chapter-1

1. Introduction

1.1 Algorithm Definition:

An algorithm is a set of steps, which if performed will accomplish a particular task.

An algorithm must meet the following criteri

i. Input:

Zero, one or more than one finite inputs are needed

ii. Output:

At least one output is produced.

iii. Definiteness:

Every instruction should be clear and unambiguous.

iv) Finiteness:

Tracing every instruction in an algorithm must termi-nate in finite number of steps.

v) Effectiveness:

Every instruction must be very basic so that in every case, a person with a paper and pencil can perform the various steps.

Algorithms which meet criteria definiteness and finiteness is called Computational procedure.

Study of algorithm includes several areas of research;

i) How to devise algorithms?

→ Though designing algorithm is an art because the process of designing algorithm cannot be fully automated, there are different algorithm design techniques which produce good algorithms. In the past, many algorithm design strategies have been invented which can be used for designing good algorithm for solving problems.

ii) How to validate algorithms?

→ Validation of an algorithm means proving for its correctness i.e. we feed algorithm with correct sample data and in all cases, it must produce correct outputs. A mathematical proof, rather provides a good validation for any algorithm.

iii) How to analyze algorithm?

→ An algorithm is analyzed for its performance. Performance is found out by computing total execution time and total memory required for the complete execution of the algorithm.

iv) How to test algorithm?

→ Testing an algorithm involves two steps:

a) Debugging
b) Profiling

Debugging is the process of executing a program with

Sample test date with a view to find faulty output which if found should be corrected.

Performance measurement is the process of executing a correct program on a computer completely and finding out total time taken and memory used.

1.2 Algorithm Specification

Algorithms are specified using the following two techniques or tools:

i) Flowchart

ii) Pseudocode

Pseudo Code Convention:

Flowchart has a serve limitation in that it is useful for small algorithms. Therefore, algorithms are written using pseudocode. It uses english like words for the ease of understanding.

We present most of our algorithms using a pseudocod that resembles C.

i) Comments begin with // and continue until the end of line.

ii) Blocks are indicated with matching braces { and }. Statemen are delimited by ;

iii) An identifier begins with a letter. The data types of variables are not explicitly declared.

iv) Assignment of values to variables is done using the assignment Statement.

v) There are two boolean values true and false. In order to produce these values, the logical operators and, or and not and the relational operators $<$, $\leq$, $=$, $\neq$, $\geq$ and $>$ are provided.

vi) Elements of multidimensional arrays are accessed using [and].

vii) The following looping statements are employed : for, while and repeat-until.

```
while (condition) do
{
  (statement 1)
    :
  (statement n)
}
```

```
repeat
  (statement 1)
    ;
    ;
  (statement n)
until (condition)
```

```
for variable := value 1 to value 2
{
  (statement 1)
    :
  (statement n)
}
```

viii) A conditional statement has the following forms:

if (condition) then (Statement)

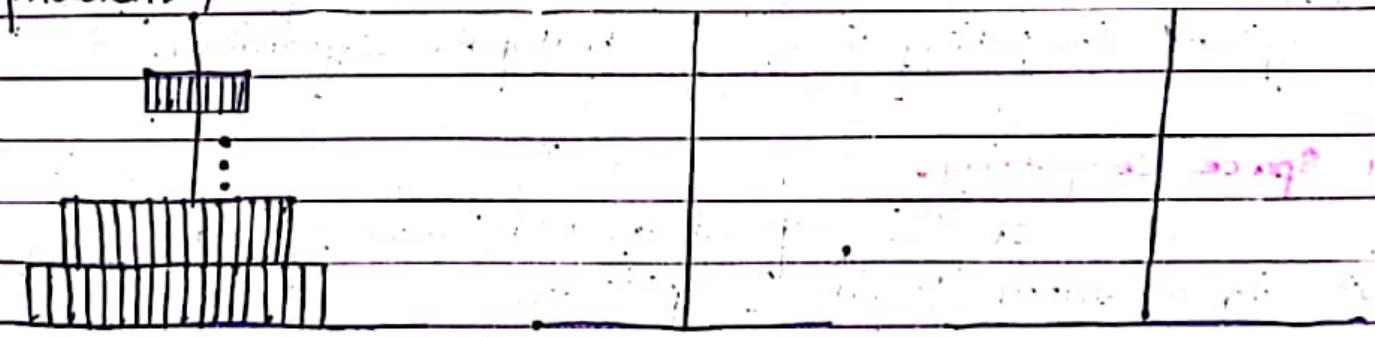
if (Condition) then (Statement 1) else (Statement 2)

ix) Input and output are done using the instructions read and write.

1.2.2 Recursive Algorithms

A recursive function is a function that is defined in terms of itself. Similarly, an algorithm is said to be recursive if the same algorithm is invoked in the body. An algorithm that calls itself is direct recursive. Algorithm A is said to be indirect recursive if it calls another algorithm which in turn calls A.

In the first example, we consider the Towers of Hanoi problem,



Assume that the number of disks is n. To get the largest disk to the bottom of tower B, we move the remaining n-1 disks to tower C. and then move the largest to tower B. Now we are left with the task of moving the disks from tower C to tower B.

```
Algorithm TowersOfHanoi (n, x, y, z)
// move the top n disks from tower x to tower y
{
    if (n >= 1) then
    {
        TowerOfHanoi (n-1, x, z, y);
        write ("move top disk from tower", x, "to top of tower", y);
        Towers of Hanoi (n-1, z, y, x);
    }
}
```

1.3 Performance Analysis:

Performance analysis is based on evaluation of total amount of memory space and total Computing time taken by the algorithm to run to completion.

Lesser the amount of memory using and Computing time, the better the algorithm.

1.3.1 Space Complexity:

It is defined as total memory required to run an algorithm to its completion.

Example: Find out space complexity for an algorithm that sums all elements of an integer array.

```
Algorithm Sum(a,n)  // a is an integer array having n elements
{
    int s, n, i;
    s := 0;
```

```
for i:=1 to n do
s:= s+a[i];
return(s);
}
```

Memory required i.e. space required $s(s) = n+3$.

1.3.2 Time Complexity :

It is defined as total execution time for an algorithm to run to its completion.

Example: Find time complexity of the algorithm to add element of an integer array.

We shall use the step table approach.

Stepno.	Instruction	s/e	f	s/e * f = Total
1.	Algorithm Sum(a,n)	0	–	0
2.	{	0	–	0.
3.	int s, n, i ;	0	0	0
4.	s:=0;	1	1	1
5.	for i:=1 to i<=n do	1	n+1	n+1
6.	s:= s+a[i]	1	n	n
7.	return(s);	1	1	1
8.	}	0	–	0

Total no. of steps = $1+n+1+n+1$

$= 2n+3$

Example: Write algorithm for calculating sum of two matrices also find its time complexity.

```
Algorithm Sum(a, b, c, m, n)   // a and b are matrices having m rows and n column to be added resulting into matrix c.
{
    int i, j;
    for i:= 1 to m
        for j:= 1 to n
            c(i, j):= a(i, j) + b(i, j);
    return c;
}
```

step table

step	Instruction	s/e	f	s/e * f = Total
1.	Algorithm Sum(a,b,c, m,n)	0	-	0
2.	{	0	-	0
3.	int i, j;	0	0	0
4.	for i:=1 to m	1	m+1	m+1
5.	for j:=1 to n	1	m*(n+1)	m(n+1)
6.	c(i, j):= a(i, j) + b(i, j);	1	mn	mn
7.	return c;	1	1	1
8.	}	0	-	0

Total Count = $(m+1) + m(n+1) + mn + 1$

$= m + 1 + mn + m + mn + 1$

$= 2m + 2 + 2mn$

$= 2m(1+n) + 2$

1.3.3 Asymptotic Notation (O, Ω, θ)

Asymptotic notations permit substantial simplifications even when we are interested in measuring something more tangible than computing time such as the number of times a given instruction in a program is executed.

These notations are called asymptotic "because they deal with the behaviour of functions in the limit ie. for sufficiently large values of their parameters. The following notations are used to express the asymptotic complexity.

A) Big oh-notation [O-notation]:

A function T(n) is said to be big oh of another function g(n) if and only if there exists constants c and n_0 such that $T(n) \leq c * g(n)$, c and n_0 are positive integers and $n \geq n_0$.

$T(n) = Og(n)$ implies the behaviour of f(n) with reference to another function g(n) and it signifies that g(n) is the upper bound of f(n) as shown in figure for all $n \geq n_0$

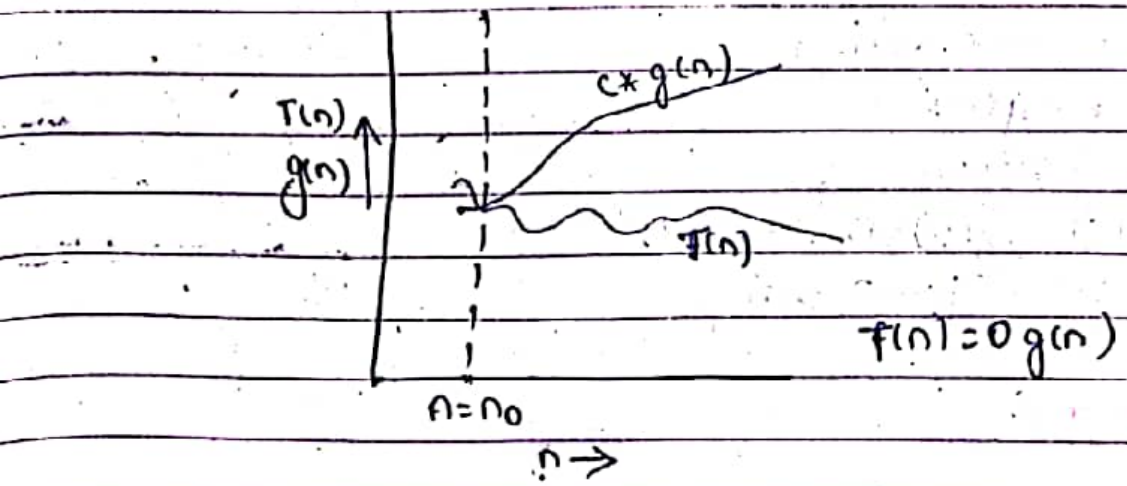


fig: Graphical notation of O-notation

Example: Express the asymptotic notations using big O of f(n) defined as follows. In each case decide the value of C and n_0.

a) $f(n) = 3n + 2$

Soln:

According to O-notation

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

$3n + 2 \leq 4n$

$\therefore c = 4, g(n) = n, n_0 = 2$

$f(n) = O(n)$ for $n \geq 2$.

b) $f(n) = 3n + 3$

Soln:

According to O-notation

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

$3n + 3 \leq 4n$

$\therefore c = 4, g(n) = n, n_0 = 3$

$f(n) = O(n)$ for $n \geq 3$

c) $f(n) = 100n + 6$

Soln:

According to O-notation

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

$100n + 6 \leq 101n$

$\therefore c = 101, g(n) = n, n_0 = 6$

$f(n) = O(n)$ for $n \geq 6$

d) $f(n) = 10n^2 + 4n + 2$

Soln:

According to O-notation

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

$10n^2 + 4n + 2 \leq 11n^2$

$\therefore c = 11$, $g(n) = n^2$, $n_0 = 5$

$f(n) = O(n^2)$ for $n \geq 5$

e) $f(n) = 1000n^2 + 100n + 6$

Soln:

According to O-notation

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

$1000n^2 + 100n + 6 \leq 1001n^2$

$\therefore c = 1001$, $g(n) = n^2$, $n_0 = 100$

$f(n) = O(n^2)$ for all $n \geq 100$

g) $f(n) = c$, where c is a positive Constant.

Soln:

According to O-notation,

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

$c \leq c * g(n)$

$g(n) = 1$

$f(n) = O(1)$

B) Big omega notation (Ω-notation):

A function $T(n)$ is said to be Ω of $g(n)$ written as $T(n) = \Omega(g(n))$, if and only if there exist positive integer values c and n_0 such that $T(n) \geq c(g(n))$, where $n \geq n_0$ (read as "f of n is omega of g of n")

$f(n) = \Omega(g(n))$ implies the behaviour of $f(n)$ with reference to another function $g(n)$ and it signifies that $g(n)$ is the lower bound of $f(n)$ as shown below:

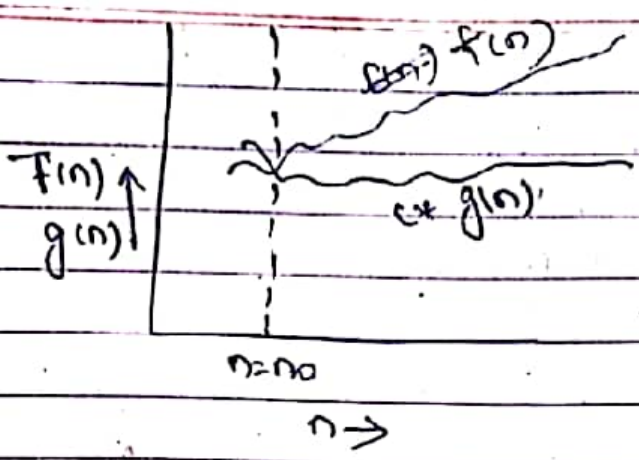


fig: Graphical representation of Ω-notation

Examples: Give the Ω-notation for the following.

a) $f(n) = 3n+2$

Soln:

According to Ω-notation,

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

$3n+2 \geq 3n$

$\therefore c=3,\ g(n)=n,\ n_0=1$

$\therefore f(n) = \Omega(n)$ for all $n \geq 1$

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

$3n+2 \geq 2n$

$g(n)=n,\ c=2,$

$n_0=1$

b) $f(n) = 3n+3$

Soln:

According to Ω-notation,

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

$3n+3 \geq 3n$

$c=3,\ g(n)=n,\ n_0=1$

$f(n) = \Omega(n)$ for all $n \geq 1$

c) $f(n) = 100n+6$

Soln:

According to Ω-notation;

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

$100n + 6 \geq 100n$

$c = 100,\ g(n) = n,\ n_0 = 1$

$\therefore f(n) = \Omega(n)$ for $n \geq 1$

d) $f(n) = 10n^2 + 4n + 2$

Soln:

According to Ω-notation,

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

$10n^2 + 4n + 2 \geq 10n^2$

$c = 10,\ g(n) = n^2,\ n_0 = 1$

$f(n) = \Omega(n^2)$ for $n \geq 1$

e) $f(n) = 1000n^2 + 100n + 6$

Soln:

According to Ω-notation,

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

$1000n^2 + 100n + 6 \geq 1000n^2$

$c = 1000,\ g(n) = n^2,\ n_0 = 1$

$f(n) = \Omega(n^2)$ for $n \geq 1$.

c) Big theta notation (Θ-notation):

The function $f(n) = \Theta(g(n))$ (read as "f of n is theta of g of n") if and only if there exist positive constants c_1, c_2 and n_0 such that

$c_2 * f(n) \leq T(n) \leq c_1 * g(n)$ for all n, $n \geq n_0$

$f(n) = \Theta(g(n))$ implies another behaviour of $T(n)$ with reference

to another function g(n) and f(n). It signifies that T(n) is bounded both lower and upper by f(n) and g(n) respectively as shown below:

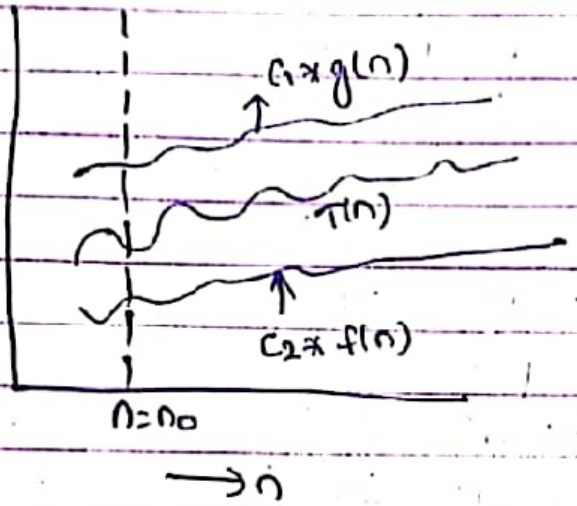


fig: Graphical representation of Θ-notation.

Example: Give Θ-notation for the following.

a) $f(n) = 3n + 2$

Soln:

According to Θ-notation

$C_2 * f(n) \leq T(n) \leq C_1 * g(n)$

$3n \leq 3n+2 \leq 4n$

$\therefore C_2 = 3,\ C_1 = 4$ and $f(n) = n,\ g(n) = n$

$n_0 = 1 \quad n_0 = 2$

$f(n) = \Theta(n)$ for all $n \geq 2$

b) $f(n) = 100n + 6$

Soln:

$100n + 6 \geq 100n$ for all $n \geq 1$

$C_2 = 100$

$100n + 6 \leq 101n$ for all $n \geq 6$

$c_1 = 101$

$\therefore f(n) = O(n)$ for all $n \geq 6$

c) $f(n) = 10n^2 + 4n + 2$

Soln:

$10n^2 + 4n + 2 \geq 9n^2$ for all $n \geq 1$

$c_2 = 9$

$10n^2 + 4n + 2 \leq 11n^2$ for all $n \geq 5$

$c_1 = 11$

$\therefore f(n) = \Theta(n^2)$ for all $n \geq 5$

1.3.4 Practical Complexities :

We have seen that the time Complexity of an algorithm is generally some function of the instance characteristics. The Complexity function can also be used to compare two algorithms p and Q that perform same task.

To get a feel for how the various functions grow with n, we should draw table as below

log n	n	n log n	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296

fig: function values

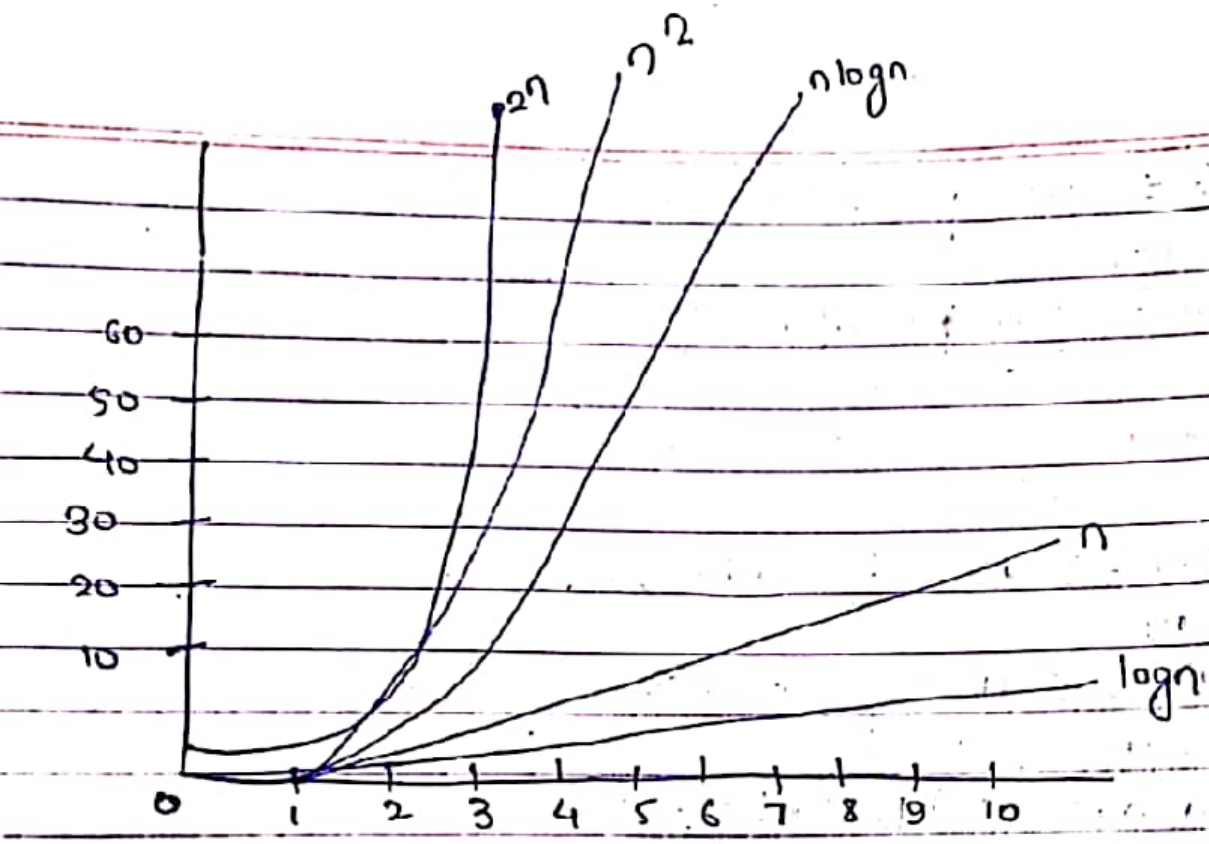


fig: Plot of function values

1.3.5 Performance measurement:

performance measurement is concerned with obtaining the space and time requirements of a particular algorithm. Compilation time is not taken into account of performance measurement. Space required to run an algorithm is also not calculated.

Each program is compiled once and then executed several times. Certainly, the space and time needed for compilation are important during program testing, when more time is spent on this task than in running the compiled code.

Suppose we wish to measure the worst-case performance of the sequential search algorithm. Before we do this, we need to

1) decide the values of n for which the times are to be obtained
2) determine for each of the above values of n, the data that exhibit the worst-case behaviour.

```
Algorithm seqsearch (a, x, n)
// search for x in a[1:n]. a[0] is used as additional spa
{
    i:= n;
    a[0]:= x;
    while(a[i] ≠ x) do i:= i-1;
    return i;
}
```

```
Algorithm Seqsearch(a, x, n)
{
    i=n
    a[0]= x;
    while(a[i] ≠ x) then i= i-1;
    return i;
}
```

Chapter-3

Greedy Method

3.1 The General Method:

The greedy method is the most straightforward design technique. Most, though not all, of these problems have n inputs and require us to obtain a subset that satisfies some constraints. Any subset that satisfies these constraints is called a feasible solution. We need to find a feasible solution that either maximizes or minimizes a given objective function. A feasible solution that does this is called an optimal solution.

