2^3[M(n-3)+2^2+2+1

Hence after I substitution M(n) becomes

M(n)=2^iM(n-i)+2^{i-1}+2^{i-2}+2^{i-3}+…….2+1

=2^iM(n-i)+2^i-1

Therefore te general formula is **2^iM(n-i)+2^i-1**

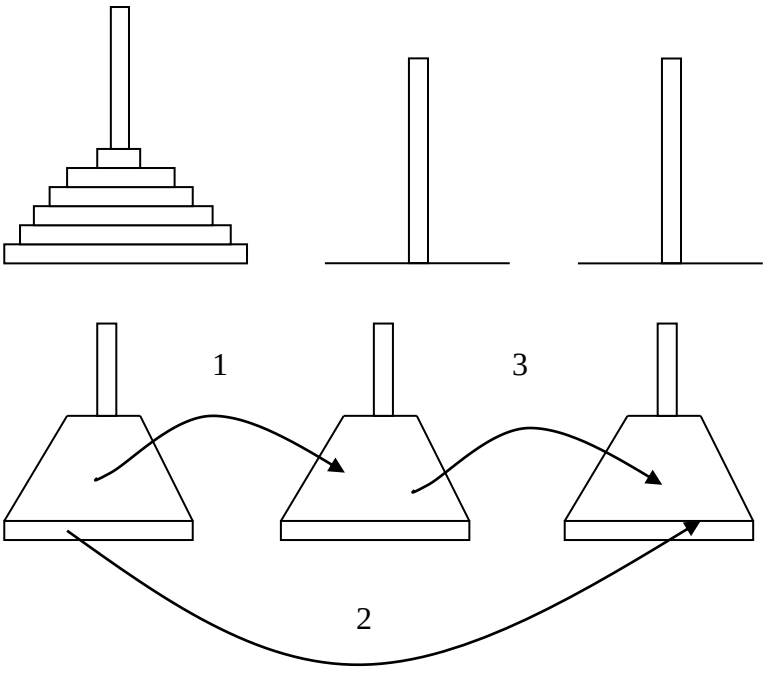


Fig. recursive solution to the Tower of Hanoi puzzle

Solution to recurrence relation is

Since the initial condition is n=1 becomes i=n-1.

The recurrence relation is

$$M(n)=2^iM(n-i)+2^i-1 \quad \text{.....(1)}$$

Substitute I=n-1 in (1)

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

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

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

$$=2^n-1$$

$M(n)= 2^n-1$ Thus this is an exponential algorithm, It runs unimaginably long time for moderate values of n.

Example 3 :To find the number of binary digits in binary representation

```
Algorithm BinRec(n)
//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
if n=1
    return 1
else
    return BinRec([n/2])+1
```

Recurrence and Initial Condition

A Recurrence for the number of addition A(n) made by the algorithm is the number of addition made in computing BinRec([n/2]) is A([n/2]) plus one more addition is made

Thus recurrence is

A(n)=A([n/2])+1,for $\geq$n

A(n)->number of addition made by the algorithm

A([n/2])->number of addition made to compute A9[n/2])

The recursive call end when n is equal to 1 and no addition is made.

The initial condition is **A(1) = 0**

To solve the recurrence, backward substitutions cannot be used.The reason is the presence on [n/2] in the functions argument and the value of n is not power of 2.

A theorem called **Smoothness rule** is used to solve the recurrence.

The standard approach for solving such recurrence is to solve it only for $n = 2^k$.

The order of growth observed for $n = 2^k$ gives a correct answer about the order of growth of all values of n.

$n = 2^k$ takes the form

$A(2^k) = A(2^{k-1}) + 1$ for $k > 0$,

$A(2^0) = 0$.

Now, backward substitutions can be applied.

Backward Substitution Method

$A(2^k) = A(2^{k-1}) + 1$

substitute $A(2^{k-1}) = A(2^{k-2}) + 1$

$= [A(2^{k-2}) + 1] + 1$

$= A(2^{k-2}) + 2$

substitute $A(2^{k-2}) = A(2^{k-3}) + 1$

$= [A(2^{k-3}) + 1] + 2$

$= A(2^{k-3}) + 3$ ... ...

After i iteration

$A(2^k) = A(2^{k-i}) + i$

$= A(2^{k-k}) + k$

$= A(2^0) + k$

$= A(1) + k$

Thus, we end up with

$A(2^k) = A(1) + k = k$

After returning to the original variable

$n = 2k$ and hence $k = \log_2 n$,

$A(n) = \log_2 n \in \Theta(\log n)$

Example 4: Fibonacci series

A sequence of Fibonacci numbers is 0,1,1,2,3,5,8,13,21,34……..

The Fibonacci sequence can be defined by the simple recurrence

$F(n)=F(n-1)+F(n-2)$,for $n>1$……1

The two initial conditions are

$F(0)=0$
$F(1)=1$

Explicit formula for the n^{th} Fibonacci number

Backward substitution method is not used to solve the recurrence $F(n)=F(n-1)+F(n-2)$,for $n>1$,because which fails to produce easily discernible pattern.

So, the theorem that describes solution to a homogeneous second order linear recurrence with constant coefficient is used to solve the problem.

The homogenous with constant coefficient is

$$ax(n)+bx(n-1)+cx(n-2)=0 \quad \ldots\ldots\ldots\ldots\ldots(2)$$

Where,

a,b,c are fixed real numbers called the coefficients of recurrence and $a \neq 0$

x(n) is the unknown sequence to be found

The characteristics equation of the recurrence equation is

$$\mathbf{Ar^2+br+c=0} \quad \ldots\ldots\ldots\ldots\ldots\ldots(3)$$

The recurrence relation can be written as

$$\mathbf{F(n)-F(n-1)-F(n-2)=0} \ldots\ldots\ldots\ldots(4)$$

The characteristics equation for (4)

$$\mathbf{r^2-r-1=0}$$

The roots are

$$\mathbf{R_{1,2}}=\frac{-1 \pm \sqrt{1-4(1)}}{2}$$

$$\mathbf{R_{1,2}}=\frac{1 \pm \sqrt{5}}{2}$$

$$\mathbf{R_1}=\frac{1+\sqrt{5}}{2}$$

$$\mathbf{R_2}=\frac{1-\sqrt{5}}{2}$$

The characteristics equation has two distinct real roots.

Now the recurrence relation is

$$\mathbf{X(n)=\alpha r_1^n+\beta r_2^n} \quad \ldots\ldots\ldots(5)$$

Substitute r1 and r2 in (5),

$$\mathbf{F(n)= \alpha\left(\frac{1+\sqrt{5}}{2}\right)^n+\beta\left(\frac{1+\sqrt{5}}{2}\right)^n} \quad \ldots\ldots\ldots(6)$$

Now substitute the value of f(0) and F(1) in equation(6)

$$\mathbf{F(0) = \alpha\left(\frac{1+\sqrt{5}}{2}\right)^0+\beta\left(\frac{1+\sqrt{5}}{2}\right)^0 =0} \quad \ldots\ldots\ldots(7)$$

$$\mathbf{F(1)= \alpha\left(\frac{1+\sqrt{5}}{2}\right)^1+\beta\left(\frac{1+\sqrt{5}}{2}\right)^1 =0} \quad \ldots\ldots\ldots(8)$$

By solving equation (7) and (8),the linear equation in two unknown α and β

$$\alpha+\beta=0$$

$$\alpha\left(\frac{1+\sqrt{5}}{2}\right)+\beta\left(\frac{1+\sqrt{5}}{2}\right)=0 \quad \ldots\ldots\ldots(11)$$

(11)-(10) gives

$$\left(\frac{1+\sqrt{5}}{2}\right)\beta-\left(\frac{1+\sqrt{5}}{2}\right)\beta=-1$$

$$\frac{\beta}{2}+\frac{\sqrt{5}}{2}\beta-\frac{\beta}{2}+\frac{\sqrt{5}}{2}\beta=-1$$

$$2\,\frac{\sqrt{5}}{2}\beta=-1$$

$$\beta=-\frac{\sqrt{5}}{2}$$

Substitute $\beta=-\frac{\sqrt{5}}{2}$ in (9)

$$\alpha+\beta=0$$

$$\alpha-\frac{1}{\sqrt{5}}=0$$

$$\alpha=\frac{1}{\sqrt{5}}\quad \beta=-\frac{\sqrt{5}}{2}$$

Substitute the value of α and β in equation (6)

$$F(n)=\frac{1}{\sqrt{5}}\left[\frac{1+\sqrt{5}}{2}\right]^n-\frac{1}{\sqrt{5}}\left[\frac{1+\sqrt{5}}{2}\right]^n$$

$$F(n)=\frac{1}{\sqrt{5}}[\phi n-\phi\hat{}n]$$

Where

$$\Phi=\frac{1+\sqrt{5}}{2}$$

$$\Phi=1.61803$$

$$\Phi\hat{}=-\frac{1}{\Phi}$$

$$\Phi\hat{}=-0.61803$$

The constant Φ is known as, Golden Ratio.

The value of Φ^ is lies between -1 and 0.

When n goes to infinity, Φ^ gets infinitely small value. So, it can be omitted.

Therefore $F(n)=\frac{1}{\sqrt{5}}\Phi^n$

So, for every non negative n, $F(n)=\frac{1}{\sqrt{5}}\Phi^n$ is rounded to the nearest integer.

Algorithm for computing Fibonacci numbers

First method

Algorithm F(n)

//Computes the nth Fibonacci number recursively by using its definition.

//Input: A nonnegative integer n

//Output: The nth Fibonacci number

```
if n<1
        return n
```

```
    Else
        return F(n-1)+(n-2)
```

the algorithm's basic operation is addition.

Let A(n) is the number of additions performed by the algorithm to compute F(n).

The number of additions needed to compute F(n-1) is A(n-1) and the number of additions needed to compute F(n-2) is A(n-2).

The algorithm needs one more addition to compute the sum of A(n-1) and A(n-2).

Thus the recurrence for A(n) is

A(n)=A(n-1) + A(n-2)+1, for n>1

A(0)=0

A(1)=0

The recurrence A(n)-A(n-1)-A(n-2)=1 is same as F(n)-F(n-1)-F(n-2)=0, but its right hand side not equal to zero. These recurrences are called **inhomogeneous recurrences**.

General techniques are used to solve inhomogeneous recurrences.

The inhomogeneous recurrences is converted into homogeneous recurrence by rewriting the in homogeneous recurrence as,**A(n)+1]-[A(n-1)+1]-[A(n-2)+1]=0 (14)**

Now substitute, **B(n)=A(n)+1**

Now (14) becomes, **B(n)-B(n-1)-B(n-2)=0**

B(0)=0

B(1)=1

Here **B(n)=F(n+1)**

Since **B(n)=A(n)+1**

B(n-1)=A(n)

So **A(n)=B(n)-1**

Substitute F(n+1)-1 ……(15)

We know that

F(n)=$\frac{1}{\sqrt{5}}(\varnothing^n - \varnothing^n)$

F(n+1)=$\frac{1}{\sqrt{5}}(\varnothing^{n+1} - \varnothing^{n+1})$ ……(16)

Substitute (16) in (15)

A(n)=$\frac{1}{\sqrt{5}}(\varnothing^{n+1} - \varnothing^{n+1})$**-1**

Hence

A(n)€ $\Theta(\varnothing^n)$

The poor efficiency class of algorithm could be anticipated from the class of recurrence

The reason behind the algorithm inefficiency can be traced by looking at the tree of recursive calls n=6

The same values of the function are evaluated again and again which is extremely inefficiently.

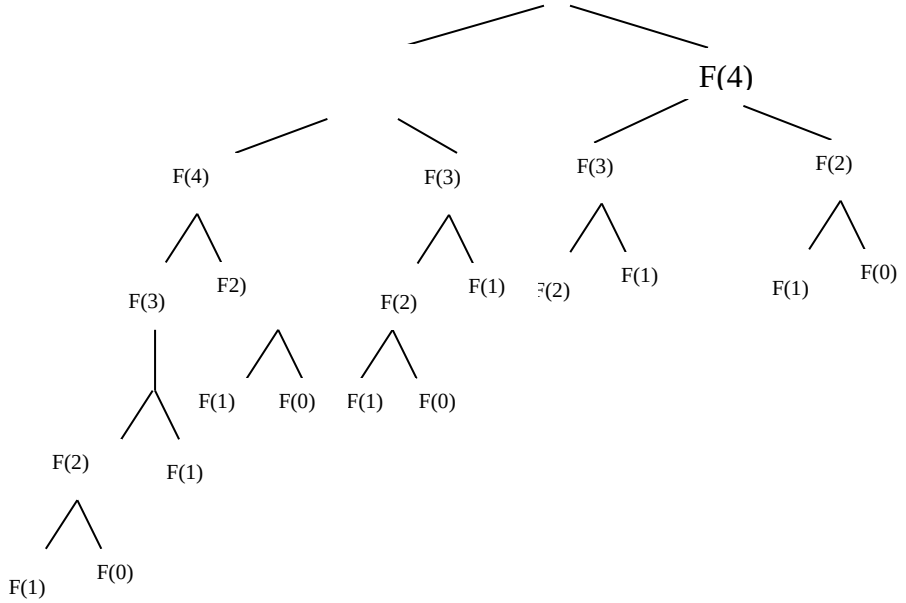


Fig Tree of recursive calls for computing the Fibonacci number for n = 6

7. Find the time complexity and space complexity of the following problems. Factorial using recursion and compute the nth Fibonacci number using iterative statements. *Dec 2012*

8.Solve the following recurrence relations: or solve the following recurrence equation:

T(n)=T(n/2)+1,where n=2^k for all k>=0

T(n)= T(n/3)+ T(2n/3)+cn,where 'c' is a constant and 'n' is the input size.

Dec 2012 **April/May 2019**

1. T(n)= $\begin{cases} 2T(n/2)+3 & n>2 \\ 2 & n=2 \end{cases}$

T(n)=2T(n/2)+3

=2{(2T(n/2)+3)/2}+3

=2{(2T(n/4)+3/2}+3

....

=4T(n/4)+6

= 4{(2T(n/2)+3)/4}+6

.....

=8T(n/8)+9

$=2^kT(n/2^k)+3n$

T(n)=nlogn+3n

Time complexity=o(nlog n)

2. T(n)= $\begin{cases} 2T(n/2)+cn & n>1 \\ a & n=1 \end{cases}$ where a and c constants

T(n)=2T(n/2)+cn

=2{(2T(n/2)+cn)/2}+cn

=2{(2T(n/4)+cn/2}+cn

=4T(n/4)+cn+cn

= 4{(2T(n/8)+cn/4}+ cn+cn

=8T(n/8)+ cn+cn+cn

$=2^kT(n/2^k)+k(cn)$

T(n)=nlogn+ k(cn)

Time complexity=o(nlog n)

8. Show the following equalities are correct *June 2013*

i. $5n^2-6n = \Phi(n^2)$

ii. $n!=O(n^n)$

iii. $n^3+10^6n^2=\Theta(n^3)$

iv. $2n^22^n + n \log n = \Theta(n^22^n)$

i. $5n^2-6n = \Phi(n^2)$ =>higest order of grouth is n^2

ii. $n!=O(n^n)$ =>higest order of grouth O(n)

iii. $n^3+10^6n^2=\Theta(n^3)$ =>higest order of grouth $O(n^3)$

iv. $2n^22^n + n \log n = \Theta(n^22^n)$=>higest order of grouth $O(n^2)$

Nov 2010

9. Prove that for any two functions f(n) and g(n), we have f(n)-> Θ(g(n)) if and only if f(n) -> O(g(n)) and f(n) ->Ω(g(n)) *Nov 2010*

Given function:

f(n) and g(n)

f(n)= O(g(n)) when $f(n) \le C_1g(n)$ for all $n \ge n_0$--------(1)

f(n)= Ω(g(n)) when $f(n) \ge C_2g(n)$ for all $n \ge n_0$--------(2)

from (1) and (2)

$C_2\, g(n) \le f(n) \le C_{1g(n)}$ for all $n \ge n_0$ -------(3)

(i.e) Θ(g(n)) = O(g(n))Ω(g(n))

From (3) f(n) = Θ(g(n)) hence proved

10. (a)If you have to solve the searching problem for a list of n numbers, how can you take

advantage of the fact that the list is known to be sorted? Give separate answers for lists represented as arrays lists represented as linked lists. (AU april/may 2015)

For a sorted array do a binary search to divide the array in half for each query, thus O(lg n). If the list is linked you must you do a linear search which is O(n), unless you use a linked binary search tree, which is O(lg n)

11. The best-case analysis is not as important as the worst-case analysis of an algorithm". Yes or No ? Justify your answer with the help of an example. (April/May 2021)

The Best Case analysis is bogus. **Guaranteeing a lower bound on an algorithm doesn't provide any information as** in the worst case, an algorithm may take years to run. For some algorithms, all the cases are asymptotically the same, i.e., there are no worst and best cases. For example, Merge Sort.