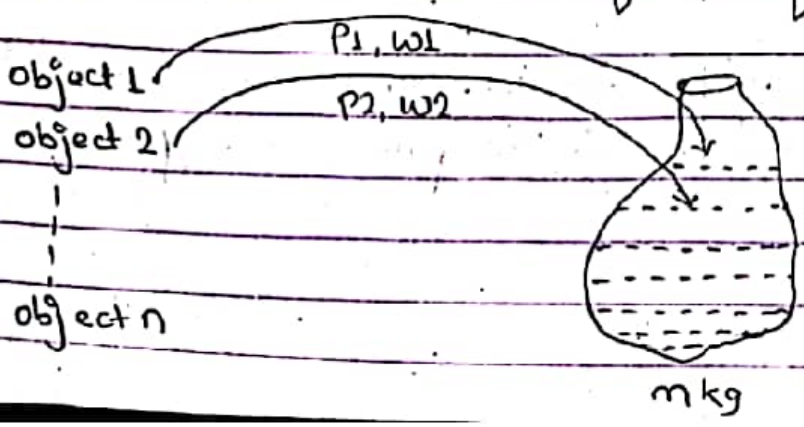
The greedy method suggests that one can devise an algorithm that works in stages, considering one input at a time. At each stage, a decision is made regarding whether a particular input is in an optimal solution. This is done by considering the inputs in an order determined by some selection procedure. If the inclusion of the next input into the partially constructed optimal solution will result in an infeasible solution, then this input is not added to the partial solution. Otherwise, it is added. The selection procedure itself is based on some optimization measure. This measure may be the objective function. This version of the greedy techniques is called the subset paradigm.

The algorithm for Greedy Method is given below:

```
Algorithm Greedy (a, n)
// a[1:n] contains the n inputs
{
    Solution := 0; // Initialize the solution [x[i]=0]
    for i := 1 to n do
    {
        x := Select (a);
        if feasible (Solution, x) then
        Solution := Union (Solution, x); (union - add)
    }
    return Solution;
}
```

3.2 knapsack problem:

Let us apply the greedy method to Solve the knapsack problem. Suppose that there are as many as n objects and a knapsack having a capacity m. Let object i has a weight w_i. Each object when put into the knapsack earns a profit of $p_i x_i$ where p_i is the profit per unit for object i. The objective is to obtain a filling of the knapsack that maximizes the total profit earned. Since the knapsack capacity is m, we require the total weight of all chosen objects to be at most m. Let x_i be the fraction of object assigned.



Objective function:

Maximize $z(\text{profit}) = \sum_{1 \le i \le n} p_i x_i$

Subject to constraints $\sum_{i \le i \le n} x_i w_i \le m$

and $0 < x_i < 1, \quad 1 \le i < n$ i.e. $x_1 = 0$ ✗

$x_2 = 1$ ✓

$x_3 = 1/2$ ✓

$x_4 = -0.5$ ✗

→ A solution is said to be feasible if it meets all constraints.

→ optimal solution is that feasible solution for which the value of objective function is maximum. Since, the solution gives us a subset of objects assigned to the knapsack, it is also called subset paradigm. We decide to choose object to be assigned to the knapsack one at a time.

Example: Consider the following instance of the knapsack problem. $n = 3$, $m = 20$, $(p_1, p_2, p_3) = (25, 24, 15)$ and $(w_1, w_2, w_3) = (18, 15, 10)$.

Soln:

There are can be many feasible solution to the above problem based on criteria chosen for obtaining the solution.

Solution1:

objects are assigned to the knapsack according to profit per unit i.e. in the descending order of p_i.

x_1, x_2, x_3

$\left(1, \frac{2}{15}, 0\right)$

Total weight $= w_i x_i$

$= w_1x_1 + w_2x_2 + w_3x_3$

$= 18 \times 1 + \frac{2}{15} \times 15 + 10 \times 0$

$= 20$

Total profit $= \sum p_i x_i$

$= p_1x_1 + p_2x_2 + p_3x_3$

$= 25 \times 1 + 24 \times \frac{2}{15} + 15 \times 0$

$= 28.2$

Solution2:

objects are assigned in ascending order of weight.

$x_1 \; x_2 \; x_3$

$\left(0, \frac{10}{15}, 1\right)$

Total profit $= 0 + \frac{24 \times 10}{15} + 15 \times 1$

$= 31$

Solution 3:

objects are assigned in descending order of $\frac{p_i}{w_i}$.

$\frac{p_i}{w_i}$; $\frac{p_1}{w_1} = \frac{25}{18} = 1.38$

$\frac{p_2}{w_2} = \frac{24}{15} = 1.6$

$\frac{p_3}{w_3} = \frac{15}{10} = 1.5$

$x_1 \quad x_2 \quad x_3$

$\left(0, 1, \frac{5}{10}\right)$

Total profit $= 0 + 1 \times 24 + \frac{5}{10} \times 15 = 31.5$

- The optimal solution is as below: $x_1 = 0$

$x_2 = 1$

$x_3 = \frac{5}{10}$

optimal profit = 31.5

Algorithm for knapsack problem is as below:

```
Algorithm GreedyKnapsack (m,n)
// p[1:n] and w[1:n] contain the profits and weights
// respectively of n objects odered such that
// p[i] / w[i] ≥ p[i+1] / w[i+1].
// m is the knapsack size and x[1:n] is the solution vector
```

```
{
    for i:= 1 to n do x[i]:= 0.0;
    U:= m;
    for i:= 1 to n do
    {
        if (w[i] > U) then break;
        x[i]:= 1.0; U:= U - w[i];
    }
    if (i ≤ n) then x[i]:= U/w[i];
}
```

18	15	10
w[1]	w[2]	w[3]

25	24	15
p[1]	p[2]	p[3]

$\frac{p2}{w2}, \frac{p3}{w3}, \frac{p1}{w1}$

U := 20 − w[2]
= 20 − 15 = 5

// i = 2
i.e. 2 ≤ 3
x[3] = 5 / w[3] = 5/10

1	0.5	0
x[1]	x[2]	x[3]

3.3 Job Sequencing with Deadlines:

We are given a set of n jobs. Associated with job is an integer deadline. $d_i \geq 0$ and a profit $P_i > 0$. For any job i the profit P_i is earned if and only if the job is completed by its deadline. To complete a job, one has to process the job on a machine for one unit of time. Only one machine is available for processing jobs.

The problem to be solved here is to find out a subset of jobs all of which can be processed by their deadlines and give us the maximum profit.

Since a subset of jobs maximizing profit is to be found, it is a subset paradigm.

Example:

Let $n=4$, $(p_1, p_2, p_3, p_4) = (100, 10, 15, 27)$ and $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$. Find out optimal job sequences.

Soln:

four job cannot be performed, as they require four units of time. Similarly three jobs cannot be completed by their deadlines.

Since, the largest deadline is 2 and one job requires one unit of time, only two jobs at the most can be completed by their deadlines.

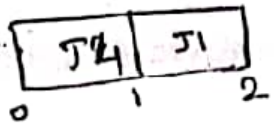
Feasible Solution	Processing Sequence	Total profit
(1, 2)	2, 1	110
(1, 3)	1, 3 or 3, 1	115
(1, 4)	4, 1	127
(2, 3)	2, 3	25
(3, 4)	4, 3	42
1	1	100
2	2	10
3	3	15
4	4	27

Optimal job sequence is (4, 1) and total profit = 127.

Example:

Let $n=5$, $(p_1, p_2, p_3, p_4, p_5) = (20, 15, 10, 5, 1)$ and $(d_1, d_2, d_3, d_4, d_5) = (2, 2, 1, 3, 3)$

Find out optimal job sequence.



Let $n = 4$, $(p_1, p_2, p_3, p_4) = (100, 10, 15, 27)$ and
$(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$

Find out optimal job sequence.

Soln:

J	assigned slots	job considered	action	profit
{}	none	1	assign to [1,2]	0
{1}	[1,2]	2	assign to [0,1]	100
{1,2}	[0,1] [1,2]	3	can't fit, reject	127
{1,2}	[0,1] [1,2]	4	can't fit reject	127

Arranging jobs in deecending order of profit

i.e. $(p_1, p_4, p_3, p_2) = (100, 27, 15, 10) \Rightarrow (2, 1, 2, 1)$
$J_1 \; J_4 \; J_3 \; J_2$

$\therefore$ profit = 127 and optimal sequence is $\{J_4, J_1\}$

1	1	100
2	2	10
3	3	15
4	4	27

Optimal job sequence is (4, 1) and total profit = 127.

Example:

Let $n = 5$, $(p_1, p_2, p_3, p_4, p_5) = (20, 15, 10, 5, 1)$ and
$(d_1, d_2, d_3, d_4, d_5) = (2, 2, 1, 3, 3)$.

Find out optimal job sequence.

Soln:

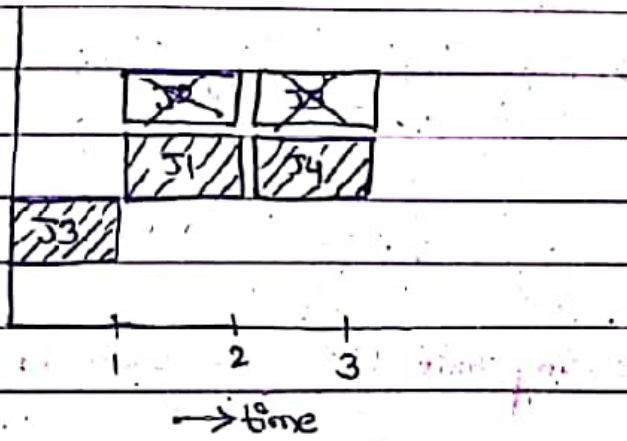
Ascending order of deadlines

J3 J1 J2 J4 J5

J3 J1 J2 J5 J4

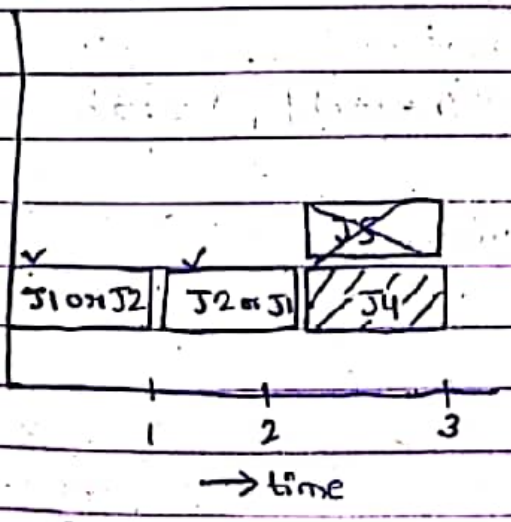
J3 J2 J1 J4 J5

J3 J2 J1 J5 J4



profit = 10 + 20 + 5 = 35

Job sequence = J3 J1 J4



Total profit = 20 + 15 + 5 = 40

Job sequence = J1 J2 J4

optimal sequence is (J1 J2 J4) and total profit = 40

High-level description of Job sequencing Algorithm

```
Algorithm GreedJob ( d , J, n)
// J is a set of jobs that can be completed by their deadlines
{
    J := {1};
    for i := 2 to n do
    {
        if (all jobs in J∪{i} can be Completed
            by their deadlines) then J := J∪{i}
    }
}
```

Greedy algorithm for sequencing unit time jobs with deadlines and profits :

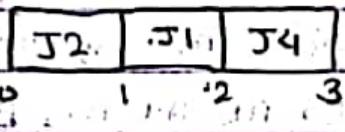
```
Algorithm JS(d, j, n)
// d[i] ≥ 1 , 1 ≤ i ≤ n are the deadlines , n ≥ 1 . The jobs are
// ordered such that p[1] ≥ p[2] ≥ ---- ≥ p(n) . J[i] is the ith
// job in the optimal solution , 1 ≤ i ≤ k .
// Also, at termination d[j[i]] ≤ d J[i+1]] , 1 ≤ i ≤ k
{
    d[0] := J[0] := 0; // Initialize
    J[1] := 1; // Include job 1
    k := 1;
    for i := 2 to n do;
    {
    // Consider jobs in non-increasing order of p[i].
    // Find position for i and check feasibility of insertion.
    r := k;
```

```
while ((d[J[r]] > d[i]) and (d[J[r]] ≠ r)) do r := r-1;
if (d[J[r]] ≤ d[i]) and (d[i] > r)) then
{
    // Insert i into J[]
    for q := k to (r+1) step -1 do J[q+1] := J[q];
    J[r+1] := i; k := k+1;
}
}
}
return k;
}
```

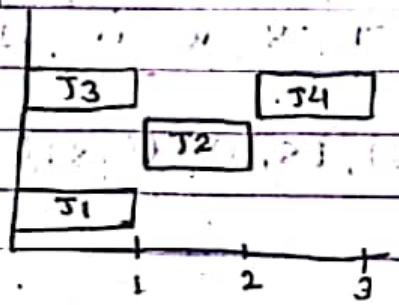
Example: Let n = 5, $(p_1, \ldots p_5) = (20, 15, 10, 5, 1)$ and $(d_1, d_2 \ldots d_5) = (2, 2, 1, 3, 3)$.

Soln:

J	assigned slots	job considered	action	profit
{ }	none	1	assign to [1,2]	0
{1}	[1, 2]	2	assign to [0,1]	20
{1,2}	[0,1], [1,2]	3	can't fit, reject	35
{1,2}	[0,1] [1,2]	4	assign to [2,3]	35
{1,2,4}	[0,1][1,2][2,3]	5	reject	40



The optimal solution is = {1, 3, 4} and profit = 40



Sequence = J[2], J[1], J[4]

Example: Given 10 activities along with their start and finish time as

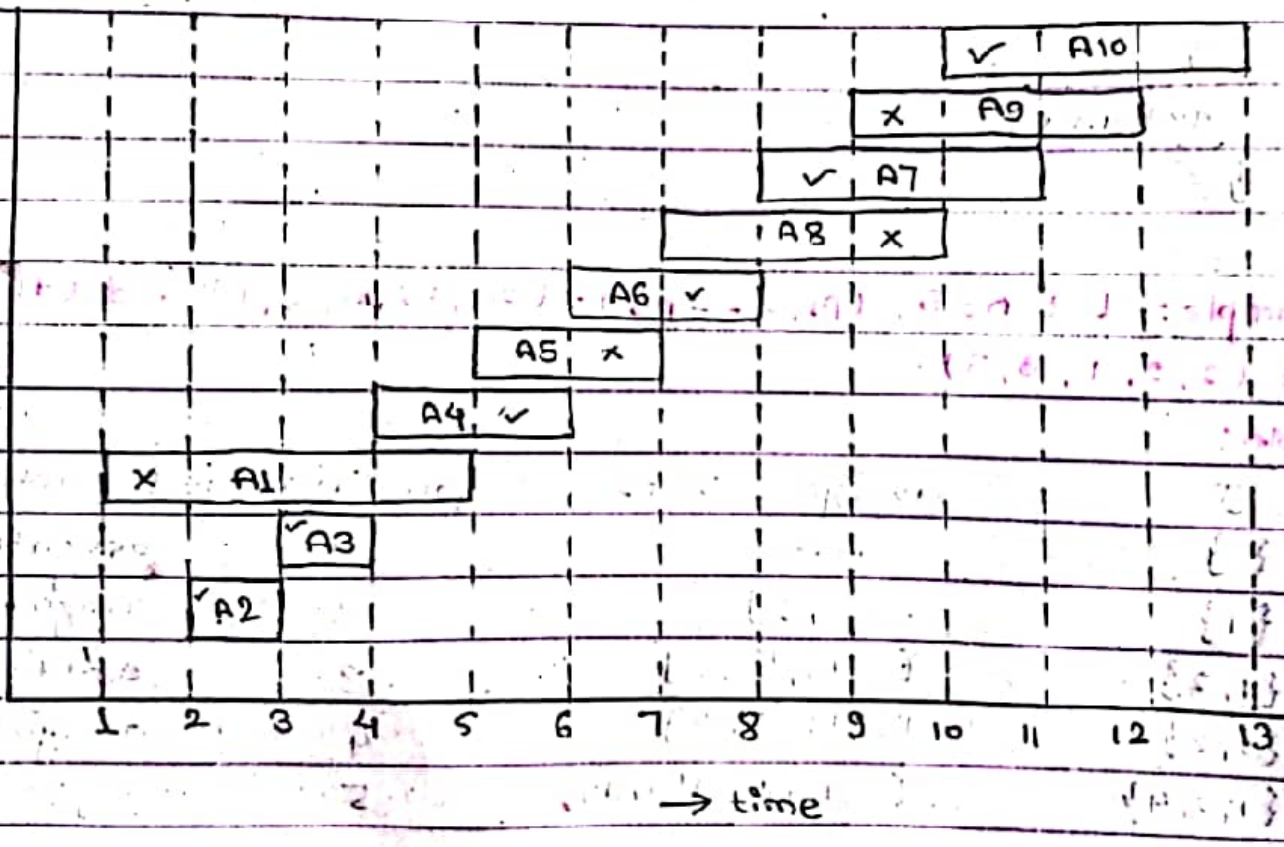
$S = \langle A1, A2, A3, A4, A5, A6, A7, A8, A9, A10 \rangle$

$si = \langle 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 \rangle$

$fi = \langle 5, 3, 4, 6, 7, 8, 11, 10, 12, 13 \rangle$

Perform activity scheduling using greedy strategy.

Soln:



Arrange the activities in increasing order of finish time.

$S_1 = \langle A2, A3, A1, A4, A5, A6, A8, A7, A9, A10 \rangle$

$si = \langle 2, 3, 1, 4, 5, 6, 7, 8, 9, 10 \rangle$

$fi = \langle 3, 4, 5, 6, 7, 8, 10, 11, 12, 13 \rangle$

(2, 3), (3, 4), (1, 5), (4, 6), (5, 7), (6, 8), (7, 10), (8, 11), (9, 12), (10, 13)

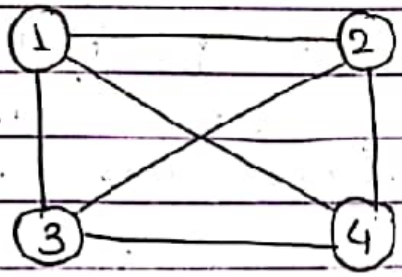
Sequence = A1 → A3 → A4 → A6 → A7 → A10

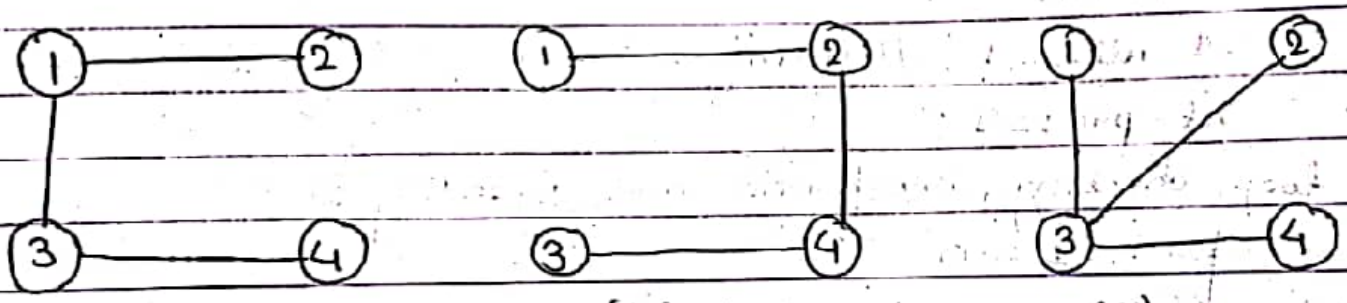
Algorithm

```
Algorithm Activity-schedule (s, f)
// the above algorithm computes the optimum set of activities.
// for determining interferences variable 'pre' holds the index of the
// recently scheduled activity at any time. The start and finish time
// of activities are denoted by 's' and 'f' respectively. Activities are
// sorted by their finish times
// Initialization
    Set Alist := 1 ; // Activity 1 is scheduled first
    Set pre := 1 ;
// loop, checking, interference and appending list
    for i := 2 to n
    {
        if s[i] ≥ f [pre] then
        {
            append i to Alist;
            pre := i;
        }
    return Alist;
}
```

3.4. Minimum Cost Spanning Tree :

Let $G_1 = (V, E)$ be an undirected connected graph. A sub-graph $t = (V, E')$ of G_1 is a spanning tree of G_1 if and only if t is a tree.

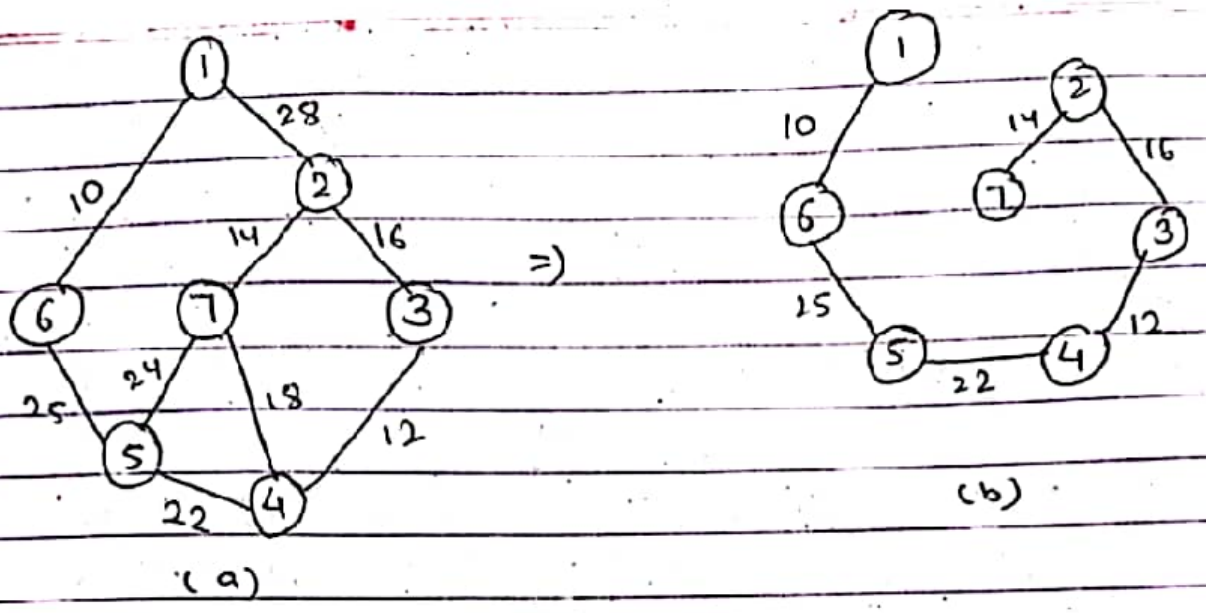




(a) (b) (c)

Above figure shows the complete graph of on four nodes together with three of its spanning trees.