11.Derive the worst case analysis of merge sort using suitable illustration (AU april/may 2015)

Efficiency of Merge Sort

- In merge sort algorithm the two recursive calls are made. Each recursive call focuses on n/2 elements of the list .
- After two recursive calls one call is made to combine two sublist i.e to merge all n elements.
- Hence we can write recurrence relation as

T(n) = T(n/2) + T(n/2) + cn

T(n/2) = Time taken by left sublist

T(n/2) = time taken by right sublist

T(n) = time taken for combining two sublists

where n> 1 T (1) = 0

The time complexity of merge sort can be calculated using two methods

- Master theorem
- Substitution method

Master theorem Let , the recurrence relation for merge sort is

T(n) = T(n/2) + T(n/2) + cn

Let T(n) = aT(n/b) + f(n) be a recurrence relation

i.e. T(n) = 2T(n/2) + cn ------- (1)

T(1) = 0 ---------- (2)

As per master theorem $T(n) = \Theta (n^d \log n)$ if a = b

As equation (1),a =2 , b = 2 and f(n) = cn and $a = b^d$ i.e $2 = 2^`$

This case gives us , $T (n) = \Theta (n \log 2 n)$

Hence the average and worst case time complexity of merge sort is

$$C_{worst}(n) = (n \log 2 n)$$

Substitution method Let, the recurrence relation for merge sort be

T(n) = T(n/2) + T(n/2) + cn for n>1

i.e. T(n) = 2T(n/2) + cn for n>1 ------- (3)

$T(1) = 0$ -------(4)

Let us apply substitution on equation (3) .

Assume $n=2^k$

$T(n) = 2T(n/2) + cn$

$T(n) = 2T(2^k/2) + c.2^k$

$T(2^k) = 2T(2^k-1) + c.2^k$

If k = k-1 then,

$T(2^k) = 2T(2^{k-1}) + c.2^k$

$T(2^k) = 2[2T(2^{k-2}) + c.2^k -1] + c.2^k$

$T(2^k) = 2^2 T(2^{k-2}) + 2.c.2^k -1 + c .2^k$

$T(2^k) = 2^2 T(2^{k-2}) + 2.c.2^k /2 + c.2^k$

$T(2^k) = 2^2 T(2^{k-2}) + c.2^k + c.2^k$

$T(2^k) = 2^2 T(2^{k-2}) + 2c .2^k$

Similarly we can write,

$T(2^k) = 2^3 T(2^{k-3}) + 3c .2^k$

$T(2^k) = 2^4 T(2^{k-4}) + 4c .2^k$

…..

….

$T(2^k) = 2^k T(2^{k-k}) + k.c.2^k$

$T(2^k) = 2^k T(2^0) + k.c.2^k$

$T(2^k) = 2^k T(1) + k.c.2^k$ -------- (5)

But as per equation (4), T(1) =0

There equation (5) becomes ,

$T(2^k) = 2^k .0 +. k. c . 2^k$

$T(2^k) = k. c . 2^k$

But we assumed $n=2^k$, taking logarithm on both sides.i.e. log 2 n = k

Therefore T(n) = log 2 n. cn

Therefore T (n) =Θ (n log2 n)

Hence the average and worst case time complexity of merge sort is

$C_{worst}(n) = (n \log_2 n)$

Time complexity of merge sort

Best case	**Average case**	**Worst case**
Θ (n log2 n)	Θ (n log2 n)	Θ (n log2 n)

12.write Insertion sort algorithm and estimate its running time.

- ✓ Like selection sort, insertion sort loops over the indices of the array. It just calls `insert` on the elements at indices `1,2,3,…,n−1`. Just as each call to `indexOfMinimum` took an amount of time that depended on the size of the sorted subarray, so does each call to `insert`. Actually, the word "does" in the previous sentence should be "can," and we'll see why.
- ✓ Let's take a situation where we call `insert` and the value being inserted into a subarray is less than every element in the subarray.
- ✓ For example, if we're inserting 0 into the subarray [2, 3, 5, 7, 11], then every element in the subarray has to slide over one position to the right. So, in general, if we're inserting into a subarray with `k` elements, all `k` might have to slide over by one position.
- ✓ Rather than counting exactly how many lines of code we need to test an element against a key and slide the element, let's agree that it's a constant number of lines; let's call that constant `ccc`. Therefore, it could take up to `c·k` lines to insert into a subarray of `k` elements.
- ✓ Suppose that upon every call to `insert`, the value being inserted is less than every element in the subarray to its left. When we call `insert` the first time, `k=1`. The second time, `k=2`. The third time, `k=3`. And so on, up through the last time, when `k=n−1`.

Therefore, the total time spent inserting into sorted subarrays is **$c\cdot1+c\cdot2+c\cdot3+\cdots c\cdot(n-1)=c\cdot(1+2+3+\cdots+(n-1))$**

That sum is an arithmetic series, except that it goes up to `n−1n−1n−1` rather than `nnn`. Using our formula for arithmetic series, we get that the total time spent inserting into sorted subarrays is

$c\cdot(n-1+1)((n-1)/2)=cn2/2-cn/2.$

Using big-Θ notation, we discard the low-order term `cn/2` and the constant factors `c` and 1/2, getting the result that the running time of insertion sort, in this case, is `Θ(n2)`.

Can insertion sort take *less* than `Θ(n2)` time? The answer is yes. Suppose we have the array [2, 3, 5, 7, 11], where the sorted subarray is the first four elements, and we're inserting the value 11. Upon the first test, we find that 11 is greater than 7, and so no elements in the subarray need to slide over to the right.

- ✓ Then this call of `insert` takes just constant time. Suppose that *every* call of `insert` takes constant time. Because there are `n−1` calls to `insert`, if each call takes time that is some constant `ccc`, then the total time for insertion sort is `c·(n−1)` which is `Θ(n)`, not `Θ(n2)`.
- ✓ Can either of these situations occur? Can each call to `insert` cause every element in the subarray to slide one position to the right? Can each call to `insert` cause no elements to slide? The answer is yes to both questions.
- ✓ A call to `insert` causes every element to slide over if the key being inserted is less than every element to its left. So, if every element is less than every element to its left, the running time of insertion sort is `Θ(n2)`.
- ✓ What would it mean for every element to be less than the element to its left? The array would have to start out in *reverse* sorted order, such as [11, 7, 5, 3, 2]. So a reverse-sorted array is the worst case for insertion sort.
- ✓ How about the opposite case? A call to `insert` causes no elements to slide over if the key being inserted is greater than or equal to every element to its left. So, if every element is greater than or equal to every element to its left, the running time of insertion sort is `Θ(n)`.
- ✓ This situation occurs if the array starts out already sorted, and so an already-sorted array is the best case for insertion sort.

What else can we say about the running time of insertion sort? Suppose that the array starts out in a random order. Then, on average, we'd expect that each element is less than half the elements to its left.

- ✓ In this case, on average, a call to `insert` on a subarray of `k` elements would slide `k/2` of them. The running time would be half of the worst-case running time. But in asymptotic notation, where constant coefficients don't matter, the running time in the average case would still be $\Theta(n^2)$, just like the worst case.
- ✓ What if you knew that the array was "almost sorted": every element starts out at most some constant number of positions, say 17, from where it's supposed to be when sorted?
- ✓ Then each call to `insert` slides at most 17 elements, and the time for one call of `insert` on a subarray of `kkk` elements would be at most $17 \cdot c$. Over all $n-1$ calls to `insert`, the running time would be $17 \cdot c \cdot (n-1)$, which is $\Theta(n)$, just like the best case. So insertion sort is fast when given an almost-sorted array.

To sum up the running times for insertion sort:

- **Worst case: $\Theta(n^2)$.**
- **Best case: $\Theta(n)$.**
- **Average case for a random array: $\Theta(n^2)$.**
- **"Almost sorted" case: $\Theta(n)$.**

f you had to make a blanket statement that applies to all cases of insertion sort, you would have to say that it runs in $O(n^2)$ time. You cannot say that it runs in $\Theta(n^2)$ time in all cases, since the best case runs in $\Theta(n)$ time. And you cannot say that it runs in $\Theta(n)$ time in all cases, since the worst-case running time is $\Theta(n^2)$.

13.Show how to implement a stack using two queues.Analyze the running time of the stack operations.

Show how to implement a stack using two queues. Analyze the running time of the stack operations.

- The pseudocode is as follows.

```
public class TwoQueueStack
{
  Queue q1;
  Queue q2;
  int flag = 0;
  // 0: the stack is empty;
  // 1: all data are stored in q1;
  // 2: all data are stores in q2;

  // always push into the empty queue, then move all data from
  // the other queue to the current queue.
  // always pop element from the queue containing data.
  // suppose q1 and q2 are implemented by linked list. The queues never
  // get full.

  public void push(Object e)
  {
    switch(flag)
    {
      case 0: q1.enQueue(e);
              flag = 1;
              break;
      case 1: q2.enQueue(e);
              while (!q1.isEmpty())
                 q2.enQueue(q1.deQueue());
              flag = 2;
              break;
      case 2: q1.enQueue(e);
              while (!q2.isEmpty())
                 q1.enQueue(q2.deQueue());
              flag = 1;
              break;
      default: error 'illegal state';
    }
  }


 public Object pop()
 {
   switch(flag)
   {
     case 0: error 'underflow -- stack is empty, can't pop';
     case 1: retElement = q1.deQueue();
             if (q1.isEmpty())
                flag = 0;
             return retElement;
     case 2: retElement = q2.deQueue();
             if (q2.isEmpty())




                       flag = 0;
                  return retElement;
           default: error 'illegal state';
         }
       }
     }
```

- The running time for `push` operation is $O(n)$. The running time for `pop` operation is $\Theta(1)$.

14. find the closest asymptotic tight bound by solving the recurrence equation *T(n)=8T(n/2)+n2* with *(T(1)=1)* using recursion tree method.[Assume that T(1)ЄΘ(1)]

Example: Solve $T(n) = 8T(n/2) + n^2 \quad (T(1) = 1)$

$$\begin{aligned}
T(n) &= n^2 + 8T(n/2) \\
&= n^2 + 8(8T(\frac{n}{2^2}) + (\frac{n}{2})^2) \\
&= n^2 + 8^2T(\frac{n}{2^2}) + 8(\frac{n^2}{4})) \\
&= n^2 + 2n^2 + 8^2T(\frac{n}{2^2}) \\
&= n^2 + 2n^2 + 8^2(8T(\frac{n}{2^3}) + (\frac{n}{2^2})^2) \\
&= n^2 + 2n^2 + 8^3T(\frac{n}{2^3}) + 8^2(\frac{n^2}{4^2})) \\
&= n^2 + 2n^2 + 2^2n^2 + 8^3T(\frac{n}{2^3}) \\
&= \dots \\
&= n^2 + 2n^2 + 2^2n^2 + 2^3n^2 + 2^4n^2 + \dots
\end{aligned}$$

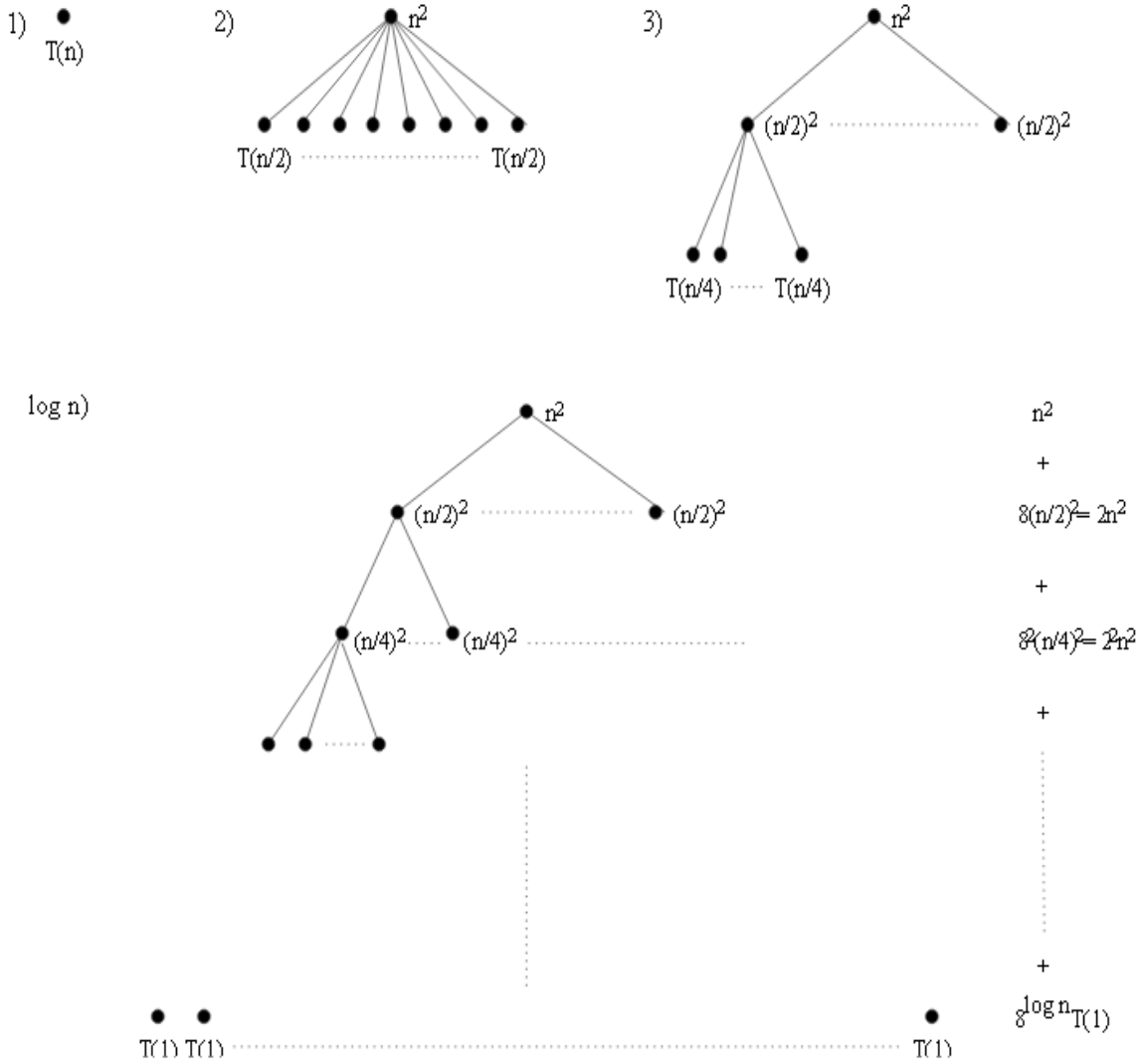
- Recursion depth: How long (how many iterations) it takes until the subproblem has constant size? i times where $\frac{n}{2^i} = 1 \Rightarrow i = \log n$
- What is the last term? $8^iT(1) = 8^{\log n}$

$$\begin{aligned}
T(n) &= n^2 + 2n^2 + 2^2n^2 + 2^3n^2 + 2^4n^2 + \dots + 2^{\log n - 1}n^2 + 8^{\log n} \\
&= \sum_{k=0}^{\log n - 1} 2^k n^2 + 8^{\log n} \\
&= n^2 \sum_{k=0}^{\log n - 1} 2^k + (2^3)^{\log n}
\end{aligned}$$

- Now $\sum_{k=0}^{\log n - 1} 2^k$ is a geometric sum so we have $\sum_{k=0}^{\log n - 1} 2^k = \Theta(2^{\log n - 1}) = \Theta(n)$
- $(2^3)^{\log n} = (2^{\log n})^3 = n^3$

$$\begin{aligned}
T(n) &= n^2 \cdot \Theta(n) + n^3 \\
&= \Theta(n^3)
\end{aligned}$$

- we draw out the recursion tree with cost of single call in each node—running time is sum of costs in all nodes
- if you are careful drawing the recursion tree and summing up the costs, the recursion tree is a direct proof for the solution of the recurrence, just like iteration and substitution
- Example: $T(n) = 8T(n/2) + n^2 \quad (T(1) = 1)$



$$T(n) = n^2 + 2n^2 + 2^2n^2 + 2^3n^2 + 2^4n^2 + \ldots + 2^{\log n - 1}n^2 + 8^{\log n}$$

15.Derive a loose bound on the following equation: $F(x)=35\ x^8 -22x^7+14x^5 -2x^4 -4x^2+x-15$

$f(x) = 35x^8 - 22x^7 + 14x^5 - 2x^4 - 4x^2 + x - 15$

Solution : Let, f(n) and g(n) are two non-negative functions.