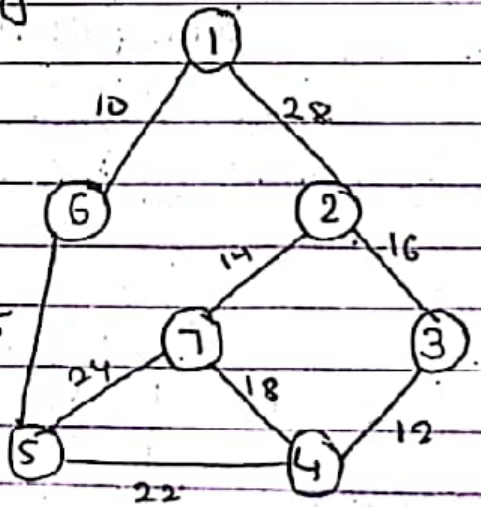
Spanning trees have many applications. for example, they can be used to obtain an independent set of circuit for an electric network. Another application of spanning trees arises from the property that a spanning tree is a minimal sub-graph G' of G such that $V(G') = V(G)$ and G' is connected. Any connected graph with n vertices must have at least n-1 edges and all connected graphs with n-1 edges are trees. If the nodes of G represent cities and edges represent possible communication links connecting two cities, then the minimum no. of links needed to connect the n cities is n-1. The spanning trees of G represent all feasible choices.



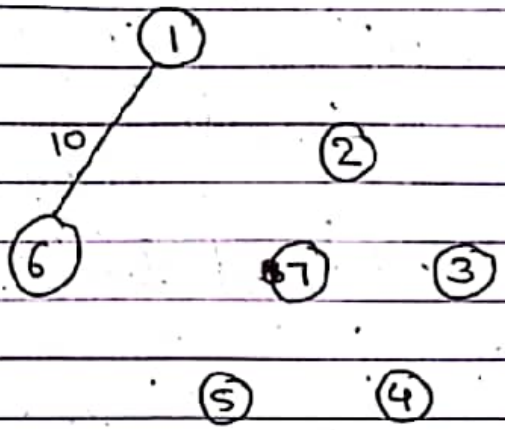
3.4.1 Prim's Algorithm:

A greedy method to obtain a minimum-cost spanning tree builds this tree edge by edge. The next edge to include is chosen according to some optimization criteria. The simplest criteria is to choose an edge that results in a minimum increase in the sum of the costs of the edges so far included: In the first, the set of edges selected so far form a tree. The next edge (u, v) to be included in A is a minimum-cost edge not in A with the property that $A \cup \{(u, v)\}$ is also called a tree. The corresponding algorithm is known as prim's algorithm.

Eg:

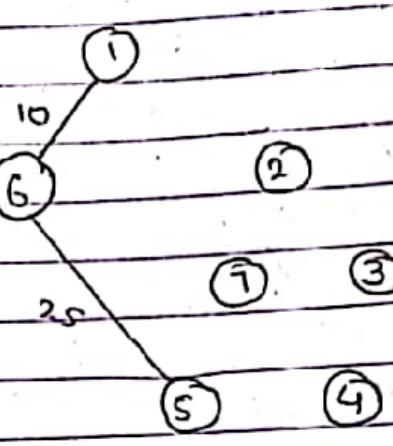


Soln: root node = 1



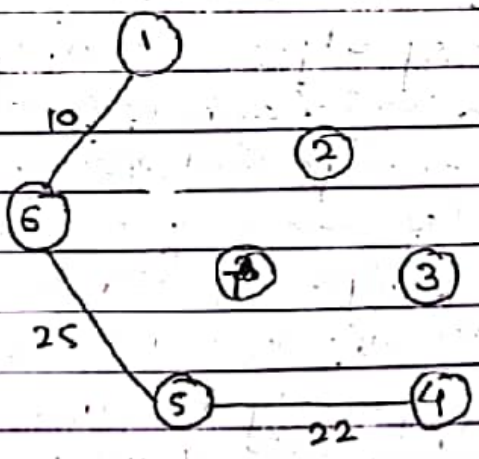
(a)

Min = {28, 25}



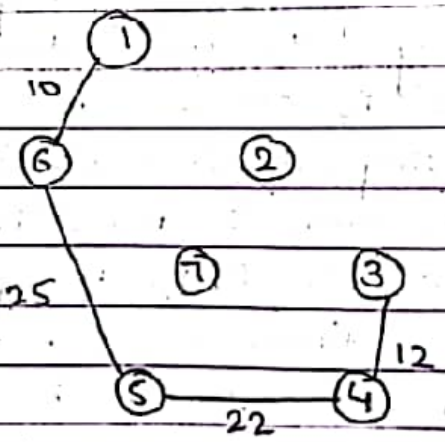
(b)

Min = {28, 22, 24}



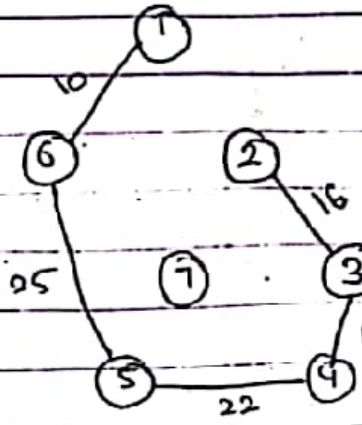
(c)

Min = {28, 24, 18, 12}

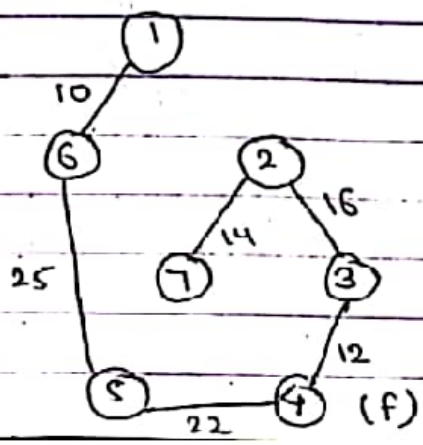


(d)

min = {28, 24, 18, 16}



(e) Min = {28, 24, 18, 14}



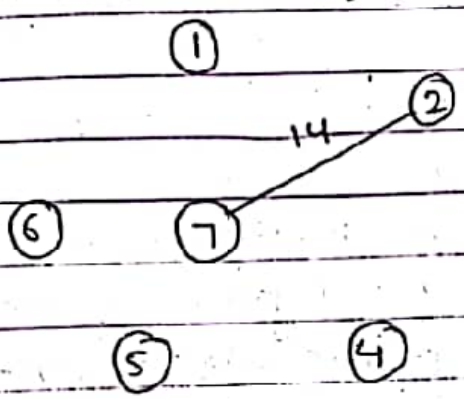
(f)

Minimum cost = 10 + 25 + 22 + 12 + 16 + 14

= 99

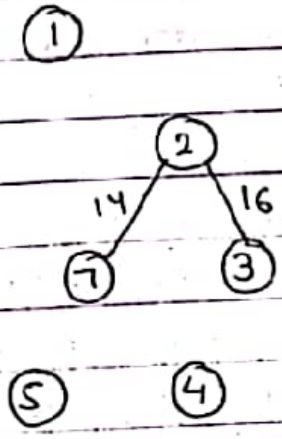
OR

root node = 7 (take)



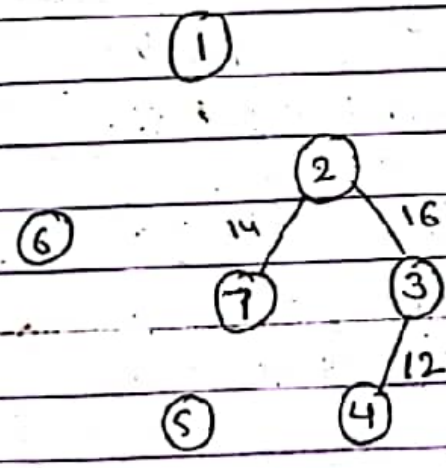
(a)

Min = { 18, 24, 28, 16 }



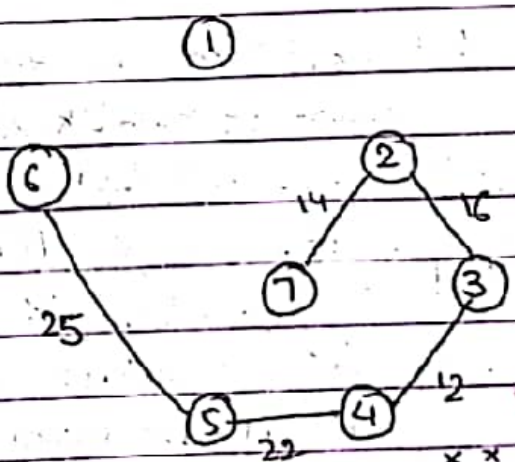
(b)

Min = { 18, 24, 28, 12 }



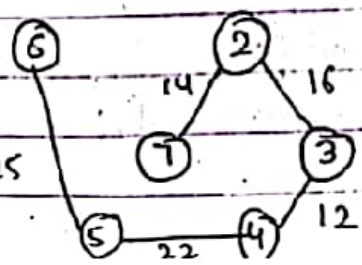
(c)

Min = { 18, 24, 28, 22 }

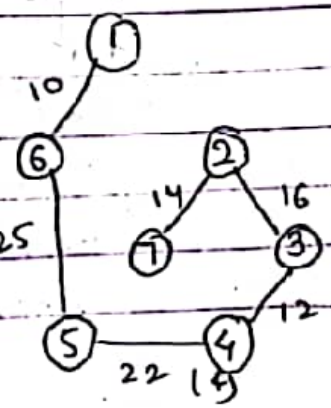


(d) Min = { 18, 24, 28, 25 }

(e)



(f) Min = { 18, 24, 28, 10 }



Total Cost = 10 + 25 + 22 + 12 + 16 + 14

= 99 //

~~3.4.2 Kruskal's Algorithm~~:

Algorithm for prim's method:

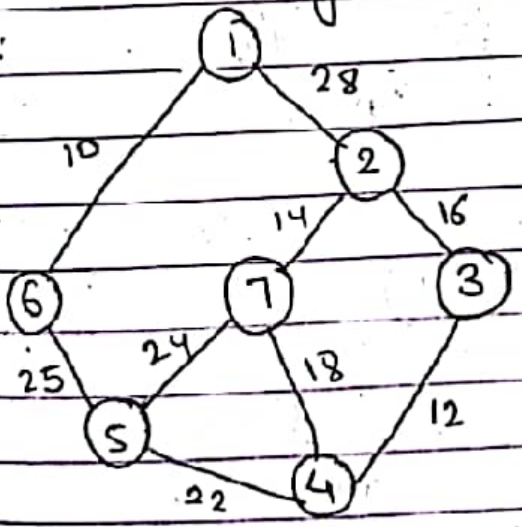
```
Algorithm Prim (E, Cost, n, t)
// E is the set of edges in G. Cost[1:n, 1:n] is the cost
// adjacency matrix of an n vertex graph such that cost[i, j] is
// either positive real no. or ∞ if no edge (i, j) exists.
// A minimum spanning tree is computed and stored as a set
// of edges in the array t[1:n-1, 1:2]. (t[i,1], t[i,2]) is an
// edge in the minimum-cost spanning tree. The final cost is returned
{
  Let (k, l) be an edge of minimum cost in E;
  mincost := cost[k, l];
  t[1,1] := k; t[1,2] := l;
  for i := 1 to n do // initialize near.
    if (cost[i, l] < cost[i, k]) then near[i] := l;
    else near[i] := k;
  near[k] := near[l] := 0;
  for i := 2 to n-1 do
  { // find n-2 additional edges for t.
    Let j be an index such that near[j] ≠ 0 and
    cost[j, near[j]] is minimum;
    t[i, 1] := j; t[i, 2] := near[j];
    mincost := mincost + cost[j, near[j]];
    near[j] := 0;
```

```
for k:=1 to n do
    if ((near[k] ≠0) and cost[k, near[k]] > cost[k, j]))
        then near k:= j;
}
return mincost;
}
```

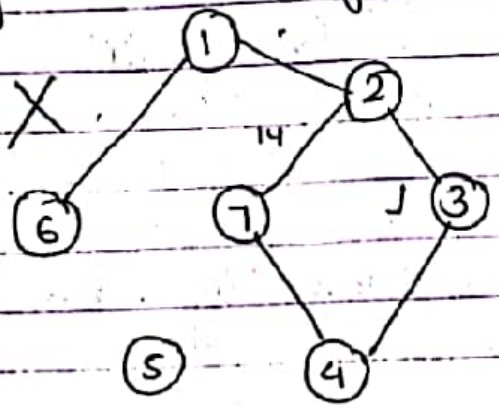
3.4.2 Kruskal's Algorithm:

This interpretation is that the set of t edges selected for the spanning tree be such that it is possible to complete t into a tree. Thus, t may not be a tree at all stages in the algorithm. It will only be a forest since the set of edges t can be completed into a tree iff there are no cycles in t.

Eg:



Soln: We begin with no edges selected.



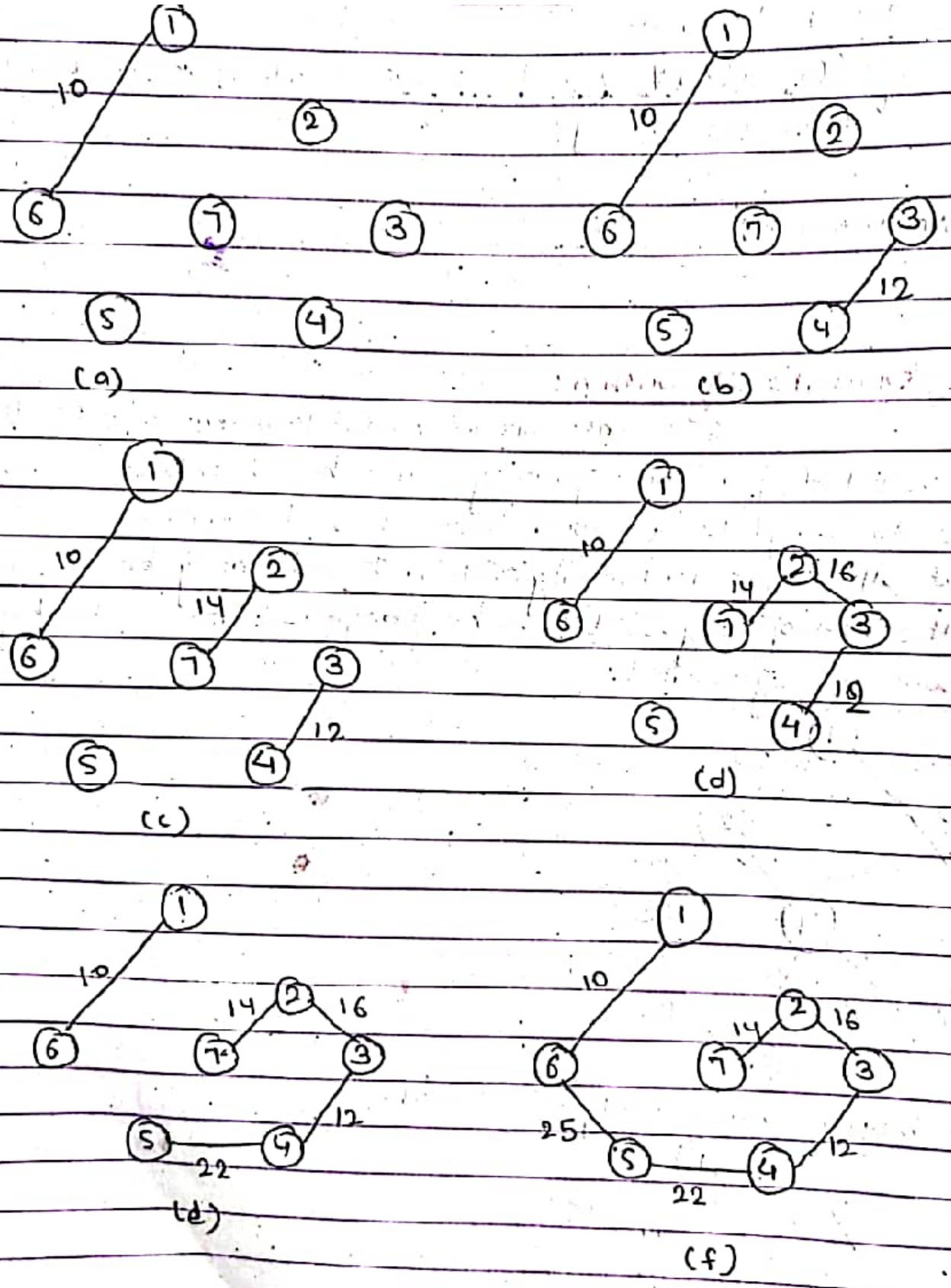


fig: stages of kruskal's Algorithm

Algorithm for kruskal Method

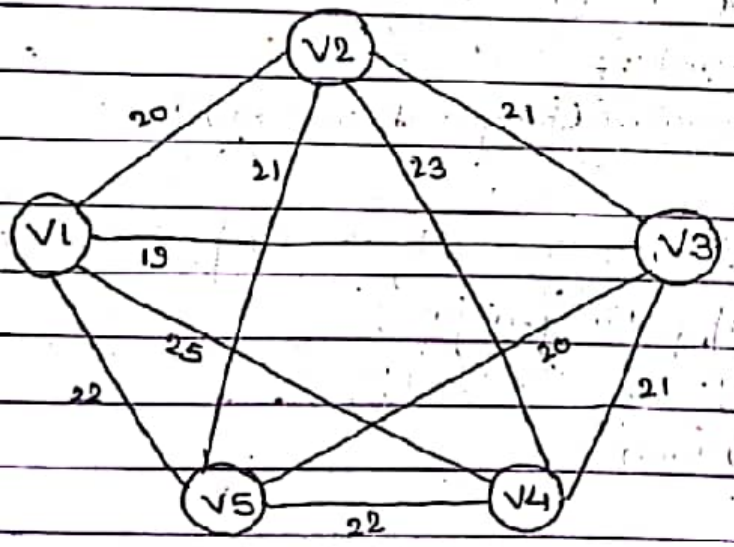
```
Algorithm kruskal (E, Cost, n, t)
// E is the set of edges in G. G has n vertices. Cost[u,v] is t
// Cost of edge(u,v). t is the set of edges in the minimum-
// cost spanning tree. The final cost is returned.
{
  Construct a heap out of the edge costs using Heapify;
  for i:= 1 to n do parent[i]:= -1;
  // Each vertex is in a different set.
   i:= 0; mincost:= 0.0;
  while ((i<n-1) and (heap not empty)) do.
  {
   Delete a minimum cost edge.(u,v) from the heap
   and reheapify using Adjust;
   j:= find(u); k:= find(v);
   if (j≠k) then
   {
    i:= i+1;
   t[i,1]:= u; t[i,2]:=v;
   mincost:= mincost + Cost[u,v];
    Union(j,k);
   }
   if (i≠n-1) then write ("No Spanning tree");
   else return mincost;
  }
```

Travelling Salesman Problem:

(loop is allowed)

Example:

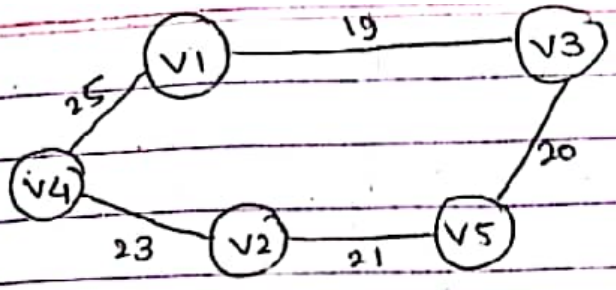
Given a graph $G = \{V, E, w\}$ shown below, where vertices refer to cities, edges refers to connection between the cities, and weight is associated with each edge which represents the cost. Using the Greedy strategy find out the solution for this travelling salesman problem.



soln:

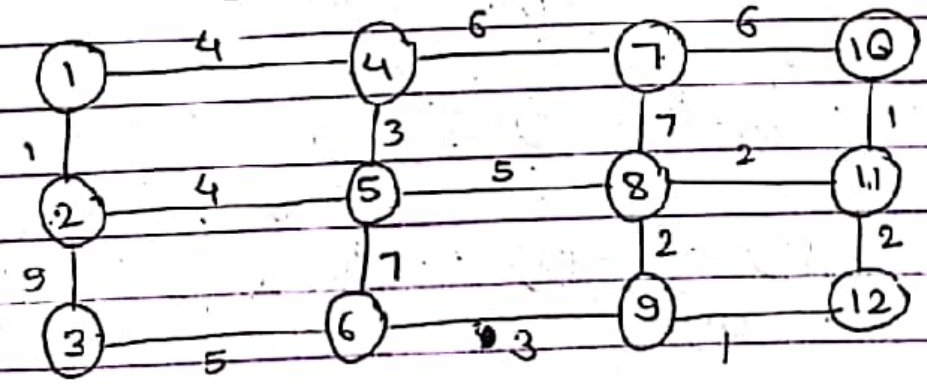
	V1	V2	V3	V4	V5
V1	0	20	(19)	25	22
V2	20	0	21	(23)	21
V3	19	21	0	21	(20)
V4	(25)	23	21	0	22
V5	22	(21)	20	22	0

$V1 \xrightarrow{19} V3 \xrightarrow{20} V5 \xrightarrow{21} V2 \xrightarrow{23} V4 \xrightarrow{25} V1$



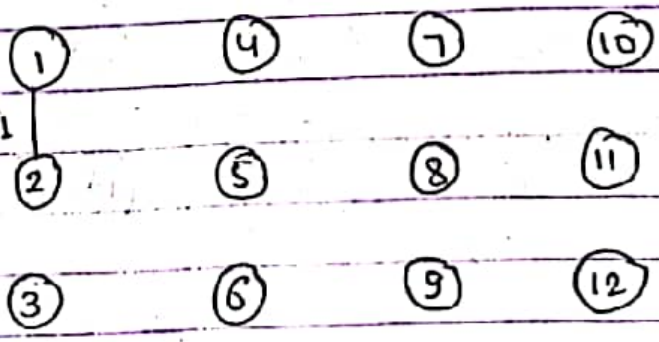
Total Cost = 19 + 20 + 21 + 23 + 25 = 108

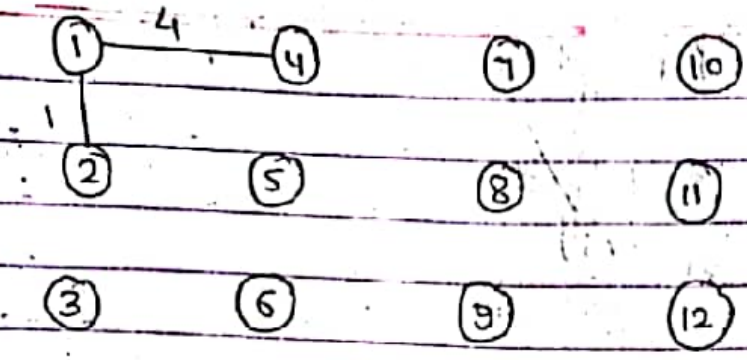
Q. There is a network given the figure below as a highway map and the number recorded next to each arc as the maximum elevation encountered in traversing the arc. A traveller plans to drive from node 1 to 12 on this highway. The traveller dislikes high altitudes and so would find a path connecting node 1 to node 12 that minimizes the maximum altitude. Find the best path for the traveller using a minimum spanning tree.



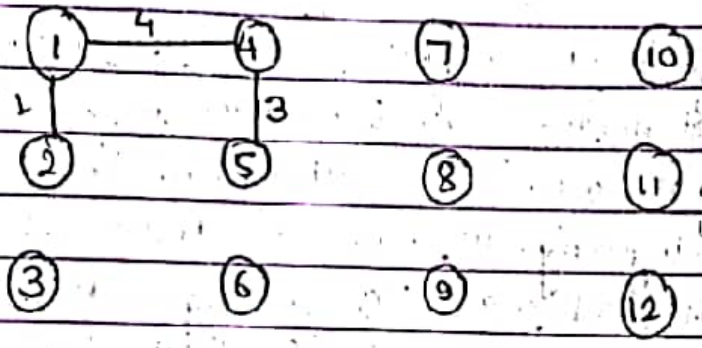
Soln:

Prim's algorithm

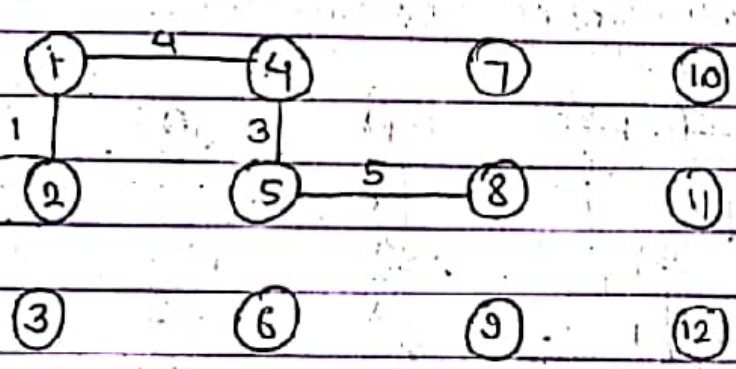



1
4
7
10
1
2
5
8
11
3
6
9
12

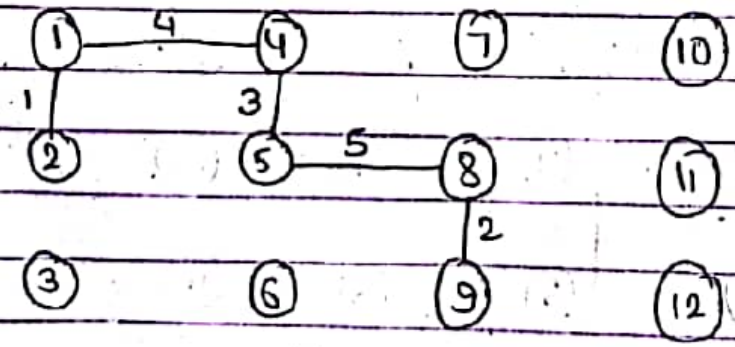
(b)


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

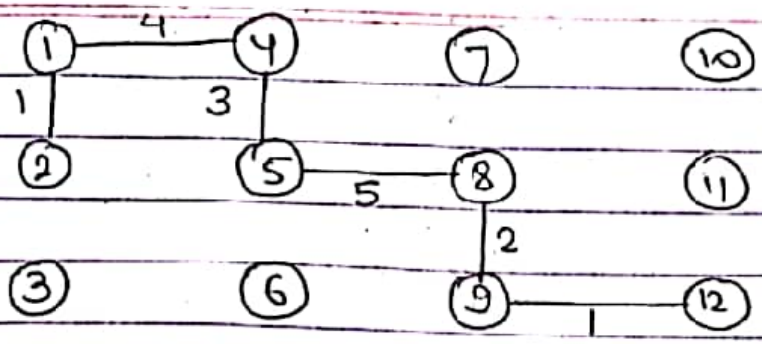
(c)


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

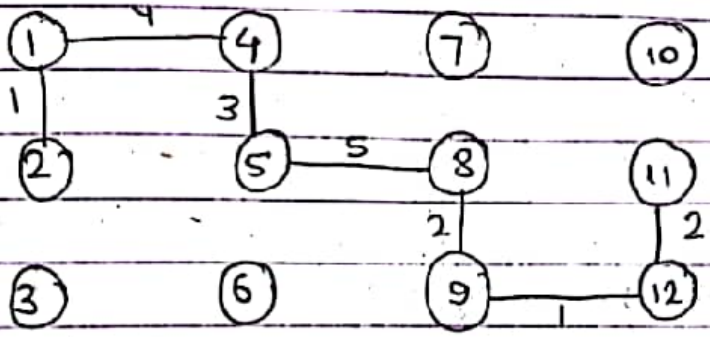
(d)


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

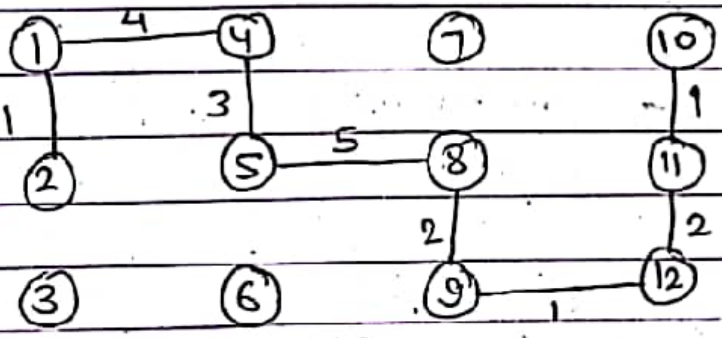
(e)


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

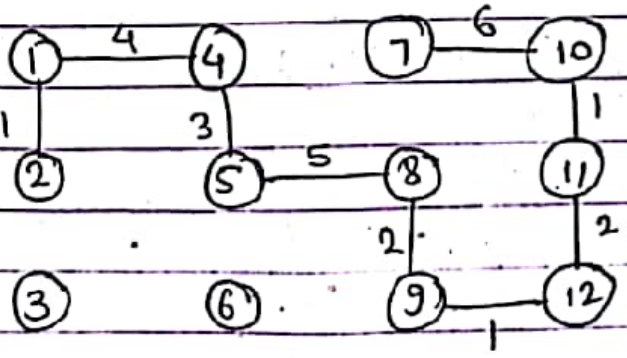
(f)


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

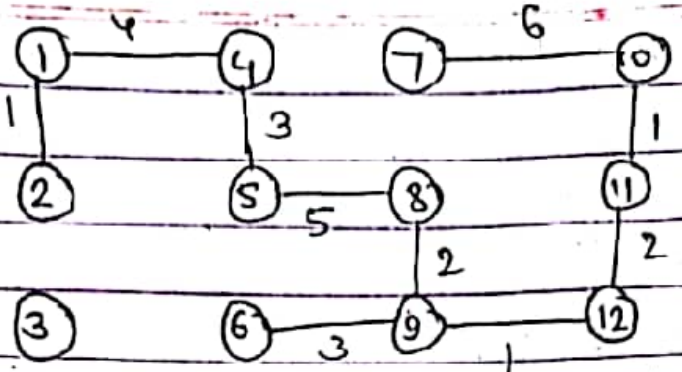
(g)


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

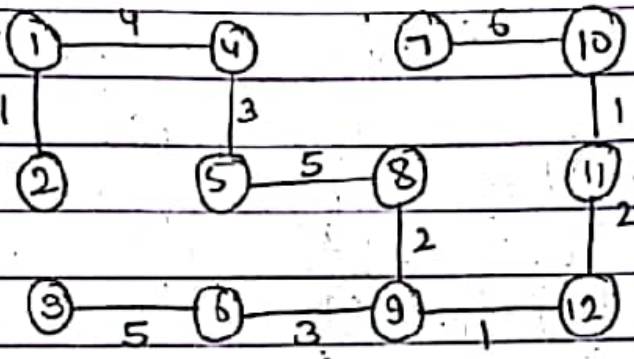
(h)


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

(i)



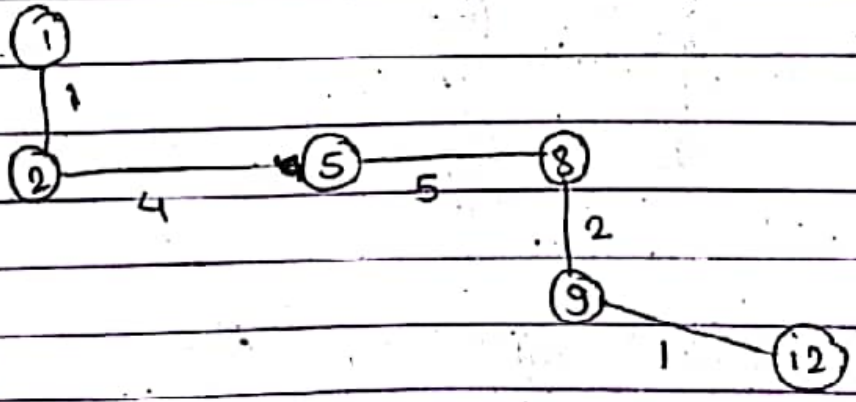
(j)



Cost = 1 + 4 + 3 + 5 + 2 + 1 + 2 + 1 + 6 + 3 + 5

= 33

From (1) to (12) he ~~be~~ can travel in



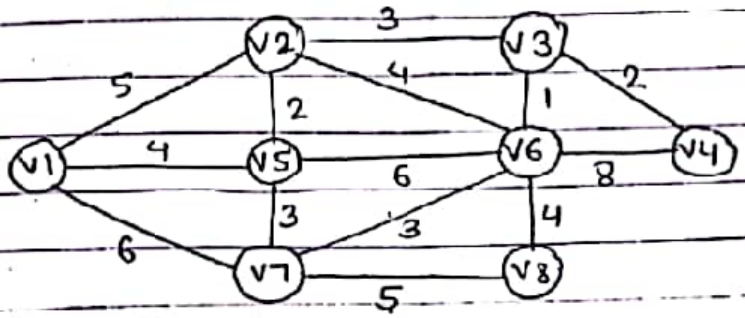
Cost = 1 + 4 + 5 + 2 + 1

= 13

2) For the graph shown below obtain the following:

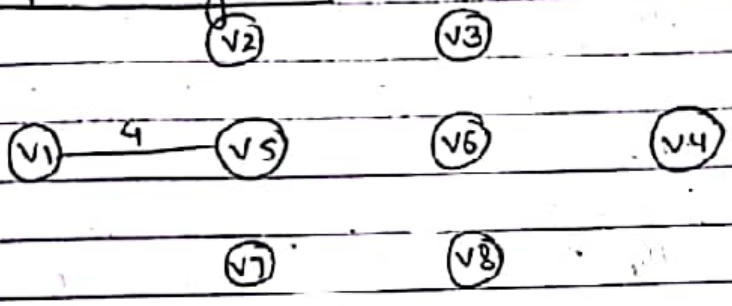
i) MST by Kruskal's algorithm

ii) MST by prim's algorithm

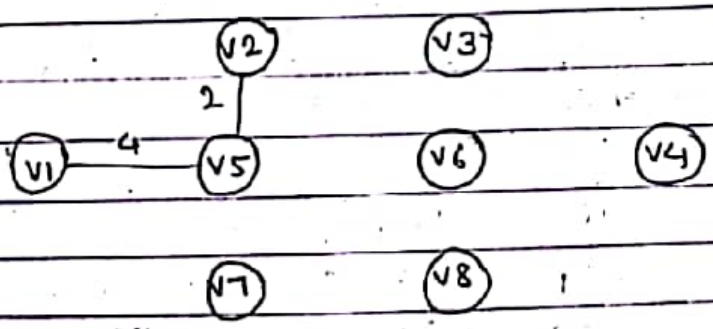


Soln:

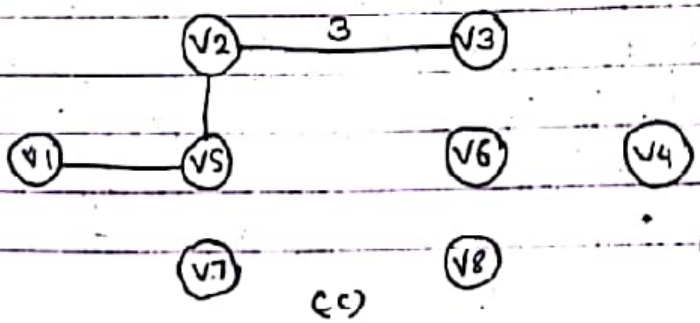
By prim's algorithm



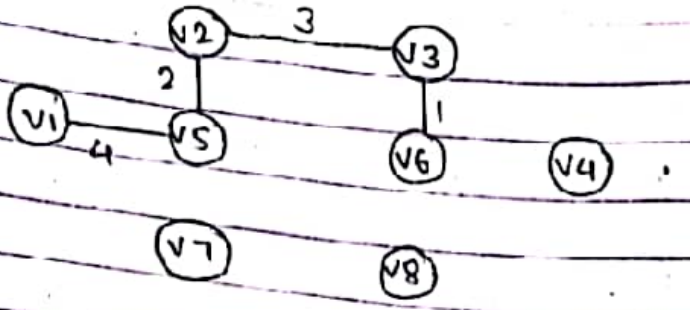
(a)

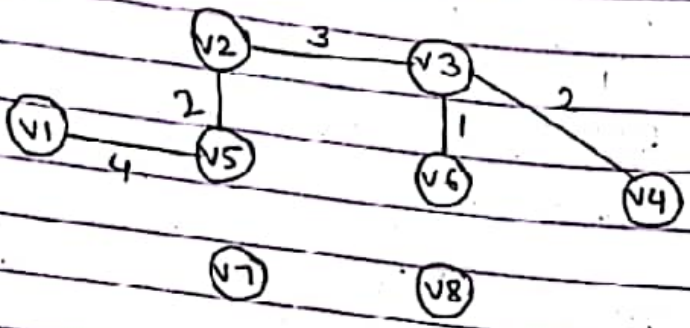


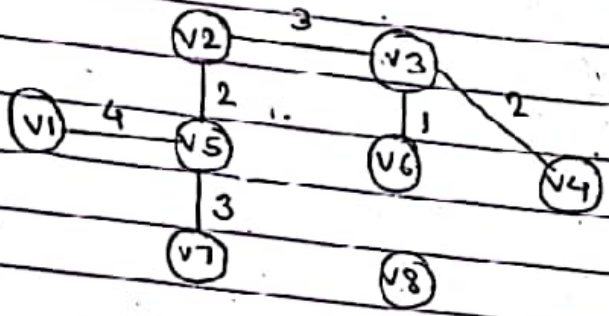
(b)

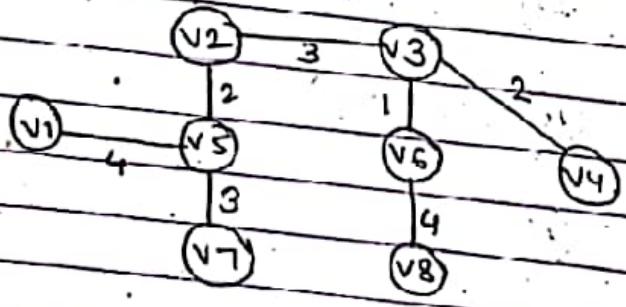


(c)





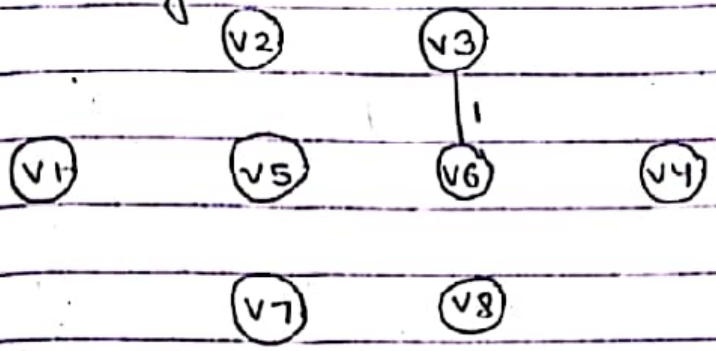




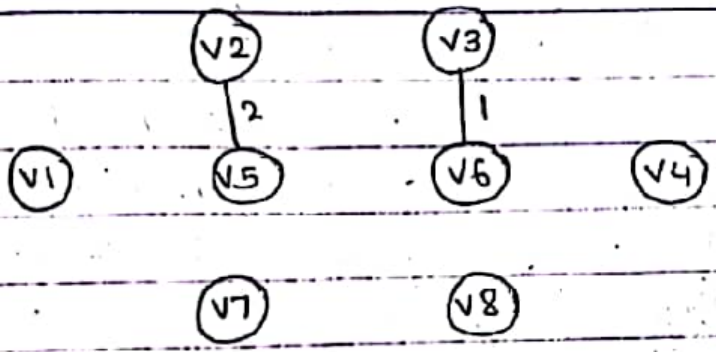
Min - Cost = 4+3+2+3+1+4+2

= 19

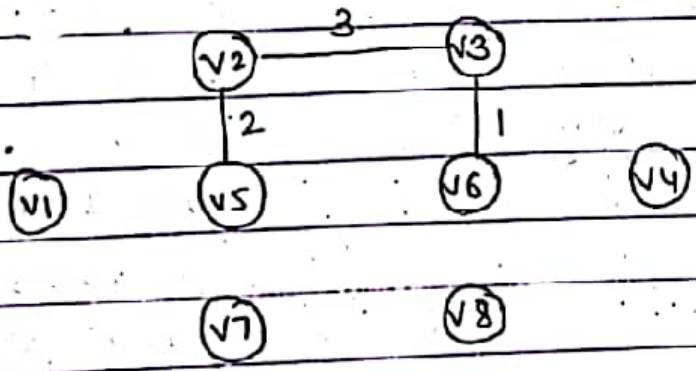
By kruskal's algorathim:



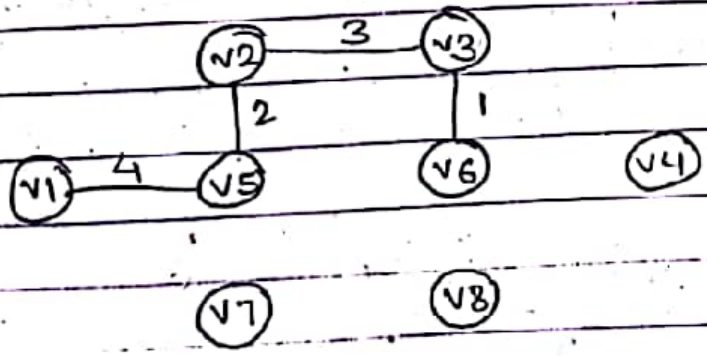
(a)

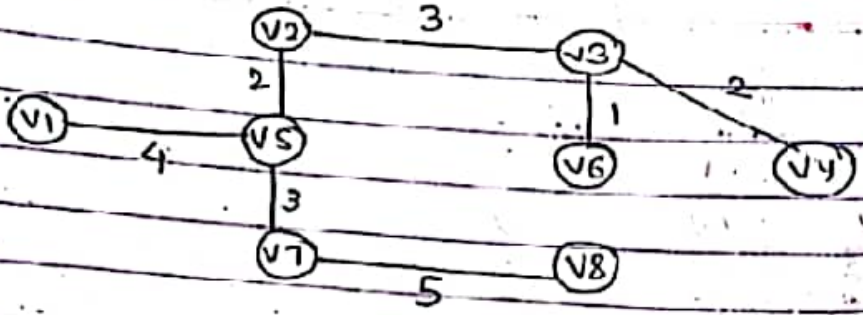


(b)



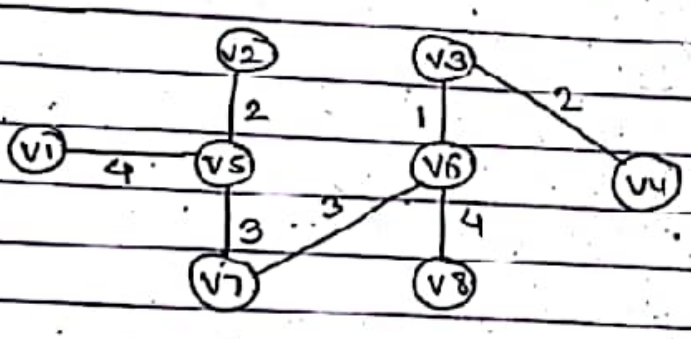
(c)





Cost = 4+2+3+5+3+1+2
= 20

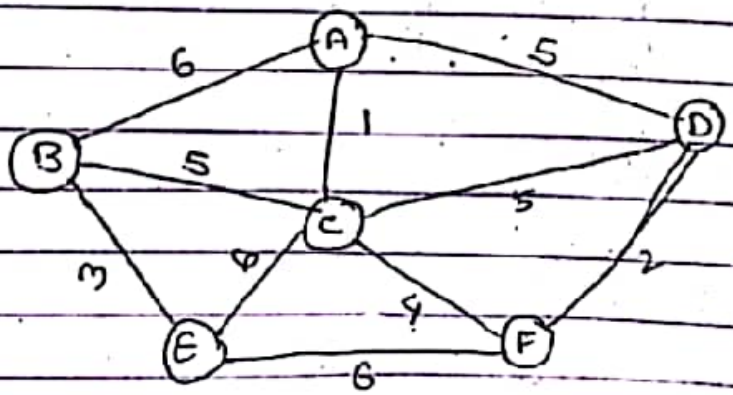
By OR



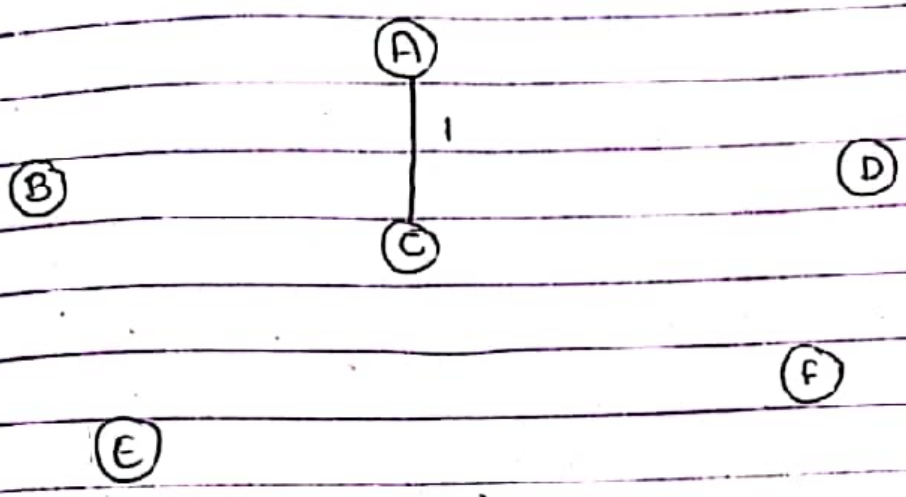
V3-V6 = 1
V2-V5 = 2
V3-V4 = 2
V5-V7 = 3
V6-V7 = 3
V2-V6 = 4
V6-V8 = 4
V1-V5 = 4
V7-V8 = 5
V1-V2 = 5
V1-V7 = 6
V5-V6 = 6

Cost = 4+2+3+3+4+1+2
= 19

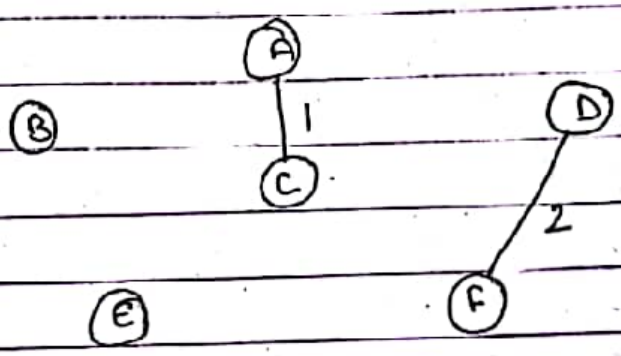
Q: Find minimum cost - spanning tree.