Let c be some constant.

The equation

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

then $f(n) \in O(g(n)$ with tight bound.

But if $f(n) < c * g(n)$

then $f(n) \in O(g(n)$ with loose bound.

Consider the function

$f(x) = 35x^8 - 22x^7 + 14x^5 - 2x^4 - 4x^2 + x - 15$

and $g(x) = x^8$

If $x = 1$, then

$f(x) = 35(1)^8 - 22(1)^7 + 14(1)^5 - 2(1)^4 - 4(1)^2 + 1 - 15$

$f(x) = 7$

$g(x) = x^8 = (1)^8$

If we assume $c = 35$ then

We will always get

$f(x) < g(x)$ for $x \geq 1$

16.Solve the recurrence relations

X(n) =x(n-1) +5 for n > 1 x(1)=0

X(n) =3x(n-1) for n > 1 x(1)=4

X(n) =x(n-1) +n for n > 0 x(0)=0

X(n) =x(n/2) +n for n > 1 x(1)=1 (solve for n= 2^k)

X(n) =x(n/3) +1 for n > 1 x(1)=1 (solve for n= 3^k)

X(n) =x(n-1) +5 for n > 1 x(1)=0

X(1)=0
If n=2
X(2)=x(2-1)+5
=x(1)+5
=0+5
=5
If n=3
X(3)=x(3-1)+5

=x(2)+5
=5+5
=10

If n=4
X(4)=x(4-1)+5
=x(3)+5
=10+5
=15.....

17.Use the most appropriate notation to indicate the time efficiency class of sequential search algorithm in the worst case,best case and the average case.

Solution : Sequential search

"Given a target value and a random list of values, find the location of the target in the list, if it occurs, by checking each value in the list in turn"

```
get (NameList, PhoneList, Name)
i = 1
N = length(NameList)
Found = FALSE
while ( (not Found) and (i <= N) ) {
   if ( Name == NameList[i] ) {
      print (Name, "'s phone number is ", PhoneList[i])
      Found = TRUE
   }
   i = i+1
}
if ( not Found ) { print (Name, "'s phone number not found!") }
```

Central unit of work: operations that occur most frequently

Central unit of work in sequential search:

Comparison of target Name to each name in the list

Also add 1 to i

Typical iteration: two steps (one comparison, one addition)

Given a large input list:

Best case: smallest amount of work algorithm must do

Worst case: greatest amount of work algorithm must do

Average case: depends on likelihood of different scenarios occurring

- **Best case:** target found with the first comparison (**1 iteration**)
- **Worst case**: target never found or last value (*N* iterations)
- **Average case:** if each value is equally likely to be searched, work done varies from 1 to *N*, on average ***N/2*** **iterations**

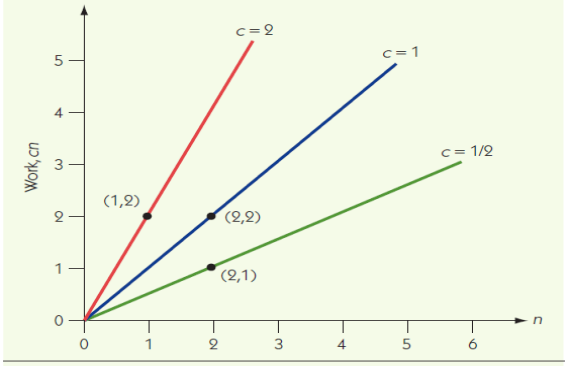
Sequential search worst case (N) grows linearly in the size of the problem 2N steps (one comparison and one addition per loop) Also some initialization steps...

On the last iteration, we may print something...After the loop, we test and maybe print...

To simplify analysis, disregard the "negligible" steps (which don't happen as often), and ignore the coefficient in 2N Just pay attention to the dominant term (N)

Order of magnitude O(N): the class of all linear functions (any algorithm that takes C1N + C2 steps for any constants C1 and C2)

FIGURE 3.4



Work = *cn* for various values of *c*

18.(i) Prove that if g(n) is Ω(f(n)) then f(n) is O(g(n)).May/June 2018

f(n) ∈ Ω(g(n)) ⟺ g(n) ∈ O(f(n))

Proof:

$O(f(n))=\{g:N\to N \mid \exists c,n0\in N \ \forall n\geq n0: g(n)\leq c\cdot f(n)\}$

$\Omega(g(n))=\{f:N\to N \mid \exists c,n0\in N \ \forall n\geq n0: f(n)\geq c\cdot g(n)\}$

Step 1/2: $f(n)\in\Omega(g(n)) \iff g(n)\in O(f(n))$

$\exists c,n0\in N\ \forall n\geq n0$: $f(n)\geq c\cdot g(n)\Rightarrow f(n)g(n)\geq c\Rightarrow 1g(n)\geq cf(n)\Rightarrow g(n)\leq 1c\cdot f(n)$

And this is exactly the definition of O(f(n)).

Step 2/2: $f(n)\in\Omega(g(n))\Leftarrow g(n)\in O(f(n))$

$\exists c,n0\in N\ \forall n\geq n0$: $g(n)\leq c\cdot f(n)\Rightarrow ...\Rightarrow f(n)\geq 1c\cdot g(n)$

Hence proved.

19. Explain briefly about Empirical Analysis of Algorithm.

The principal alternative to the mathematical analysis of an algorithm's efficiency is its empirical analysis. This approach implies steps spelled out in the following plan.

General Plan for the Empirical Analysis of Algorithm Time Efficiency

1. Understand the experiment's purpose.

2. Decide on the efficiency metric *M* to be measured and the measurement unit(an operation count vs. a time unit).

3. Decide on characteristics of the input sample (its range, size, and so on).

4. Prepare a program implementing the algorithm for the experimentation.

5. Generate a sample of inputs.

6. Run the algorithm (or algorithms) on the sample's inputs and record the data observed.

7. Analyze the data obtained.

1. **Purpose:**
   - To ensure theoretical assertion about the algorithm's efficiency
   - comparing the efficiency of several algorithms for solving the same problem or different implementations of the same algorithm
   - developing a hypothesis about the algorithm's efficiency class
   - ascertaining the efficiency of the program implementing the algorithm on a particular machine.
2. **how & What to measure**

- Include a variable counter, to count the number of times the algorithm's basic operation is executed.
- In the implementing the algorithm , measure the running time of basic operation

Example

- In unix, the system command time may be used.
- computing the difference between the two $(t_{finish} - t_{start})$.

Disadvantages of Measuring the system time

1. System's time is typically not very accurate, and you might get somewhat different results on repeated runs of the same program on the same inputs. An obvious remedy is to make several such measurements and then take their average (or the median) as the sample's observation point.
2. In the high speed of modern computers, the running time may fail to register at all and be reported as zero. The standard trick to overcome this obstacle is to run the program in an extra loop many times, measure the total running time, and then divide it by the number of the loop's repetitions.
3. The computer running under a time-sharing system such as UNIX, the reported time may include the time spent by the CPU on other programs, which obviously defeats the purpose of the experiment. Therefore, you should take care to ask the system for the time devoted specifically to execution of your program. (In UNIX, this time is called the "user time," and it is automatically provided by the time command.)

Advantage of Measuring physical running time

(i) the physical running time provides very specific information about an algorithm's performance in a particular computing environment

(ii) Measuring time spent on different segments of a program can pinpoint a bottleneck in the program's performance that can be missed by an abstract deliberation about the algorithm's basic operation profiling.

4. Deciding on a sample of inputs

   Sample size: (it is sensible to start with a relatively small sample and increase it later if necessary)

   Range of input sizes: (typically neither trivially small nor excessively large)

   - procedure for generating instances in the range chosen.
   - The instance sizes can either adhere to some pattern (e.g., 1000, 2000, 3000, . . . , 10,000 or 500, 1000, 2000, 4000, . . . , 128,000) or be generated randomly within the range chosen.
   - Several instances of the same size should be included or not.

5. Generate a sample of inputs (random numbers)

Typically, its output will be a value of a (pseudo)random variable uniformly distributed in the interval between 0 and 1. If a different (pseudo)random variable is desired, an appropriate transformation needs to be made. For example, if x is a continuous random variable uniformly distributed on the interval $0 \le x < 1$, the variable $y = l + \lfloor x(r - l) \rfloor$ will be uniformly distributed among the integer values between integers l and $r - 1 (l < r)$.

Alternatively, you can implement one of several known algorithms for generating (pseudo)random numbers. The most widely used and thoroughly studied of such algorithms is the linear congruential method

ALGORITHM

Random*(n, m,* seed*, a , b)*

//Generates a sequence of *n* pseudorandom numbers according to the linear

/ / c o n g r u e n t i a l m e t h o d

//Input: A positive integer *n* and positive integer parameters *m*, seed, *a , b*

//Output: A sequence *r*1, . . . , *rn* of *n* pseudorandom integers uniformly

/ / d i s t r i b u t e d a m o n g i n t e g e r v a l u e s b e t w e e n 0 a n d *m*−1

//Note: Pseudorandom numbers between 0 and 1 can be obtained

/ / b y t r e a t i n g t h e i n t e g e r s g e n e r a t e d a s d i g i t s a f t e r t h e d e c i m a l p o i n t

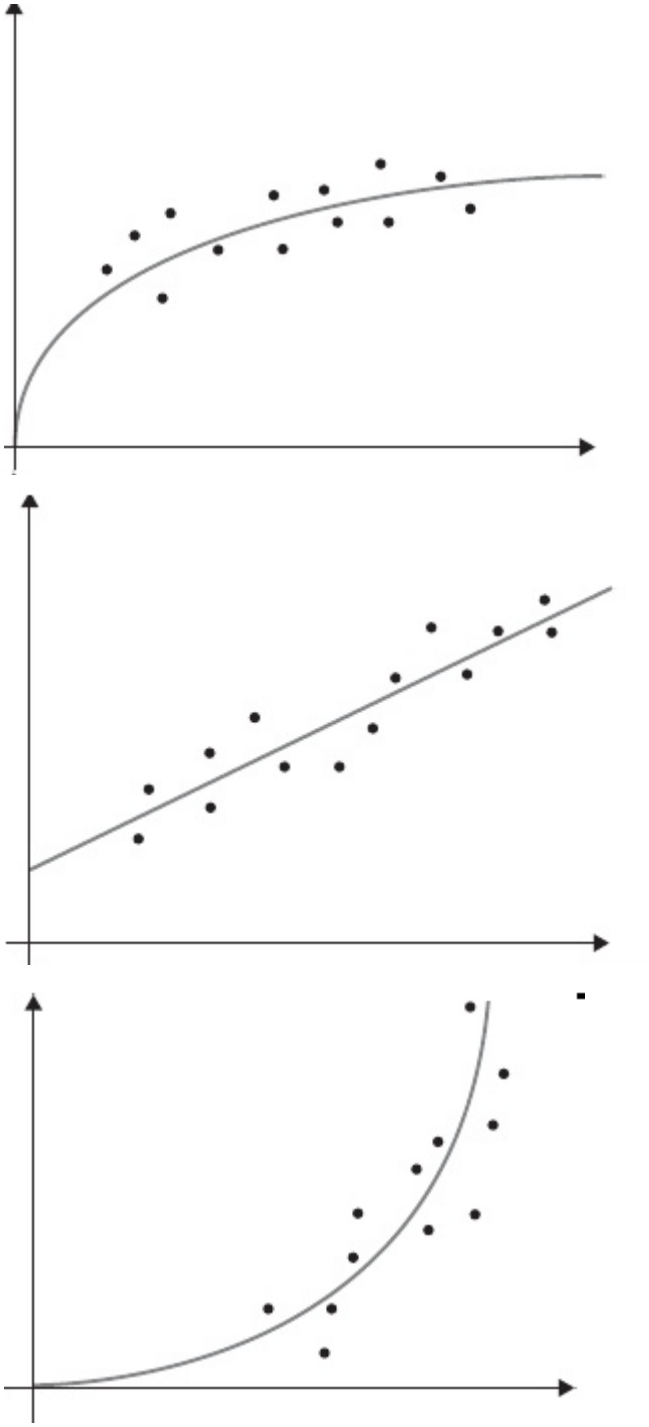
*r*0← seed

for *i* ←1 **to** *n* **do**

ri ←*(a* ∗*ri*−1+*b)* mod *m*

6. Data analysis
    - It is a good idea to use both these options whenever it is feasible because both methods have their unique strengths and weaknesses.
    - The advantages of tabulated data lies in the opportunity to manipulate it easily and to find efficiency class of the algorithm.
    - The Scatter plot representation helps in the analysis of algorithm efficiency class as given in figure

Shape of the scatter plot	**Efficiency class**
Concave shape	Logarithmic
Point around straight line or between two straight line	Linear
Convex shape	Quadratic and nlogn
Convex shape with rapid increase in the metrics valus	Cubic



Typical scatter plots. (a) Logarithmic. (b) Linear. (c) One of the convex functions

Application:

1. Predicting the algorithm performance on a sample size not included in the experiment sample.
2. The standard techniques of statistical data analysis and prediction can also be done.

20. Explain briefly about Algorithm Visualization.

Algorithm visualization is defined as the use of images to convey some useful information about algorithms. That information can be a visual illustration with the following combinations.

1. Algorithm's operation on different kinds of inputs
2. Same input for different algorithms to compare the execution speed.

An algorithm visualization uses graphic elements—points, line segments, two- or three-dimensional bars, and so on—to represent some "interesting events" in the algorithm's operation.

There are two principal variations of algorithm visualization:

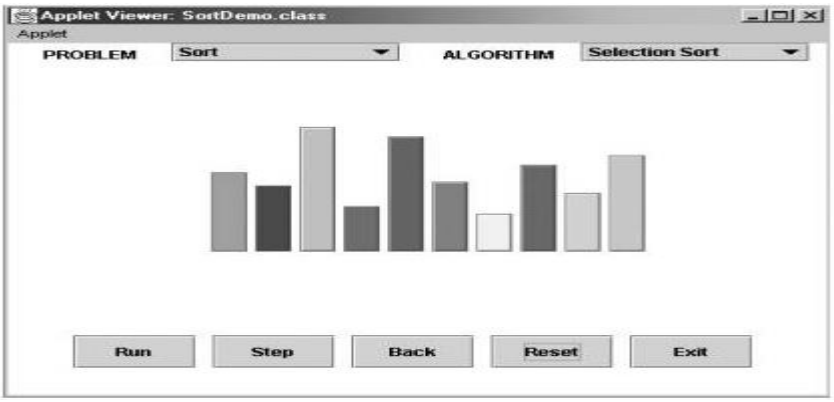
1. Static algorithm visualization
2. Dynamic algorithm visualization, also called algorithm animation

Static algorithm visualization shows an algorithm's progress through a series of still images. Algorithm animation, on the other hand, shows a continuous, movie-like presentation of an algorithm's operations. Animation is an arguably more sophisticated option, which, of course, is much more difficult to implement.