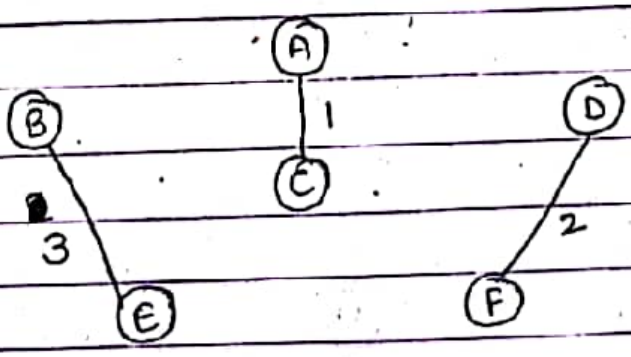
Soln: using Kruskal's algorithm



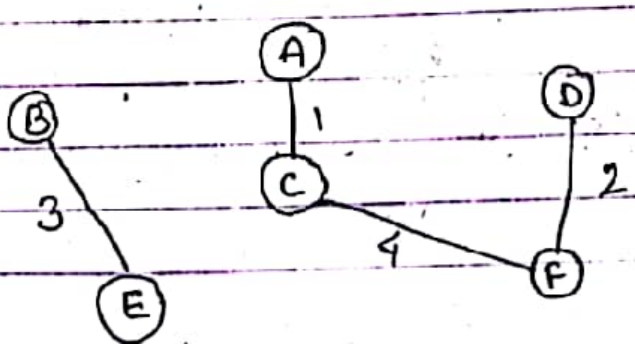
(a)

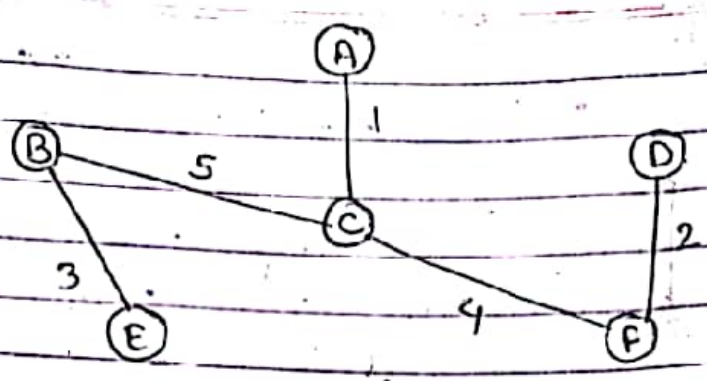


(b)



(c)

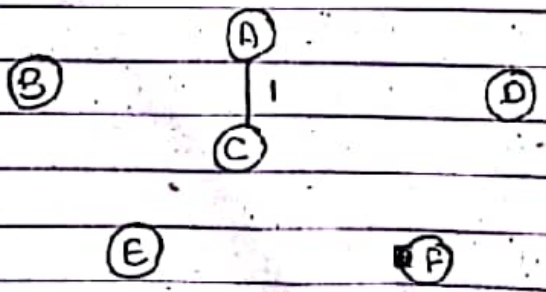




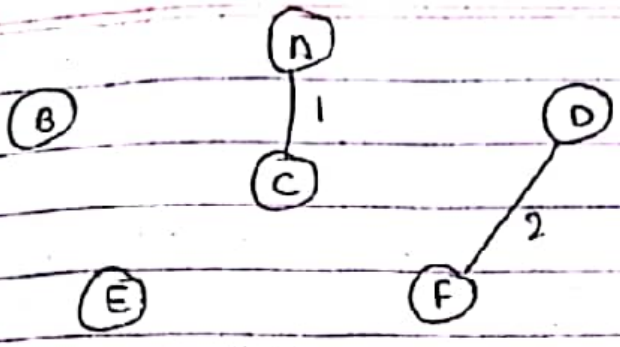
(d)

Using kruskal's algorithm

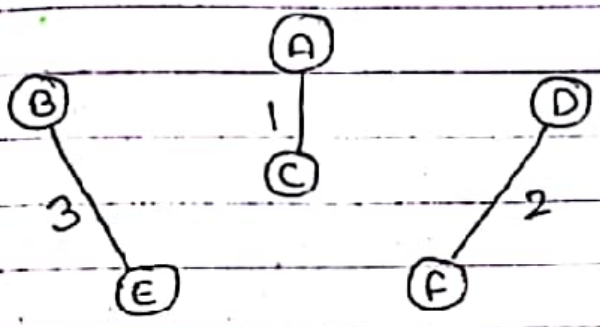
Edges	Cost
AC	1
DF	2
BE	3
CF	4
CD	5
AD	5
BC	5
AB	6
CE	6
EF	6



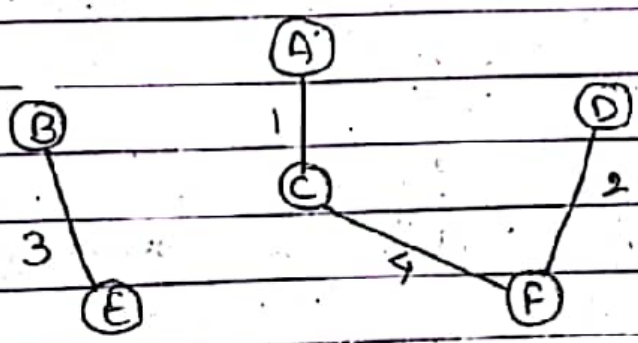
(a)



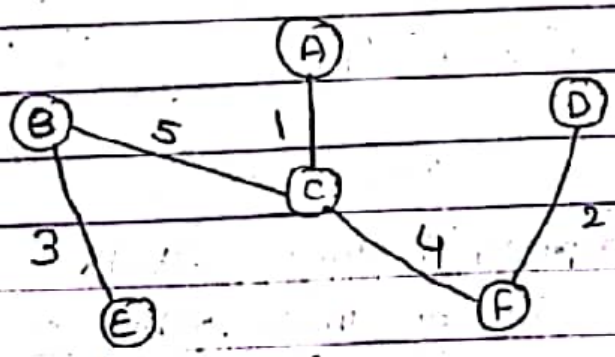
(b)



(c)



(d)



(e)

Cost = 3 + 5 + 1 + 4 + 2

= 15

Chapter-2
Divide and Conquer

2.1 General Method:

Given a function to compute on 'n' inputs, the divide and conquer strategy suggests splitting the inputs into 'k' distinct subsets, $1 < k \leq n$, yielding k sub problems. These sub-problems must be solved, and then a method must be found to combine sub-solutions, into a solution of the whole. If the sub-problems are still large, the the divide and conquer strategy can be re-applied. often the sub-problems resulting from a divide and conquer design are of the same type as the original problem. for those cases the re-application of the divide and conquer principal is naturally expressed by a recursive algorithm. Now, smaller and smaller sub-problems of the same kind are generated until eventually subproblems that are small enough to be solved without splitting are produced.

The following algorithm gives us an overall design of divide and conquer strategy:

```
Algorithm D And C(p) // P is the problem to be solved
{
  if small(p) then return 'S(p)';
  else
  {
  divide p into smaller instances P1, P2, ---. Pk, k >= 1;
  Apply D And C to each of these subproblems;
  return combine (D And C(P1), D And C(P2), ---, D And C(Pk));
  }
}
```

Small(p) is a Boolean-valued function that determines whether the input size is small enough that the answer can be computed without splitting. If this is so, the fn S is invoked. Otherwise the problem p is divided into smaller subproblems. These subproblems $P_1, P_2, \ldots P_k$ are solved by recursive applications of D And C. Combine is a function that determines the solution to P using the solutions to the k subproblems.

* If the size of p is n and the size of the k sub problems are $n_1, n_2, \ldots, A_k$ respectively, then the computing time of D and C is described by the recurrence relation.

$$T(n) = \begin{cases} g(n) \\ T(n_1) + T(n_2) + \cdots + T(n_k) + f(n) \text{ otherwise} \end{cases}$$

Where g(n) is the time to Compute the answer directly for Small inputs. The function f(n) is the time dividing p and Combining the Solutions to Subproblems.

The Complexity of many divide-and-Conquer algorithms is given by recurrences of the form

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

Where, a = number of subproblems
b = size of each subproblems
a, b and f(n) are Constants

Example: Consider the case in which $a=2$ and $b=2$; let $T(1)=2$ and $f(n)=n$.

Soln:

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

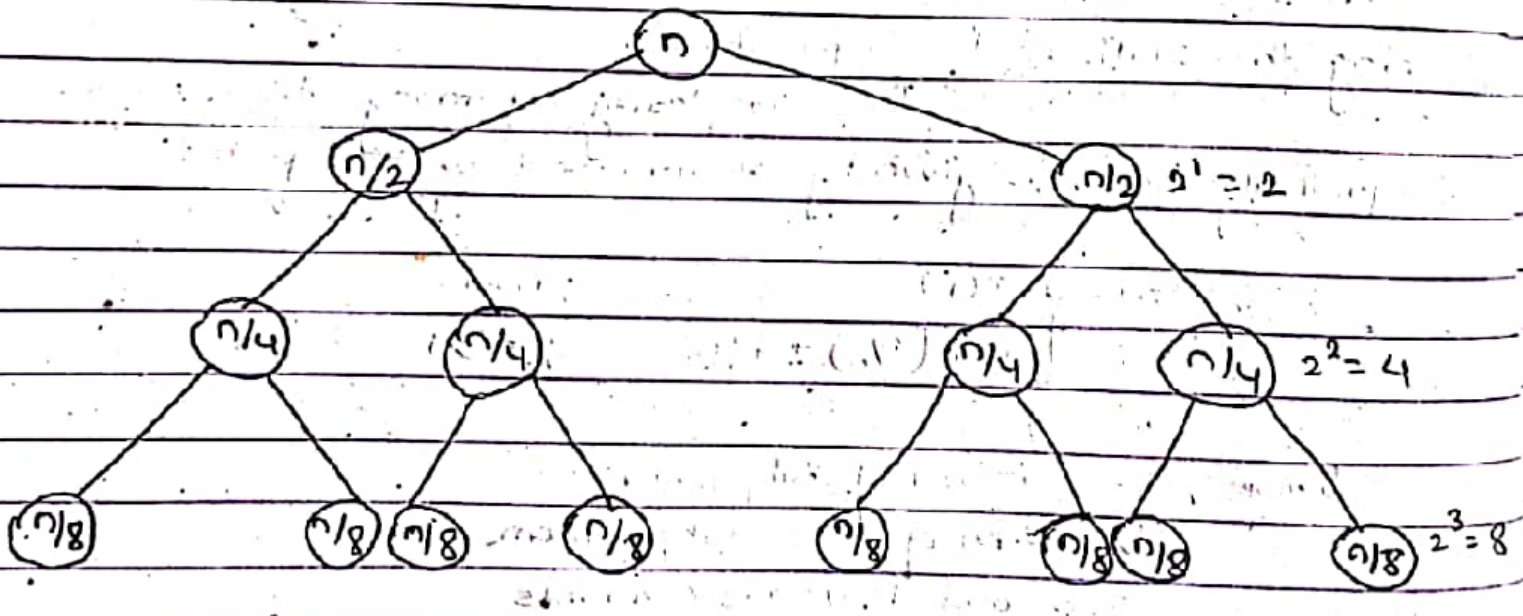
$$= 2\left[2T\left(\frac{n}{4}\right) + \frac{n}{2}\right] + n$$

$$= 4T(n/4) + 2n$$

$$= 4\left[2T(n/8) + n/4\right] + 2n$$

$$= 8T\left(\frac{n}{8}\right) + 3n$$

$$\therefore T(n) = 2^i T\left(\frac{n}{2^i}\right) + i \cdot n \qquad \left[2^3 T\left(\frac{n}{2^3}\right) + 3n \Rightarrow i = 3\right]$$



height $= \log_2 n + 1$

$n=2, \quad \log_2 2 + 1 = 1 + 1 = 2 = h$

Put $i = \log_2 n$

$$T(n) = 2^{\log_2 n}\, T\left(\frac{n}{2^{\log_2 n}}\right) + n\log_2 n$$

2.2 Binary Search:

Example: let us select the 14 entries

1	2	3	4	5	6	7	8	9	10	11	12	13	14
-15	-6	0	7	9	23	54	82	101	112	125	131	142	151

Case - I

Search for 151

low = 1 ; high = 14

$$mid = \frac{1+14}{2} = 7$$

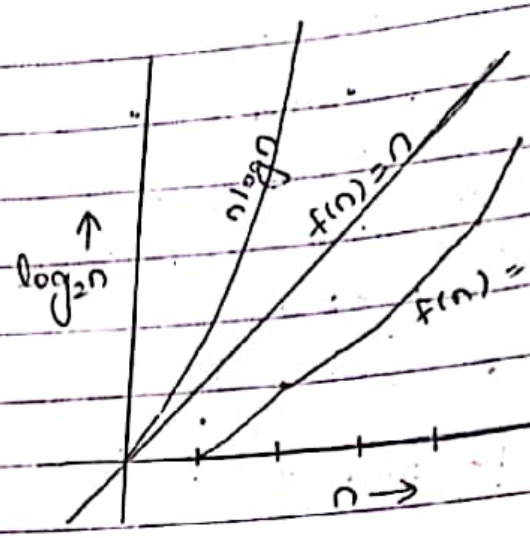
	low	high	mid	
Case - I	1	14	7	
Case - II	7+1	14	11	a > mid
Case - III	11+1	14	13	where a is no. to be search
Case - IV	13+1	14	14	found

If the value of a is more than mid - value then

low = mid + 1

else

high = mid - 1 where $mid = \frac{high + low}{2}$



minimum complexity $= \log_2 n$

Recursive Algorithm for Binary Search:

```
Algorithm Binsrch (a, i, l, x)
// Give an array a[i:l] of elements in nondecreasing
// order, 1 ≤ i, ≤ l, determine whether x is present, and
// if so, return j such that x = a[j]; else return 0.
{
    if (l=i) then // If small(p)
    {
        if (x = a[i]) then return i;
        else return 0;
    }
    else
    {// Reduce p into a smaller subproblem
    mid := [(i+l)/2];
    if (x = a[mid]) then return mid;
    else if (x < a[mid]) then
```

```
        return BinSrch(a, i, mid-1, x);
        else return BinSrch(a, mid+1, l, x);
    }
}
```

Iterative algorithm for binary Search:

```
Algorithm BinSearch(a, n, x)
// Give an array a[1:n] of elements in non-decreasing order
// n >= 0, determine whether x is present and if so, return j
// such that x = a[j]; else return 0.
{
    low := 1; high := n;
    while (low <= high) do
    {
        mid := [(low + high)/2];
        if (x < a[mid]) then high := mid-1;
        else if (x > a[mid]) then low := mid+1;
        else return mid;
    }
    return 0;
}
```

Time Complexity of Binary Search

Best Search, Time Complexity, $T(n) = 1$

Worst search, Time Complexity, $T(n) = \log_2 n$

Average, $T(n) = \frac{1 + \log_2 n}{2} = \log_2 n$

Binary Search:

Let a_i, $1 \le i \le n$, be a list of elements that are sorted in nondecreasing order. Consider the problem of determining whether a given element x is present in the list. If x is present, we are to determine a value j such that $a_j = x$. If x is not in the list, then j is to be set to zero. Let $p = (n, a_i, \ldots, a_l, x)$ denote an arbitrary instance of this search problem (n is the number of elements in the list, $a_i \ldots a_l$ is the list of elements, and x is the element searched for).

Divide-and-Conquer can be used to solve this problem. Let Small(p) be true if $n = 1$. In this case, S(p) will take the value i if $x = a_i$; otherwise it will take the value 0.

Iterative Binsearch has three inputs a, n and x. The while loop continues processing as long as there are more elements left to check. At the conclusion of the procedure 0 is returned if x is not present, or j is returned, such that $a[j] = x$.

2.3 Merge sort:

As another example of divide and conquer, we investigate a sorting algorithm that has the nice property that in the worst case its complexity is $O(n \log n)$. This algorithm is called merge sort. Given a sequence of n elements called keys $a[1], \ldots, a[n]$, the general idea is to imagine them split into two sets $a[1], \ldots, a[n/2]$ and $a[(n/2)+1], \ldots, a[n]$. Each set is individually sorted, and the resulting sorted sequences are merged to produce a single sorted sequence of n elements. Thus we have another ideal example of the divide-and-conquer strategy in which the splitting is into two equal-sized sets and the combining operation is the merging of two sorted sets into one.

Eg: Consider the array of 5 elements a[1:5] = (6, 3, 4, 8, 5).