The features of an animations user interface was suggested by Peter Gloor is listed below

- Be consistent
- Be Interactive
- Be clear and concise
- Be forgiving to the user
- Adapt to the knowledge level of the user
- Emphasis the visual component
- Keep the user interested
- Incorporate both symbolic and iconic representations
- Include algorithm analysis and comparisons with other algorithm for the same problem
- Include execution history

The success of *Sorting Out Sorting* made sorting algorithms a perennial favorite for algorithm animation. Indeed, the sorting problem lends itself quite naturally to visual presentation via vertical or horizontal bars or sticks of different heights or lengths, which need to be rearranged according to their sizes (Figure 2.8). This presentation is convenient, however, only for illustrating actions of a typical sorting algorithm on small inputs. For larger files, *Sorting Out Sorting* used the ingenious idea of presenting data by a scatterplot of points on a coordinate plane, with the first coordinate representing an item's position in the file and the second one representing the item's value; with such a representation, the process of sorting looks like a transformation of a "random" scatterplot of points into the points along a frame's diagonal (Figure 2.9). In addition, most sorting algorithms work by comparing and exchanging two given items at a time—an event that can be animated relatively easily.



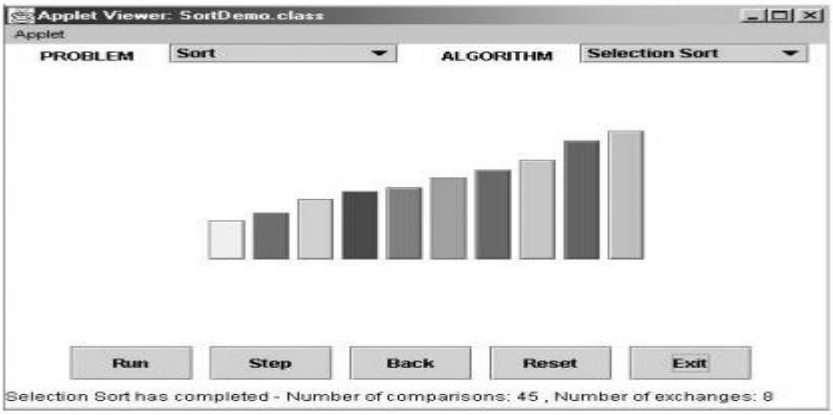


FIGURE 2.8 Initial and final screens of a typical visualization of a sorting algorithm using the bar representation.

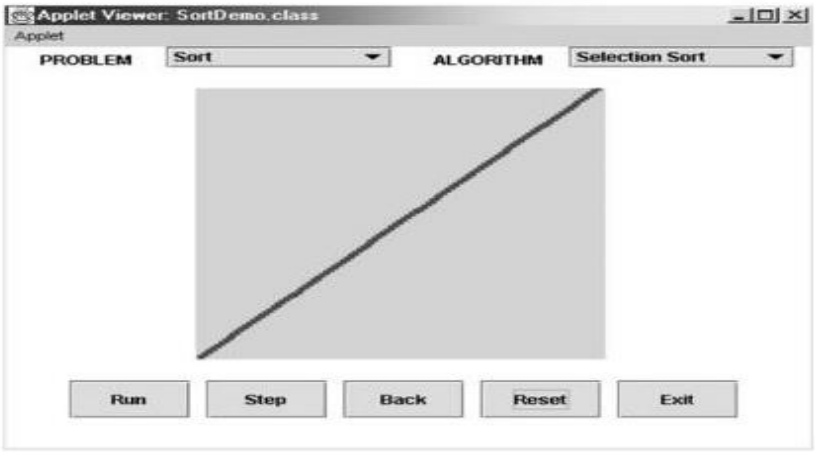
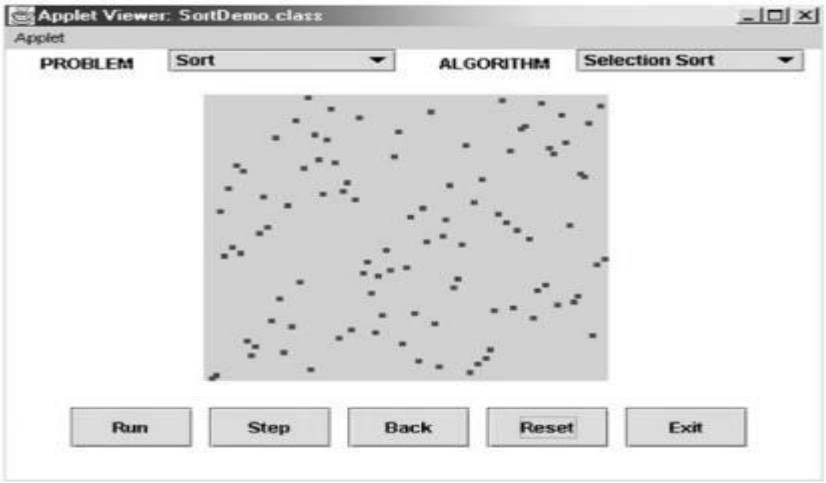


FIGURE 2.9 Initial and final screens of a typical visualization of a sorting algorithm using the scatterplot representation.



Applications:

1. **Education - Seeks to help students learning algorithms.**
2. **Research - Helps to uncover some unknown features of algorithms.**

IMPORTANT QUESTIONS

Part A

1. Show the notion of an algorithm. ***Dec 2009 / May 2013***
2. What are six steps processes in algorithmic problem solving? ***Dec 2009***
3. What is time and space complexity? ***Dec 2012***
4. Define Algorithm validation. ***Dec 2012***
5. Differentiate time complexity from space complexity. ***May 2010***
6. What is a recurrence equation? ***May 2010***
7. What do you mean by algorithm? ***May 2013***
8. Define Big Oh Notation. ***May 2013***
9. What is average case analysis? ***May 2014***
10. Define program proving and program verification. ***May 2014***
11. Define asymptotic notation. ***May 2014***
12. What do you mean by recursive algorithm? ***May 2014***
13. Establish the relation between O and Ω ***Dec 2010***
14. If $f(n) = a_m n^m + ... + a_1 n + a_0$. Prove that $f(n)=O(n^m)$.***Dec 2010***
15. Define the Fundamentals of Algorithmic Problem Solving
16. Short notes on Important Problem Types
17. .Define Fundamentals of the Analysis of Algorithm Efficiency
18. Show the Analysis Framework
19. Define Asymptotic Notations and its properties
20. Define Mathematical analysis for Recursive and Non-recursive algorithms**.**

Part B

1. Explain the notion of algorithm. ***May 2014***
2. Explain the fundamentals of algorithm. ***May 2014***
3. Find the time complexity and space complexity of the following problems. Factorial using recursion and compute the nth Fibonacci number using iterative statements. ***Dec 2012***
4. Solve the following recurrence relations: ***Dec 2012***

1. $T(n)= \begin{cases} 2T(n/2)+3 & n>2 \\ 2 & n=2 \end{cases}$

2. $T(n)= \begin{cases} 2T(n/2)+cn & n>1 \\ a & n=1 \end{cases}$ where a and c constants

5. Distinguish between Big Oh, Theta and Omega notation. ***Dec 2012***
6. Analyse the best case, average and worst case analysis for linear search. ***Dec 2012***
7. Explain how time complexity is calculated. Give an example. ***Apr 2010***
8. Elaborate on asymptotic notation with example. ***Apr 2010***
9. Briefly explain the time complexity, space complexity estimation ***June 2013***
10. Write linear search algorithm and analyse its complexity. ***June 2013***
11. Show the following equalities are correct ***June 2013***
    i. $5n^2-6n = \Phi(n^2)$
    ii. $n!=O(n^n)$
    iii. $n^3+10^6n^2=\Theta(n^3)$
    iv. $2n^2 2^n + n \log n = \Theta(n^2 2^n)$
12. What are the features of an efficient algorithm? ***June 2014***
13. What is space complexity? With an example explain the components of fixed and variable part in space complexity. ***June 2014***
14. Explain towers of Hanoi problem and solve it using recursion. ***June 2014***
15. Derive the recurrence relation for Fibonacci series algorithm : also carry out time complexity analysis. ***June 2014***
16. Discuss in details about the efficiency of the algorithm with example. ***Mar 2014***

17.Explain the procedure to calculate the time complexity of binary search using non-recursive Algorithm.

18. Explain briefly the time complexity and space complexity estimation. ***Nov 2010***

19. Write a linear search algorithm and analyse its best, worst and average case time complexity.

20. Prove that for any two functions f(n) and g(n), we have f(n)-> Θ(g(n)) if and only if f(n) - > O(g(n)) and f(n) ->Ω(g(n)) ***Nov 2010***

21.Explain the Mathematical analysis for non-recursive algorithm

ANNA UNIVERSITY APRIL/MAY 2015

PART-A

1.write algorithm to find the number of binary digits in the binary representation of a positive decimal integer **Part A – Refer Q. No. 56**

2.write down the properties of asymptotic notations. **Part A – Refer Q. No. 57**

PART-B

11.(a)if you have to solve the searching problem for a list of n numbers, how can you take advantage of the fact that the list is known to be sorted? Give separate answers for

(i) List represented as arrays

(ii) List represented as linked list Compare the time complexity involved in the analysis of both the algorithms **Refer Q. No. 27**

OR

(b)(i)Derive the worst case analysis of merge sort using suitable illustration**Refer Q.No. 28**

(ii) Derive a loose bound on the following equation:

$F(x)=35\ x^8 -22x^7+14x^5 -2x^4 -4x^2+x-15$ **Q.No. 15**

ANNA UNIVERSITY NOV/DEC 2015

PART-A

1.The *(log n)*th smallest number of n unsorted numbers can be determined in O(n) average-case time (True/False) **Refer Q. No. 60**

2. Fibonacci algorithm and its recurrence relation **Refer Q. No. 61**

PART-B

11.(a)(i)write Insertion sort algorithm and estimate its running time.(8) **Refer Q. No. 12**

(ii)find the closest asymptotic tight bound by solving the recurrence equation *T(n)=8T(n/2)+n2* with *(T(1)=1)* using recursion tree method.[Assume that T(1)ЄΘ(1)]

Refer Q. No. 14

OR

(b)(i)Suppose W satisfies the following recurrence equation and base case (where c is a constant):*W(n)=c.n+W(n/2)* and *W(1)=1*.What is the asymptotic order of W(n).

Refer Q. No. 14

(ii)Show how to implement a stack using two queues. Analyze the running time of the stack Operations. **Refer Q. No. 13**

ANNA UNIVERSITY APRIL/MAY 2016

PART-A

1. Give the Euclid's algorithm for computing gcd(m, n) **Refer Q. No. 58**

2. Compare the order of growth n(n-1)/2 and n2. **Refer Q. No. 59**

PART-B

1. a.(i) Give the definition and Graphical Representation of O-Notation.(8) **Refer Q. No. 4**

(ii) Give an algorithm to check whether all the Elements in a given array of n elements are distinct. Find the worst case complexity of the same. (8) **Refer Q. No.5(2)**

OR

(b) Give the recursive algorithm which finds the number of binary digits in the binaryrepresentation of a positive decimal integer. Find the recurrence relation and complexity. (16) **Refer Q. No.6(3)**

ANNA UNIVERSITY NOV/DEC 2016

PART-A

1.Design an algorithm to compute the area and circumference of a circle **Refer Q. No. 63**

2.Define recurrence relation. **Refer Q. No. 45**

PART-B

11.(a)(i)Use the most appropriate notation to indicate the time efficiency class of sequential search algorithm in the worst case,best case and the average case. **Refer Q. No. 17**

(ii) State the general plan for analyzing the time efficiency of nonrecursive algorithm and explain with an example(8) **Refer Q. No. 5**

(b) Solve the recurrence relations **Refer Q. No. 16**

$X(n) = x(n-1) + 5$ for $n > 1$ $x(1)=0$

$X(n) = 3x(n-1)$ for $n > 1$ $x(1)=4$

$X(n) = x(n-1) + n$ for $n > 0$ $x(0)=0$

$X(n) = x(n/2) + n$ for $n > 1$ $x(1)=1$ (solve for $n= 2^k$)

$X(n) = x(n/3) + 1$ for $n > 1$ $x(1)=1$ (solve for $n= 3^k$) (16)

ANNA UNIVERSITY APRIL/MAY 2017

PART-A

1. What is an algorithm? **Refer Q. No. 1**
2. Write an algorithm to compute the greatest common divisor of two numbers **Refer Q. No. 10**

PART-B

1. Explain briefly Big oh notation , Omega notation and Theta notation give an example **Q. No. 30**

2.Briefly explain the mathematical analysis of recursive and non recursive algorithm**Q.No.35 & 40**

ANNA UNIVERSITY NOV/DEC 2017

PART-A

1. How to measure an algorithm's running time ? **Refer Q. No. 21**
2. What do you mean by "worst case efficiency: of an algorithm. **Refer Q. No. 55**

PART-B

1. Discuss the steps in Mathematical analysis for recursive algorithms. Do the same for finding Factorial of a number **Refer Q. No. 6**

2. What are the Rules of Manipulate Big-Oh Expression and about the typical growth rates of algorithms? **Refer Q.No.4**

ANNA UNIVERSITY MAY/JUNE 2018

PART-A

1. Give the Euclid's algorithm for computing gcd of two numbers. **Refer Q. No. 58**
2. What is a basic operation? **Refer Q. No. 63**

PART-B

1. a) Define Big O notation, Big Omega and Big Theta Notation. Depict the same graphically and explain. **Refer Q.No.4**

b) Give the general plan for Analyzing the time efficiency of Recursive Algorithms and use recurrence to find number of moves for Towers of Hanoi problem. **Refer Q.No.6**

ANNA UNIVERSITY NOV/DEC 2018

PART-A

1. Define algorithm. List the desirable properties of an algorithm. **Refer Q. No. 64**
2. Define best, worst, average case time complexity. **Refer Q. No. 65**

PART-B

1. (i) Prove that if g(n) is Ω(f(n)) then f(n) is O(g(n)). **Refer Q.No.18**
   (ii) Discuss various methods used for mathematical analysis of recursive algorithms. **Refer Q.No.6**
2. Write the asymptotic notations used for best case, average case and worst case analysis of algorithms. Write an algorithm for finding maximum element in an array. Give best, worst and average case complexities. **Refer Q.No.4**

ANNA UNIVERSITY APRIL/MAY 2019

PART-A

1. How do you measure the efficiency of an algorithm? - **Refer Q.No.29**
2. Prove that the of f(n)=o(g(n)) and g(n)=o(f(n)),then f(n)=θ g(n). **- Refer Q.No.66**

PART-B

1.a) (i) solve the following recurrence equation: **- Refer Q.No.8**
   1.T(n)=T(n/2)+1,where $n=2^k$ for all k>=0
   2.T(n)= T(n/3)+ T(2n/3)+cn,where 'c' is a constant and 'n' is the input size.
(ii) Explain the steps involved in problem solving. **- Refer Q.No.8**

2.(i) write an algorithm for determining the uniqueness of an array. Determine the time complexity of your algorithm. **- Refer Q.No.5**
(ii) Explain time-space trade off of the algorithm designed **- Refer Q.No.3**

ANNA UNIVERSITY NOV/DEC 2019

PART-A

1. State the transpose symmetry property of O and Ω **- Refer Q.No.66**
2. Define recursion **- Refer Q.No.67**

PART-B

1. a) i) Solve the following recurrence equations using iterative method or tree **Refer Q.No.6**
   ii) Elaborate asymptotic analysis of an algorithm with an example. **Refer Q.No.4**
2. b) write an algorithm using recursion that determines the GCD of two numbers. Determine the time and space complexity **- Refer Q.No.1.A**

ANNA UNIVERSITY NOV/DEC 2021

PART-A

1. Define algorithm with its properties. **Refer Q.No.1**
2. List the reasons for choosing an approximate algorithm. **Refer Q.No.68**

PART-B

1. a) i) Consider the problem of counting, in a given text the number of substrings that start with an Aand end with a B. For example, there are four such substrings in CABAAXBYA. Design a brute-force algorithm for this problem and determine its efficiency class. **Refer Q.No.6**

ii) "The best-case analysis is not as important as the worst-case analysis of an algorithm". Yes or No ? Justify your answer with the help of an example**. Refer Q.No.11**

2. b) (i) Solve : T(n) = 2T(n/2) + n3**. Refer Q.No.11**

(iii) Explain the importance of asymptotic analysis for running time of an algorithm with an example. **Refer Q.No.4**

ANNA UNIVERSITY NOV/DEC 2021

PART-A

1. Define the notation big-Omega. **Refer Q.No.14**
2. What is meant time complexity of an algorithm? **Refer Q.No.7**

PART-A

11. a) Outline worst case running time, best case running time and average case running time of an algorithm with an example?

b) Outline a recursive algorithm and non recursive algorithm with an example.

Refere Q.No.35 & 40