Soln:

	1	2	3	4	5
List	6	3	4	8	5

Step1: $mid = \frac{high + low}{2} = \frac{5+1}{2} = 3$

high = 5, low = 1

6	3	4	8	5

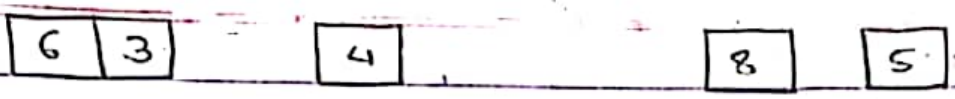
Step 2:

	1	2	3
	6	3	4

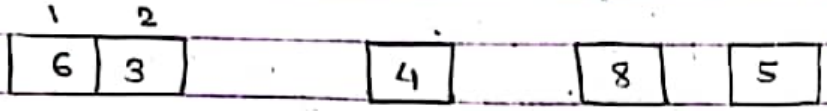
$mid = \frac{1+3}{2} = 2$

	1	2
	8	5

$mid = \frac{1+2}{2} = 1.5 \approx 1$

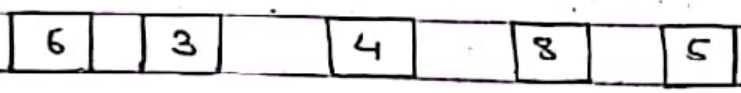


step3 :



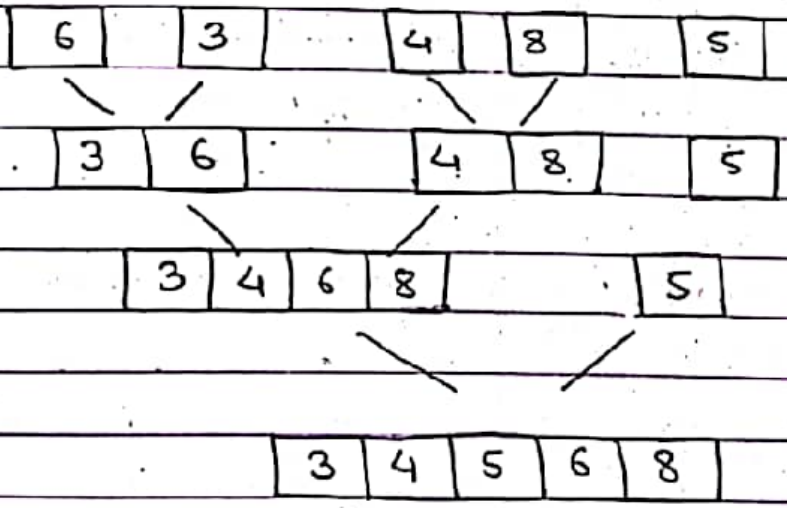
$$mid = \frac{1+2}{2} = 1.5 \approx 1$$

step 4:



(Base-case)

Now Merging in bottom up way :



Algorithm for Merge Sort :

```
Algorithm Mergesort (low, high)
// a[low:high] is a global array to be sorted.
// Small(p) is true if there is only one element to sort. In
// this case the list is already sorted.
{
    if (low < high) then // If there are more than one elements
}
```

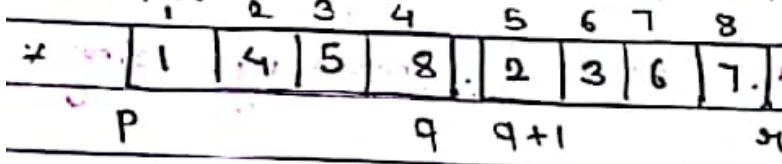
```
// Divide P into 1. Subproblems
    // find where to split the set.
        mid := [(high + low)/2];
// Solve the subproblems
    MergeSort (low, mid);
    MergeSort (mid+1, high);
// Combine the solutions
    Merge (low, mid, high);
  }
}
```

Algorithm for ~~sorting~~ Merging two sorted subarrays using auxiliary storage :-

```
Merge (A, P, q, r)
{
  n1 := q-p+1;
  n2 := r-q;
// create arrays [1..n1+1] and R[1..n2+1]
  for i := 1 to n1 do
    L[i] := A[p+i-1];
  for j := 1 to n2 do
    R[j] = A[q+j];
    L[n1+1] := ∞;
    R[n2+1] := ∞;
    i := 1;
    j := 1;
  for k := p to r do
    if L[i] <= R[j]
```

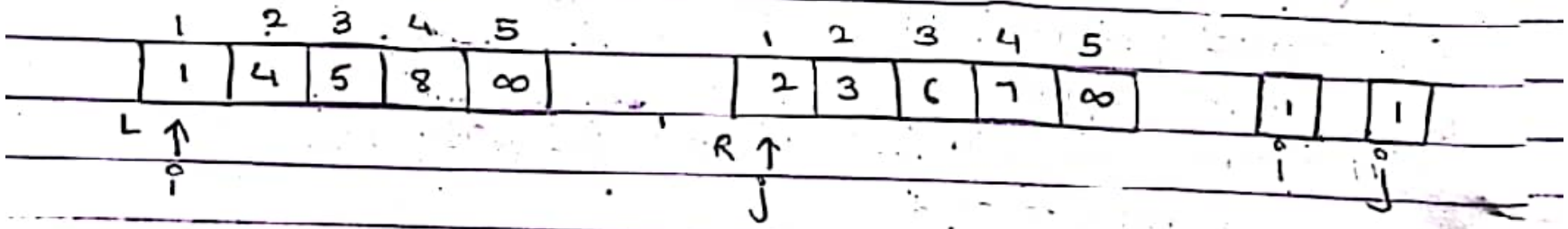
```
{
    A[k] := L[i];
    i := i+1;
}
else
{
    A[k] := R[j];
    j := j+1;
}
}
```

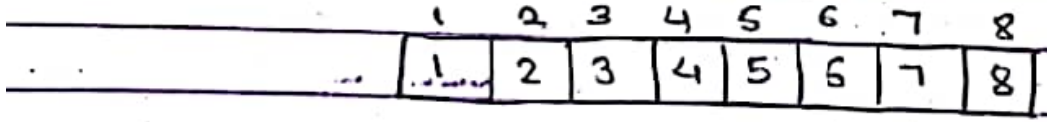
Example:



$n_1 = 4 - 1 + 1 = 4$

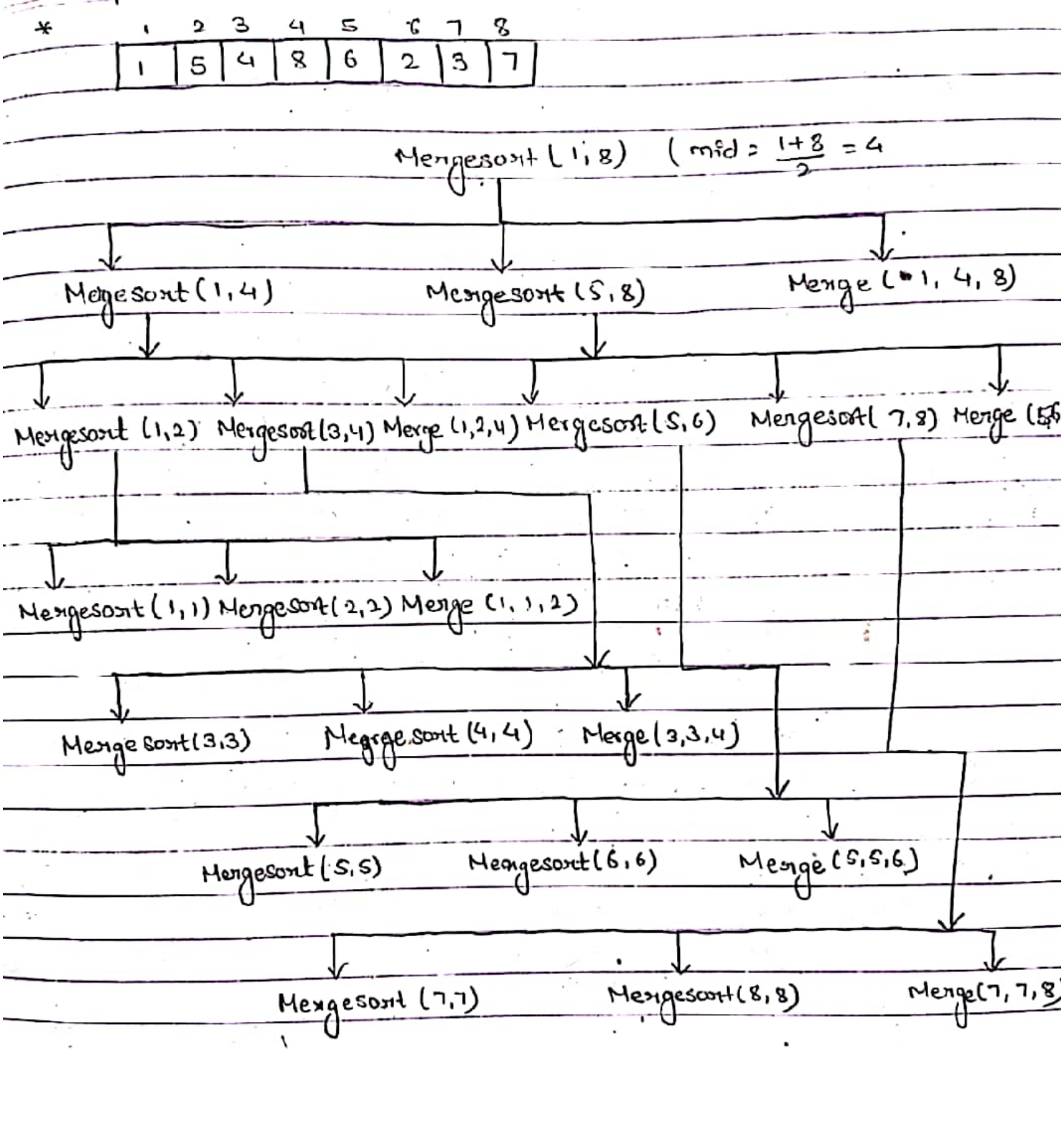
$n_2 = 8 - 4 = 4$

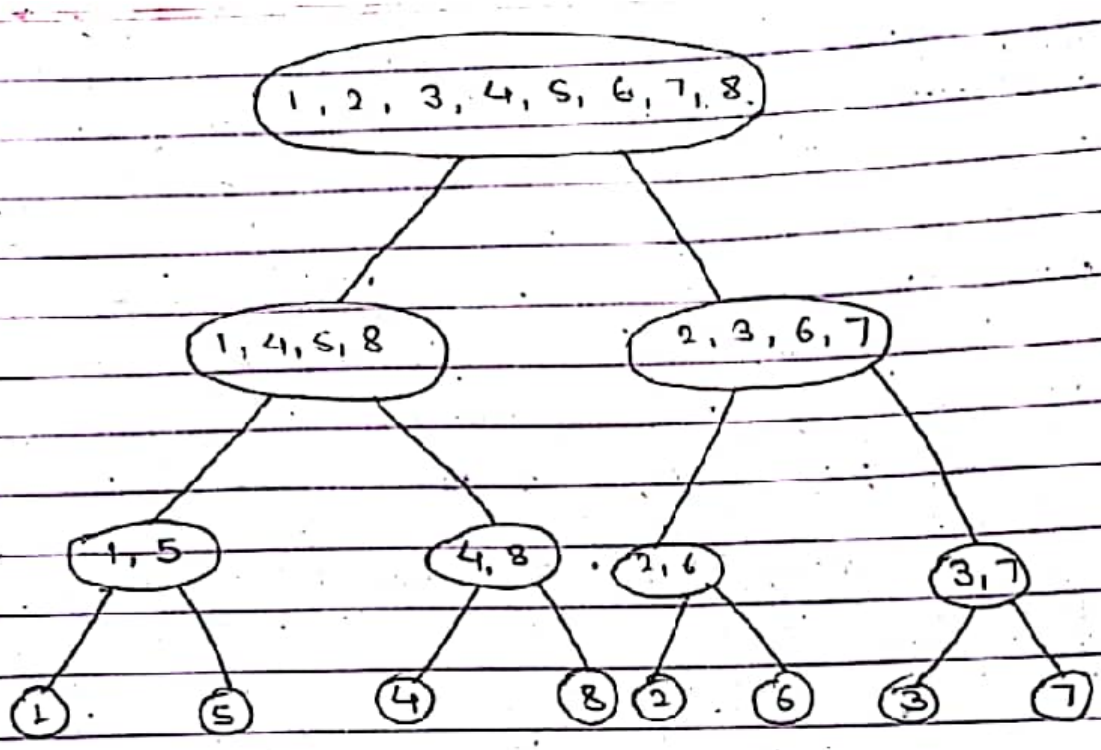




*

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





Time Complexity = $O(n) + O(n)$

$= 2O(n)$

$= O(n)$

2 is constant.

Complexity = order of n i.e. $O(n)$

for Mergesort

$T(n) = 2 * T(n/2) + O(n)$

$= O(n \log_2 n) + O(n)$

$= O(n \log_2 n)$

Mergesort (low, mid) = $T(n/2)$

Mergesort (mid+1, high) = $T(n/2)$

Merge (low, mid, high) = $O(n)$

Calculating time Complexity of Recursive algorithm:

Method 1 : Substitution method / Back Substitution method / Iterative method.

Method 2 : Recursive tree method

Method 3 : Master method

Recursive algorithm :

```
f(n)                                    T(n)
{                                       0
 if( ) then                             1  comparison
 return (f(n/2) + f(n/2));              1  for return + 2*T(n/2)
}                                       0
```

$T(n) = 1 + 1 + 2 * T(n/2)$

$= 2 + 2 * T(n/2)$

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

Back Substitution method / Iteration method :

```
f(n)                                    T(n)
{                                       0
 if (n>1) then                          1
 return ( f(n-1));                      1 + T(n-1)
}                                       0
```

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

$T(n) = T(n-1) + c$ ——(1)

$= T(n-2) + 2c$

$= T(n-3) + c + 2c$

$$T(n) = T(n-3) + 3c$$

$$= T(n-k) + kc$$

$$= T(1)$$

Now, $T(1) = 1$

$T(n) = T(n-1) + c$ if $n > 1$

i.e. $n - k = 1$ to terminate.

$k = n - 1$

$$T(n) = T(1) + kc$$

$$= 1 + nc - c$$

$$= n + (1 - c) \quad [n = nc]$$

$\therefore T(n) = \theta(n) = O(n)$

8. Find out time Complexity for the given function

$T(n) = n + T(n-1)$; $n > 1$

$= 1$; $n = 1$

Soln:

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

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

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

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

$$= n + (n-1) + (n-2) + (n-3) + T(n-4)$$

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

to terminate, $(n-(k+1)) = 1$

$n-k-1 = 1$

$n-k = 2$

$k = n-2$

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

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

$= n + (n-1) + (n-2) + (n-3) + \cdots + 2 + 1$

$= A.S.$ [Arithmetic series]

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

$= \frac{n^2+n}{2}$

$\therefore \boxed{T(n) = \theta(n^2) = O(n^2)}$

ii) Recursion tree method

$T(n) = 2T(n/2) + c \quad ; n > 1$

$T(1) = 1 \quad ; n = 1.$

$\times T(n) = 2T(n/2) + c$

$= 2[2T(n/4) + c] + c$

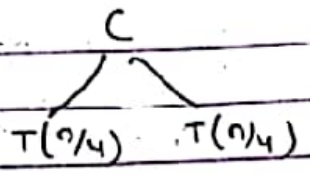
$= 4T(n/4) + 3c$

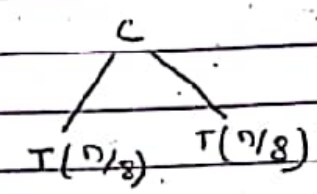
$= 4[2T(n/8) + c] + 3c$

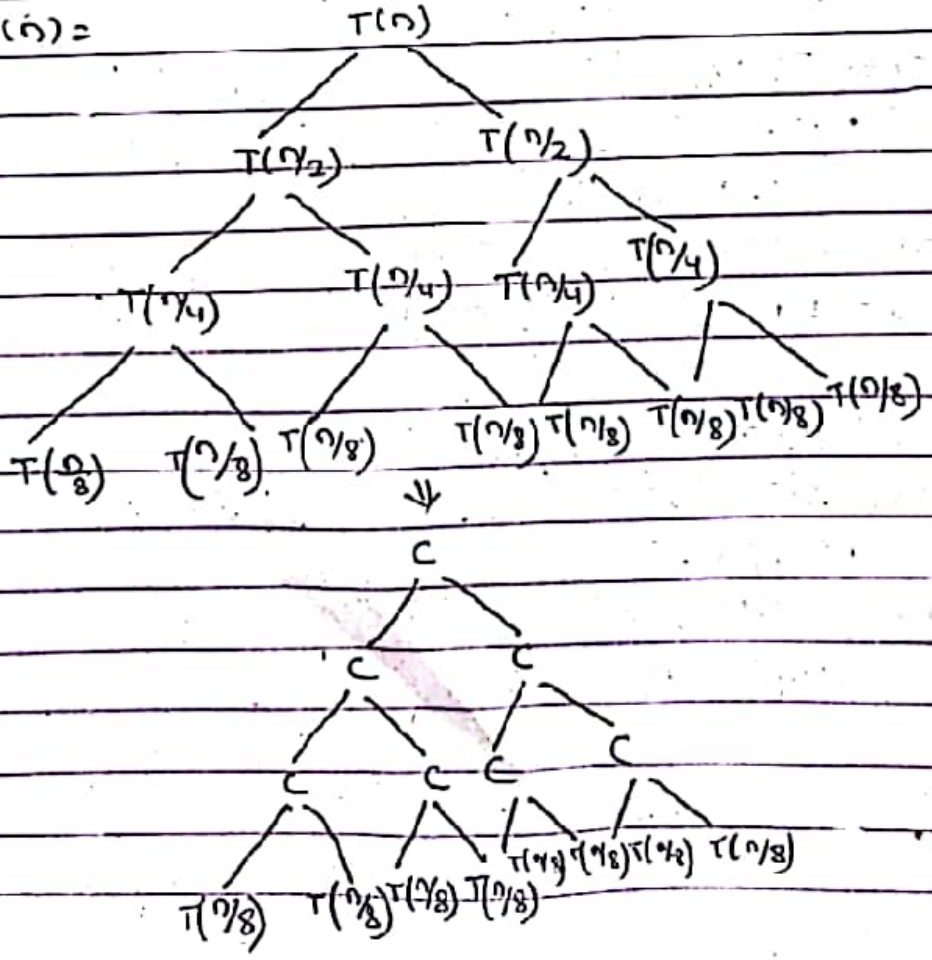
$= 8T(n/8) + 7c$

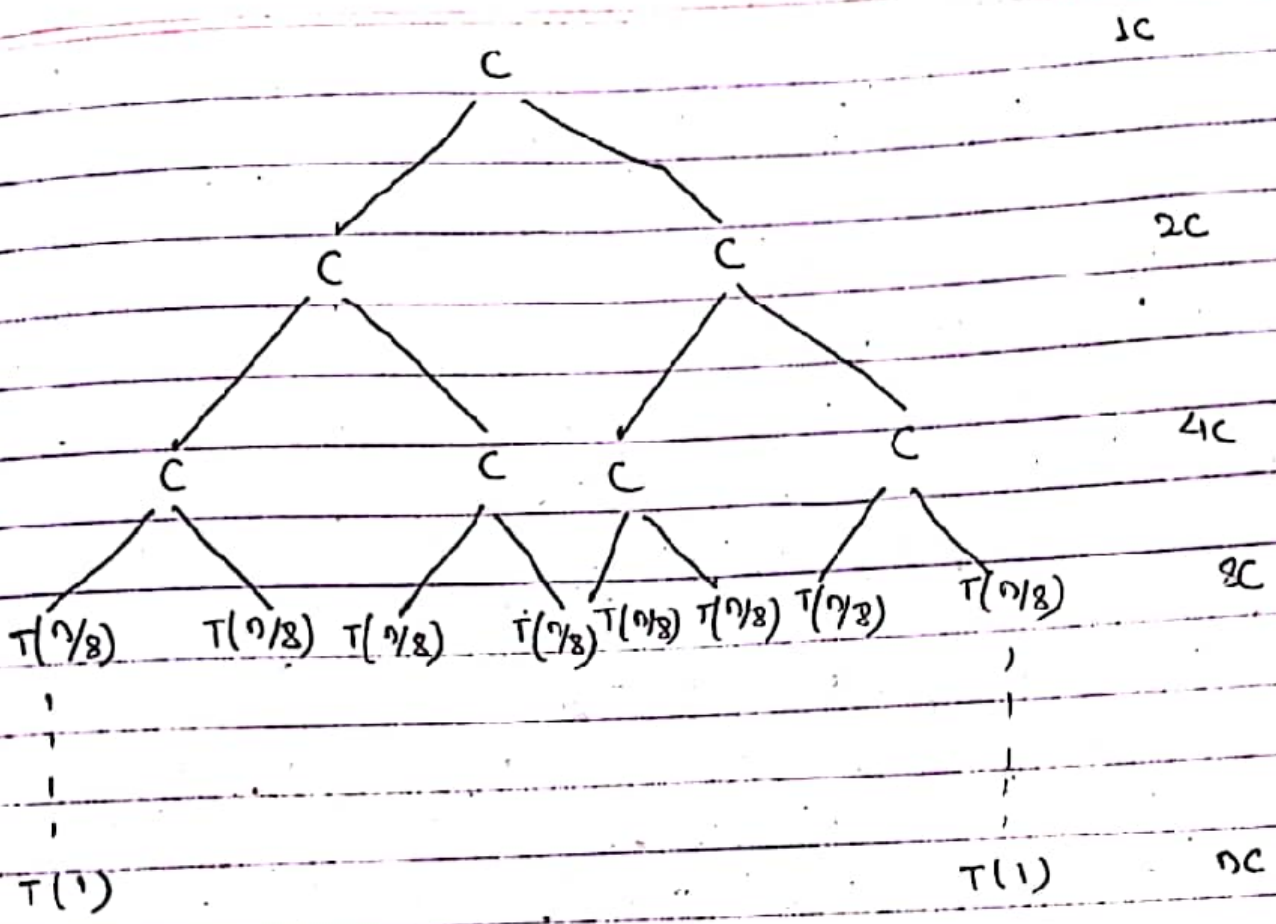
$T(n/2) = 2T(n/4) + c$

$T(n/4) = 2T(n/8) + c$

$T(n/2) =$ 

$T(n/4) =$ 

$T(n) =$ 



Geometric series

$T(n) = C + 2C + 4C + 8C + \cdots + nC$

$= C(1 + 2 + 4 + 8 + \cdots + n)$

$= C(1 + 2 + 4 + 8 + \cdots + 2^k)$ let $n = 2^k$

$= C(2^0 + 2^1 + 2^2 + 2^3 + \cdots + 2^k)$

$n = k + 1$

$= C \cdot \frac{(2^n - 1)}{2 - 1}$ $S = \frac{a(r^n - 1)}{r - 1}$

$= \frac{C \cdot (2^{k+1} - 1)}{1}$ $r = 2, a = 1$

$= C(2^k \cdot 2^1 - 1)$

$= C(n \cdot 2 - 1)$

$= C(2n - 1)$

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

· Merge Sort

Q. $T(n) = 2T(n/2) + n \quad ; n > 1$

$T(n) = 1 \quad ; n = 1$

Soln:

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

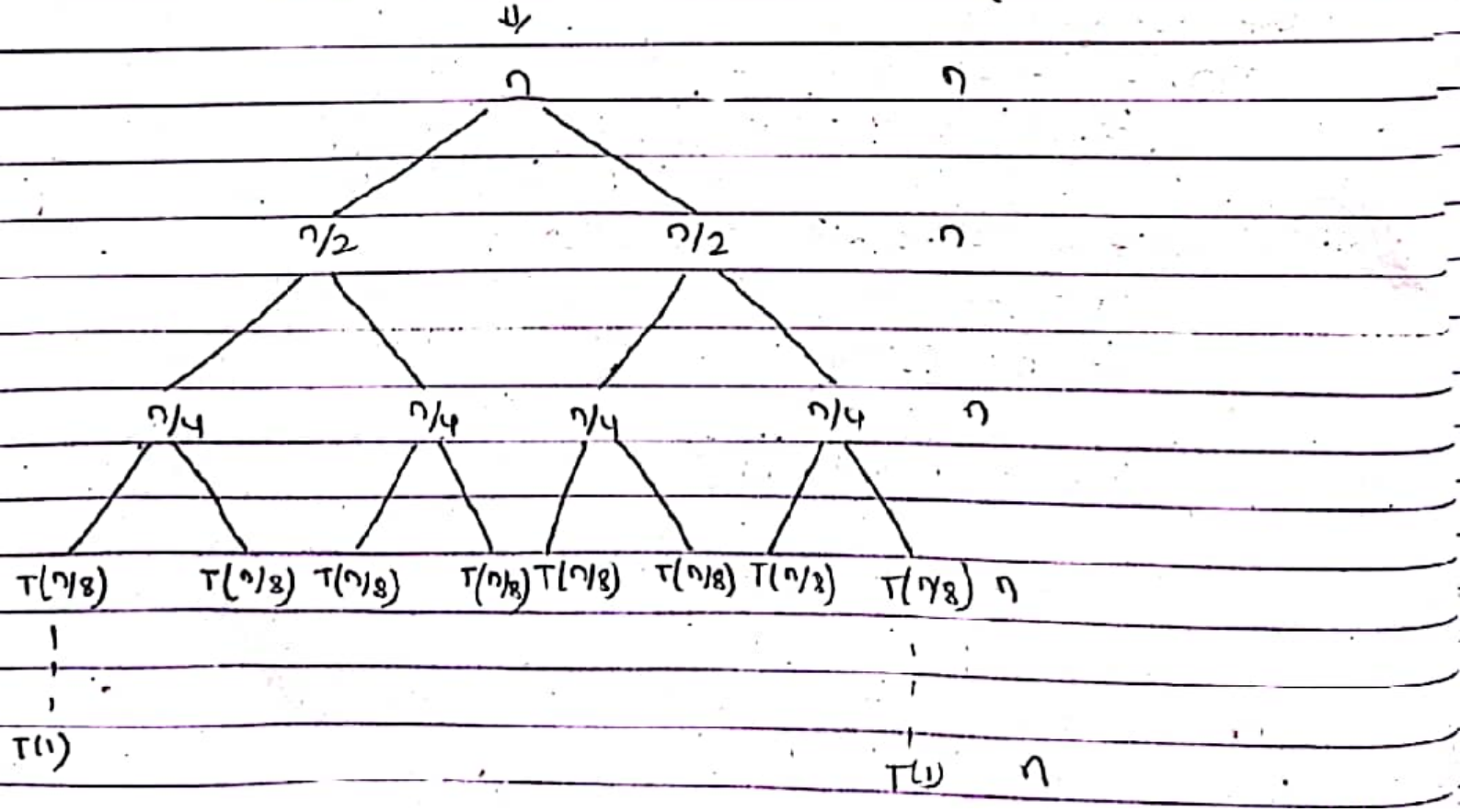
$= 2\left(2T(n/4) + (n/2)\right) + n$

$= 4T(n/4) + n + n$

$= 4\left(2T(n/8) + n/4\right) + 3n$

$= 8T(n/8) + 3n$

$= 2^i T(n/2^i)$



$$T(n) = \frac{n}{2^0} + \frac{n}{2^1} + \frac{n}{2^2} + \frac{n}{2^3} + \cdots + \frac{n}{2^k}$$

$$\frac{n}{2^k} = 1$$

$$n = 2^k$$

$$\log n = k \log 2$$

$$\therefore k = \log_2 n$$

No. of levels $= (k+1)$

$= (\log_2 n + 1)$

Size of each $= n$

$$T(n) = n(\log_2 n + 1)$$

$$\therefore T(n) = O(n \log_2 n)$$

Space Complexity of merge sort

Space Complexity $= 4$

$= \log_2 n + 1 \quad (n = 8)$

$= \log_2 8 + 1$

$= \log_2 2^3 + 1$

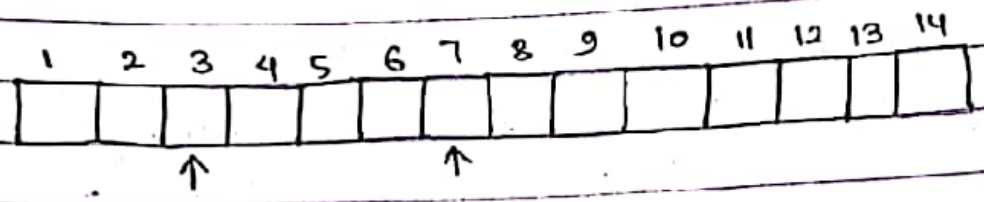
$= 3 + 1$

$= 4$

In general space Complexity $= \log_2 n + 1$

$\theta(\log_2 n)$

Time complexity of Binary Search :



$$mid = \frac{low + high}{2}$$

$$= \frac{1 + 14}{2} = 7$$

for 7

for Left of 7

low = 1, high = mid - 1 = 6

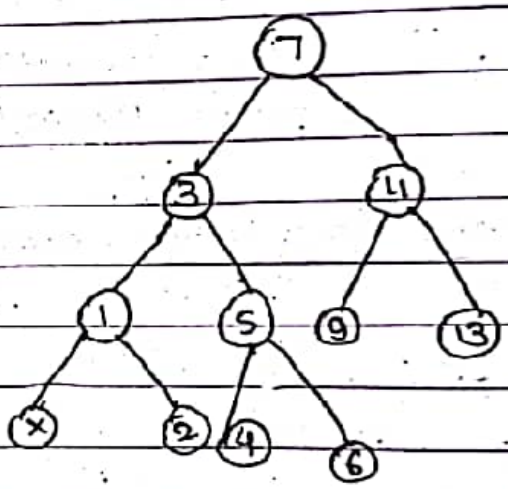
$$mid = \frac{1+6}{2} = 3$$

for right of 7

low = mid + 1, high = 14

= 7 + 1 = 8

$$mid = \frac{8+14}{2} = 11$$



for 3

for left of 3

low = 1, high = mid - 1 = 3 - 1 = 2

$$mid = \frac{1+2}{2} = 1$$

for right of 3

low = mid + 1, high = 6

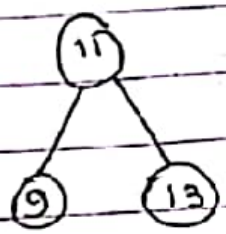
= 3 + 1

= 4

$$mid = \frac{4+6}{2} = 5$$

for 11

for left of 11



low = 8 , high = mid - 1 = 11 - 1 = 10

$mid = \frac{8+10}{2} = 9$

for right of 11

low = mid + 1 , high = 14

= 11 + 1

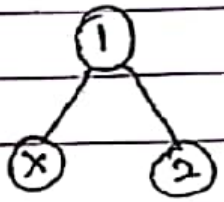
= 12

$mid = \frac{12+14}{2} = 13$

for 1

for left of 1

low = 0 , high = 0



for right of 1

low = mid + 1 high = 2

= 1 + 1

= 2

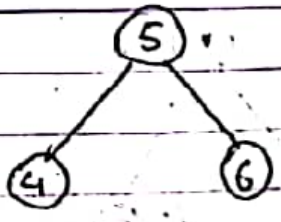
$mid = \frac{2+2}{2} = 2$

for 5

for left of 5

low = 4 , high = mid - 1 = 4

= 5 - 1



$mid = \frac{4+4}{2} = \frac{8}{2} = 4$

for right of 5 low = mid + 1 , high = 6

= 5 + 1

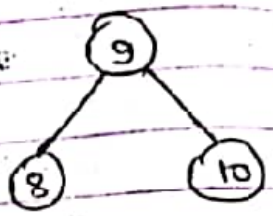
= 6

$\therefore mid = \frac{6+6}{2} = 6$

for 9

left low = 8, high = mid - 1 = 9 - 1 = 8

$$mid = \frac{8+8}{2} = 8$$



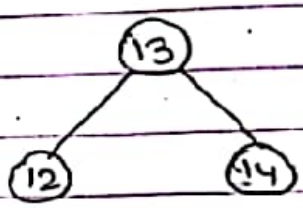
Right, low = mid+1 = 8+1 = 10, high = 10

$$mid = \frac{10+10}{2} = 10$$

for 13

left 12=low, high = mid - 1 = 13 - 1 = 12

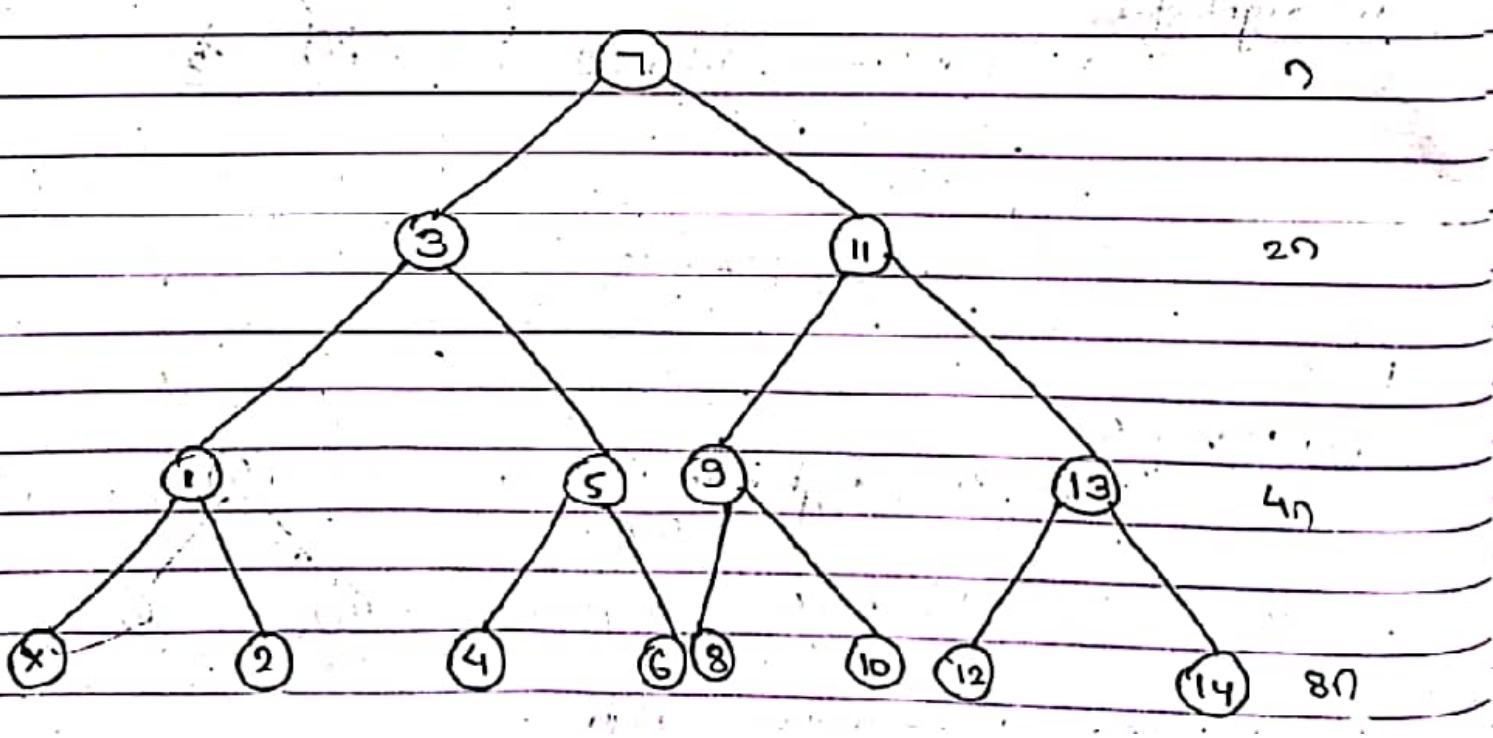
mid = 12



Right.

low = mid+1 = 13+1 = 14, high = 14

mid = 14



(worst case)

time complexity $(T(n)) = T(n/2) + 2$

$= [T(n/4) + 2] + 2$

$= [T(n/8) + 2] + 2 + 2$

$= T(n/8) + 6$

$= T(n/2^3) + 2 * 3$

$\vdots$

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

to terminate, $\frac{n}{2^k} = 1$

$n = 2^k$

$\log n = k \log 2$

$\therefore k = \log_2 n$

Therefore, $T(n) = T(1) + 2\log_2 n$

$T(n) = \theta(\log_2 n)$ is the time complexity of binary search.

$\log_2 n = \log_2 (14) = 4$

Best case $(T(n)) = 1$

worst case $T(n) = \log_2 n$

Average case $T(n) = O(\log_2 n)$

Selection Sort

The idea of Selection Sort is rather simple. We repeatedly find the next largest element in the array and move it to its final position in the sorted array. Assume that we wish to sort the array in increasing order, ie. the smallest element at the beginning of the array and the largest element at the end. We begin by selecting the largest element and moving it to the highest index position. We can do this by swapping the element at the highest index and the largest element. We can reduce the effective size of the array by the one element and repeat the process on the smaller sub array. The process stops when the effective size of the array ~~elements~~ becomes 1. Thus, the selection sort works by selecting the smallest unsorted item remaining in the list, and then swapping it with the item in the next position to be filled. The selection sort has a complexity of $O(n^2)$.

```
Selection Sort(A)
    n:= length[A]
    for j:= 1 to n-1
        smallest := j
        for i:= j+1 to n
            if A[i] < A[smallest]
            then smallest := i
        exchange(A[j], A[smallest])
```

Selection sort is very easy to analyze since none of the loops depends on the data in the array. Selecting the lowest elements requires scanning all n elements and then swapping

it into the first position. Finding the next lowest element requires scanning the remaining n-1 elements and so on, for a total of $(n-1)+(n-2)+\cdots+2+1 = \Theta(n^2)$ comparisons. Each of these scans requires one swap for a total of n-1 swaps. Thus, the comparisons dominate the running time, which is $\Theta(n^2)$.

Eg: Sort the following using selection sort A[] = {5, 2, 1, 4, 3}

Soln:

	1	2	3	4	5
A[] =	5	2	1	4	3

Here n = 5

for j = 1 to 4

j = 1, smallest = 1

for i = 2 to 5

i = 2, Smallest = 1

A[2] = 2 A[1] = 5

A[2] < A[1]

then Smallest = 2

Now, i = 3, Smallest = 2

A[3] = 1 A[2] = 2

A[3] < A[2]

then smallest = 3

Now, i = 4, smallest = 3

A[4] = 4, A[3] = 1

A[4] > A[3] No change.

Now, i = 5, smallest = 3

A[5] = 3

A[3] = 1

A[5] > A[3] No change

then

exchange (A[1], A[smallest])

ie. exchange (5, 1)

Now,

A[ ] = | 1 | 2 | 5 | 4 | 3 |

Now, j = 2 , smallest = 2

for i = 3 to 5

i = 3 , smallest = 2

A[3] = 5

A[2] = 2 A[3] > A[2] No change

Now, i = 4 , smallest = 2 no change

i = 5 , Smallest = 2 no change

Now, j = 3 , smallest = 3

for i = 4 to 5

i = 4, smallest = 3

A[4] = 4

A[3] = 5 A[4] < A[3] then Smallest = 4

Now, i = 5 smallest = 4

A[5] = 3

A[4] = 4

A[5] < A[4] then Smallest = 5

Now exchange (A[3], A[5])

then

A[ ] = | 1 | 2 | 3 | 4 | 5 |

Now, j = 4, Smallest = 4, i = 5

A[5] = 5

A[4] = 4

A[5] > A[4] no change

Hence Sorted array

1	2	3	4	5

Time Complexity of Selection Sort:

$O(n^2)$

Quick Sort:

Quick Sort is based on the concept of divide and Conquer. It is an algorithm of choice in many situations because it is not difficult to implement. It is a good "general purpose" sort and it consumes relatively fewer resources during execution. It requires only $n \log n$ time to sort n items. It has an extremely short inner loop.

Disadvantages:

→ It is recursive, the implementation is extremely complicated.

→ It requires (n^2) time in the worst-case.

Quick Sort works by partitioning a given array $A[p, \ldots r]$ into two non-empty sub arrays $A[p \ldots q]$ and $A[q+1 \ldots r]$ such that every key in $A[p \ldots q]$ is less than or equal to every key in $A[q+1 \ldots r]$. Then the two sub arrays are sorted by recursive calls to quick sort. The exact position of the partition depends on the given array and index q is computed as a part of the partitioning procedure.

Algorithm

```
Quick-Sort (A, p, r)
if p < r then                          - θ(1)
    q ← partition (A, p, r)            - θ(n)
Quick-Sort (A, p, q-1)                 - T(n/2)
Quick-Sort (A, q+1, r)                 - T(n/2)
```

Partitioning the array

```
partition (A, p, r)
    x ← A[r]
    i ← p-1
```

```
for j ← p to x-1                    - θ(n)
    do if A[j] ≤ x
then  i ← i+1
      exchange  A[i] ↔ A[j]
exchange  A[i+1] ↔ A[x]
return i+1
```

Partition selects the first key, A[p] as a pivot key about which the array will be partitioned.

Time complexity of quick sort is

$$T(n) = 2T(n/2) + \theta(n)$$
$$= \theta(n \log n)$$

↓ log is due to recursion

Example: Apply the quick sort technique on the following list:

A = { 4, 5, 1, 7, 8, 9, 2, 88 }

Solution:

	1	2	3	4	5	6	7	8
A[ ] =	4	5	1	7	8	9	2	88

Here p=1 and x=8 1<8 So apply partition function.

x = A[8] i.e. x=88

i = p-1 i.e. i = 1-1 = 0

Now,

j = 1 to 7

j = 1 and i = 0

A[j] = A[1] = 4 ≤ x i.e. 88 (True)

So, i = 0+1 i.e. i = 1

and

$A[i] \leftrightarrow A[j]$

i.e. $A[1] \leftrightarrow A[1]$ No change

Now,

$j = 2$ and $i = 1$

$A[j] = A[2] = 5 \leq 88$

$i = 1+1 = 2$ and $A[2] \leftrightarrow A[2]$ No change.

Similarly, for $j = 3, 4, 5, 6, 7$

No change.

So, $q = 8$

Now, call Quick sort $(A, 1, 8-1)$ i.e. Quick sort $(A, 1, 7)$

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

Here, $p = 1$, $r = 7$ and $x = A[7] = 2$

Now, $j = 1$ to 6

$j = 1$ and $i = 0$

$A[j] = A[1] = 4 \leq 2$ (false)

Now,

$j = 2$ and $i = 0$

$A[j] = A[2] = 5 \leq 2$ (false)

$j = 3$ and $i = 0$

$A[3] = 1 \leq 2$

So,

$i = 0+1 = 1$ and $A[i] \leftrightarrow A[j]$

i.e; $A[1] \leftrightarrow A[3]$

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

$j = 4$ and $i = 1$

$A[4] = 7 \leq 2$ (false)

Similarly, for $j = 5, 6$, condition is false.

So, now $A[2] \leftrightarrow A[7]$

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

Now, $q = 2$

Call Quick sort $(A, 1, 2-1)$ i.e. Quick sort $(A, 1, 1)$

(a | & 1) so call Quick sort $(A, 2+1, 7)$

i.e. Quick sort $(A, 3, 7)$

3	4	5	6	7
4	7	8	9	5

Here,

$p = 3$, $r = 7$ $3 < 7$ (True).

$x = A[r] = A[7]$

i.e. $x = 5$

$i = p - 1$ i.e. $i = 2$

$j = 3$ to 6

Now $j = 3$, $i = 2$

$A[3] = 4 \leq 5$ (True)

$i = 2 + 1 = 3$

and $A[3] \leftrightarrow A[3]$ No change

Now, $j = 4$ and $i = 3$

$A[4] = 7 \leq 5$ (false)

Now, $j = 5$ and $i = 3$

$A[6] = 8 \leq 5$ (false)

Now, $j = 6$ and $i = 3$

$A[6] = 9 \leq 5$ (false)

Now, $A[3+1] \leftrightarrow A[7]$

$A[4] \leftrightarrow A[7]$

So,

3	4	5	6	7
4	5	8	9	7

and return $q = 4$

Now, call Quicksort $(A, 3, 3)$ $3 < 3$ (false)

Quicksort $(A, 5, 7)$

i.e.

5	6	7
8	9	7

Here,

$p = 5$, $q = 7$ $5 < 7$ (True)

Call partition $(A, 5, 7)$

$x = A[7]$ i.e. $x = 7$

$i = 5 - 1 = 4$

$j = 5$ to 6

Now, $j = 5$ and $i = 4$

$A[5] = 8 \leq 7$ (false)

Now, $j = 6$ and $i = 4$

$A[6] = 9 \leq 7$ (false)

Now,

$A[4+1] \leftrightarrow A[7]$

$A[5] \leftrightarrow A[7]$

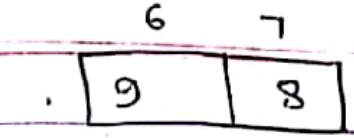
i.e.

5	6	7
7	9	8

and return $q = 5$

Now, call Quick sort $(A, 5, 4)$ $5 < 4$ (False)

Call Quick sort $(A, 6, 7)$



Here,

$p = 6 \quad r = 7$

$6 < 7$ So call partition $(A, 6, 7)$

$x = A[7] = 8$

$i = 6 - 1 = 5$

$j = 6$ to 6

Now, $j = 6$ and $i = 5$

$A[6] = 9 \leq 8$ (false)

$A[5+1] \leftrightarrow A[7]$

$A[6] \leftrightarrow A[7]$

ie.

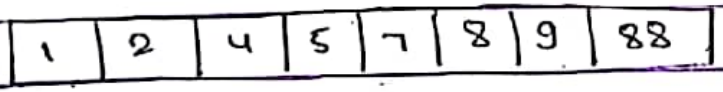
return $q = 6$

Now, call Quick sort $(A, 6, 5)$, $6 < 5$ (false)

call Quick sort $(A, 6+1, 7)$

Quick Sort $(A, 7, 7)$ $7 < 7$ (false)

So final sorted list is



2.4 Strassen's matrix multiplication:

Let A and B be two $n \times n$ matrices. The product matrix $C = AB$ is also an $n \times n$ matrix whose i, jth element is formed by taking the elements in the ith row of A and the jth column of B and multiplying them to get

$$C(i,j) = \sum_{1 \le k \le n} A(i,k)\, B(k,j)$$

for all i and j between 1 and n. To compute $C(i,j)$ using this formula, we need n multiplications. As matrix C has n^2 elements, the time for the resulting matrix multiplication algorithm, which we refer to as the conventional method is $\theta(n^3)$.

The divide-and-conquer strategy suggests another way to compute the product of two $n \times n$ matrices. For simplicity we assume that n is a power of 2 i.e. that there exists a non-negative integer k such that $n = 2^k$. In case n is not a power of two, then enough rows and columns of zeros can be added to both A and B so that the resulting dimensions are a power of two. Imagine that A and B are each partitioned into four square submatrices, each submatrix having dimensions $n/2 \times n/2$. Then the product AB can be computed by using the above formula for the product of 2×2 matrices: if AB is

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

then,

$$C_{11} = A_{11} B_{11} + A_{12} B_{21}$$

$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$
— (1)

The overall computing time $T(n)$ of the resulting divide-and-conquer algorithm is given by the recurrence

$$T(n) = \begin{cases} b & n \le 2 \\ 8T(n/2) + cn^2 & n > 2 \end{cases}$$

where b and c are constants.

This recurrence can be solved in the same way as earlier recurrences to obtain $T(n) = O(n^3)$. Hence no implementation over the conventional method has been made.

Since, matrix multiplication are more expensive than matrix additions ($O(n^3)$ versus $O(n^2)$), we can attempt to reformulate the equations for C_{ij} so as to have fewer multiplications and possibly more additions. Volker Strassen has discovered a way to compute the C_{ij}'s of eqn (1) using only 7 multiplications and 18 additions or subtractions. His method involves first computing the seven $n/2 \times n/2$ matrices P, Q, R, S, T, U and V as shown below.

$$P = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$Q = (A_{21} + A_{22})B_{11}$$
$$R = A_{11}(B_{12} - B_{22})$$
$$S = A_{22}(B_{21} - B_{11})$$
$$T = (A_{11} + A_{12})B_{22}$$
$$U = (A_{21} - A_{11})(B_{11} + B_{12})$$
$$V = (A_{12} - A_{22})(B_{21} + B_{22})$$

Then C_{ij}'s are computed using the formula

$C_{11} = P + S - T + V$

$C_{12} = R + T$

$C_{21} = Q + S$

$C_{22} = P + R - Q + U$

The resulting recurrence relation for $T(n)$ is

$$T(n) = \begin{cases} b & n \leq 2 \\ 7T(n/2) + an^2 & n > 2 \end{cases}$$

where a and b are constants, working with this formula, we get

$$T(n) = an^2 \left[1 + (7/4) + (7/4)^2 + \cdots + (7/4)^{k-1}\right] + 7^k T(1)$$

$$\leq cn^2 (7/4)^{\log_2 n} + 7^{\log_2 n}, \text{ c a constant}$$

$$= cn^{\log_2 4 + \log_2 7 - \log_2 4} + n^{\log_2 7}$$

$$= O(n^{\log_2 7})$$

$$\approx O(n^{2.81})$$

2.5 Convex Hull

A convex hull is an important structure in geometry that can be used in the construction of many other geometric structures. The convex hull of a set S of points in the plane is defined to be the smallest convex polygon containing all the points on S.

A polygon is defined to be convex if for any two points p_1 and p_2 inside the polygon, the directed line segment from p_1 to p_2 is fully contained in the polygon.

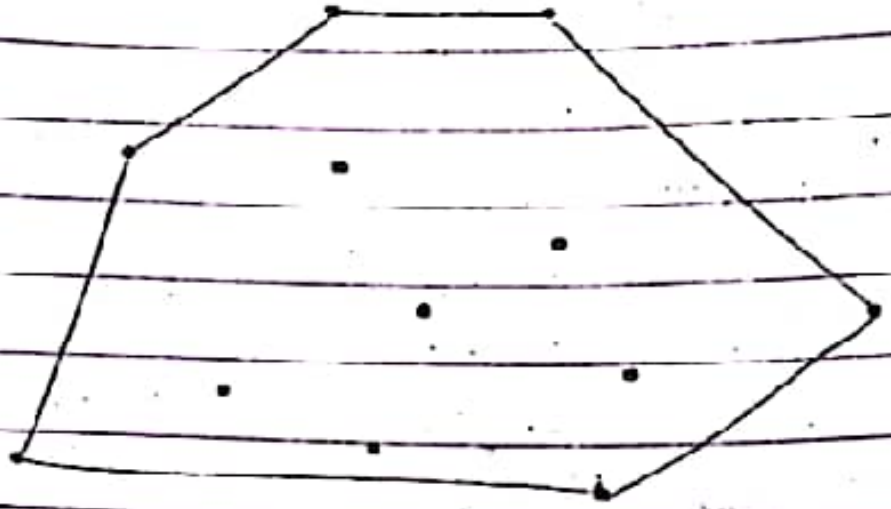


fig: Convex hull

The vertices of the convex hull of a set S of points form a subset of S. There are two variats of the convex hull problem : i) obtain the vertices of the convex hull and

ii) obtain the vertices of the convex hull in some order.

Using divide-and conquer, we can solve both versions of the convex hull problem in $\Theta(n \log n)$ time. It has a worst-case time of $\Theta(n^2)$ whereas its average time is $\Theta(n \log n)$.

Chapter-5

Backtracking

5.1 General Strategy:

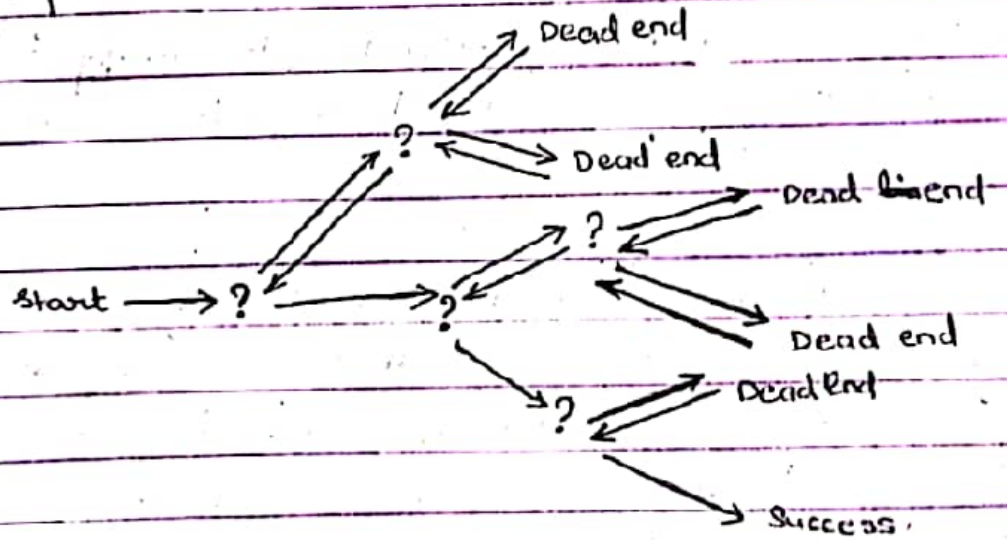
Many problems which deal with searching for a set of solutions or which ask for an optimal solution satisfying some constraints can be solutions or which ask for an optimal solution satisfying some constraints can be solved using the backtracking formulation.

Backtracking is a methodical way of trying out various sequence of decisions, until we find one that "works".

In the following figure,

→ Each non-leaf node in a tree is a parent of one or more other nodes (its children)

→ Each node in the tree, other than the root, has exactly one parent.



Backtracking can be thought of as searching a tree for a particular "goal" leaf node. Backtracking is really quite simple - we "explore" each node as follows:

To "explore" node N:

1) If N is a goal node, return "success"
2) If N is a leaf node, return "failure".

3) For each child c of N,

Explore c

If c was successful, return "Success"

4. Return "failure"

The basic idea of backtracking is to build up a vector one component at a time and to test whether the vector being formed has any chance of success. The major advantage of backtracking algorithm is that if it is realized that the partial vector generated does not lead to an optimal solution then that vector may be ignored.

Backtracking algorithm determines the solution by systematically searching the solution space for the given problem. Backtracking is a depth first search with some bounding function. All solutions using backtracking are required to satisfy a complex set of constraints. The constraints may be explicit or implicit.

5.2 8-Queens problems:

N-queens problem is to place n-queens in such a manner on an nxn chessboard that no two queens attack each other by being in the same row, column or diagonal.

It can be seen that for n=1, the problem has a trivial solution, and no solution exists for n=2 and n=3. So first we will consider the 4-queens problem and then generalize it to n-queens problem.

Given a 4x4 chessboard and number the rows and columns of the chessboard 1 through 4.

	1	2	3	4
1				
2				
3				
4				

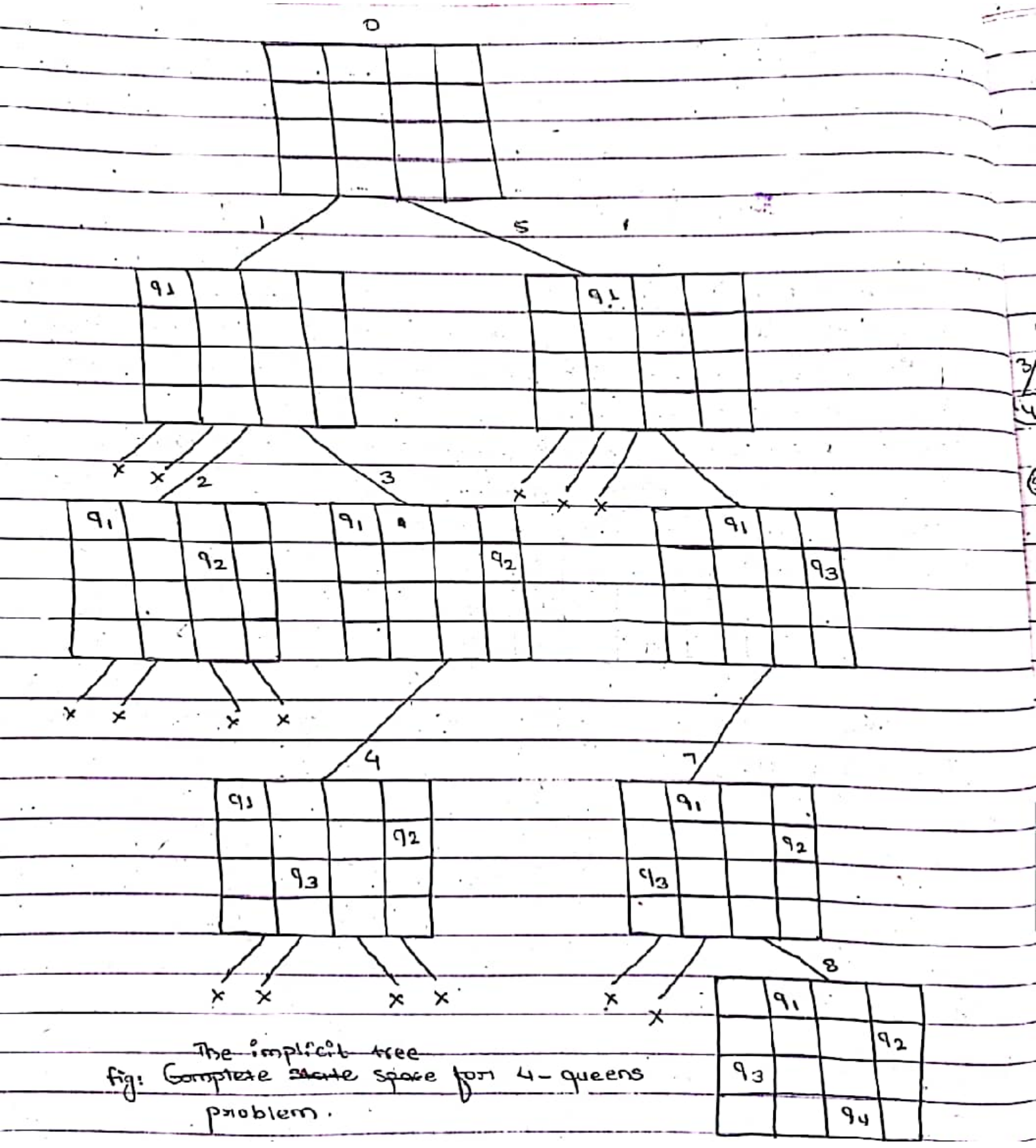
4×4 chessboard

Since, we have to place 4 queens such as q_1, q_2, q_3 and q_4 on a chessboard, such that no two queens attack each other. In such a condition each queen must be placed on a different row, i.e, we place queen 'i' on row "i".

The solution for 4-queen's problem is (3, 1, 4, 2) i.e.;

	1	2	3	4
1			q_1	
2	q_2			
3				q_3
4		q_4		

The implicit tree for 4-queen problem for solution (2,4,1,3) is as follows:



The implicit tree

fig: Complete state space for 4-queens problem.

Backtracking method is used to generate the necessary node and stop if next node violates the rule i.e., if two queens are attacking.

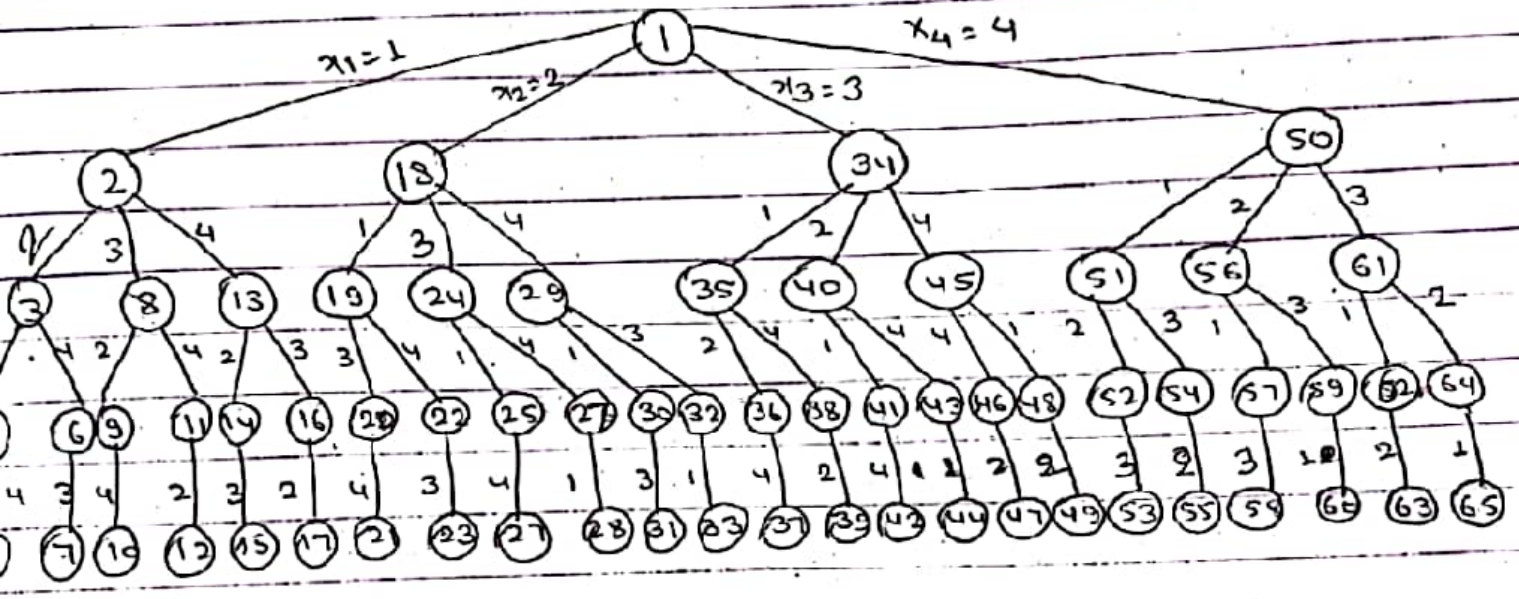


fig: 4-queens solution space with nodes numbered in DFS

It can be seen that all the solutions to the 4-queens problem can be represented as 4-tuples (x_1, x_2, x_3, x_4) where x_i represents the column on which queen q_i is placed.

One possible solution for 8 queens problem is shown below:

	1	2	3	4	5	6	7	8
1				q_1				
2						q_2		
3								q_3
4		q_4						
5							q_5	
6	q_6							
7			q_7					
8					q_8			

Thus, solution for 8-queen problem is { 4, 6, 8, 2, 7, 1, 3, 5 }

If two queens are placed at positions (i, j) and (k, l). Then, they are on the same diagonal only if $(i-j) = k-l$ or $i+j = k+l$.

Then first equation implies that $j-l = i-k$

The 2nd equation implies that $j-l = k-i$.

Therefore, two queens lie on the same diagonal if and only if $|j-l| = |i-k|$.

```
Algorithm NQueens (k,n)
{
  for i:=1 to n do
  {
    if place (k, i) then
    {
      x[k] := i;
      if (k=n) then write (x[1:n]);
      else NQueens (k+1, n);
    }
  }
}
```

8-queens problem is about placing 8 queens in 8X8 chessboard in such a way that they do not attack eachothe.

```
Algorithm place(k,i)
// Return true if a queen can be placed in kth row and
// ith column. Otherwise, it returns false. x[] is a global
// array whose first (k-1) value has been set. Abs(r) returns
// the absolute value of r.
{
  for j:=1 to k-1 do.
    if((x[j]=i) // two in the same column
      or Abs(x[j]=i)= Abs(j-k))
    // or in the same diagonal
    then return false;
    return value;
}
```

Algorithm: Can we a new queen be placed?

5.3 knapsack problem:

Given n positive weights w_i, n profits(+ve) p_i, and a positive number 'm' which is the knapsack capacity, this problem (0/1 knapsack problem) calls for choosing a subset of the weights such that

$$\sum w_i x_i \leq m \quad \text{and}$$

$$1 \leq i \leq n$$

$$\sum p_i x_i \text{ is maximized.}$$

$$1 \leq i \leq n$$

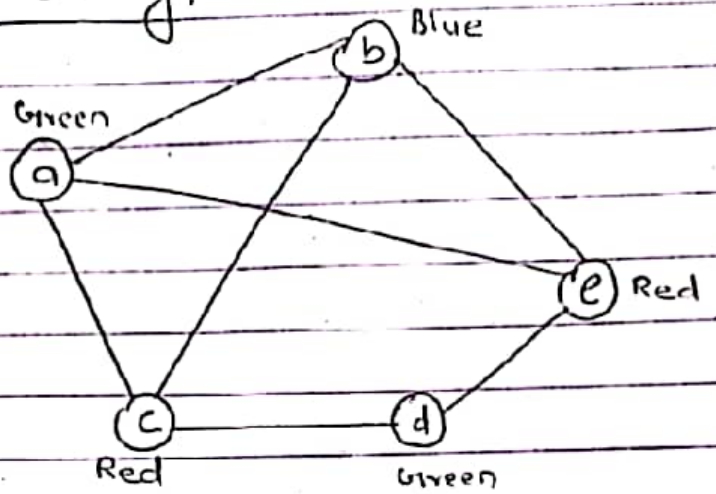
The x_i's constitute a zero-one valued vector.

The solution space for this problem consists of the 2^n distinct ways to assign 0 or 1 to the x_i's.

Backtracking algorithm for the knapsack problem can be arrived at using either of state space tree. A bounding functions are needed to kill some live nodes without expanding them. A good bounding function for this problem is obtained by using an upper bound on the value of the best feasible solution obtained by expanding the given live node.

```
Algorithm Bknap(k, cp, cw)
{
    if (cw + w[k] ≤ m) then
    {
        y[k] = 1;
        if (k < n) then Bknap(k+1, cp + p[k], cw + w[k]);
        if ((cp + p[k] > fp) and (k = n)) then
        {
            fp = cp + p[k]; fw = cw + w[k];
            for j = 1 to k do x[j] = y[j];
        }
    }
    if Bound((cp, cw, k) ≥ fp) then
    {
        y[k] = 0 if (k < n) then Bknap(k+1, cp, cw);
        if ((cp > fp) and (k = n)) then
        {
            fp = cp; fw = cw;
            for j = 1 to k do x[j] = y[j];
        }
    }
}
```

5.4) Graph coloring:



Minimum no. of colors used = 3 (R, G, B)

Let G be a graph and m be a given positive integer. We want to discover whether the nodes of G can be colored in such a way that no two adjacent nodes have the same colour yet only m colours are used. This is termed as the m-colourability decision problem.

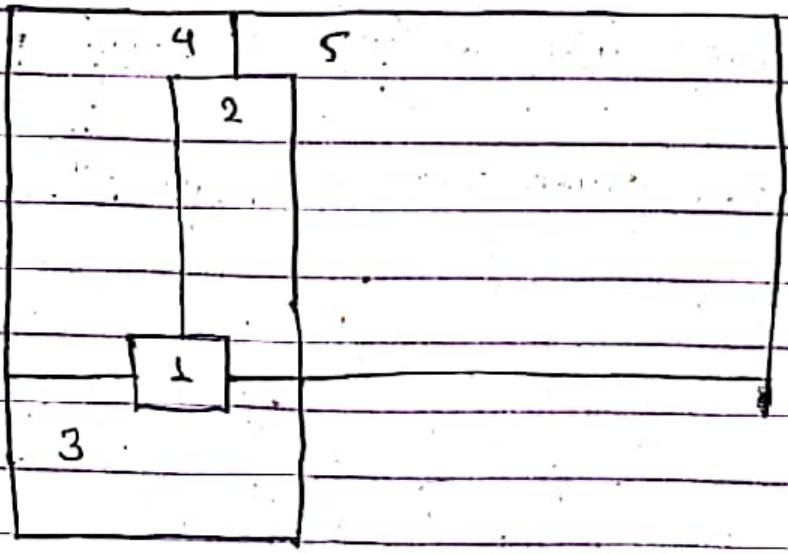
The m-colourability optimization problem asks for the smallest integer m for which the graph G can be coloured. This integer is referred to as the chromatic number of graph.

For example, for the above given graph, at least three colours are required, and so its chromatic number is 3.

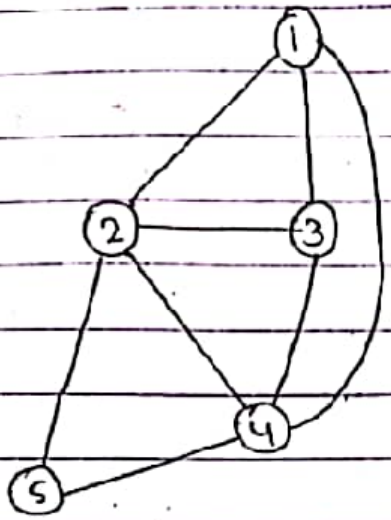
The algorithm for graph coloring is given below:

```
Algorithm m coloring(k)
// This algorithm was formed using the recursive
// backtracking schema. The graph is represented by its boolean
// adjacency matrix G[1:n, 1:n]. All assignments of 1, 2, ----, m
// to the vertices of the graph such that adjacent vertices are
// assigned distinct integers are printed. k is the index of
// the next vertex to color.
{
repeat
    { // Generate all legal assignments for x[k].
    NextValue(k); // Assign to x[k] a legal color.
    if (x[k] = 0) then return ; // No new color possible.
    if (k=n) then // At most m colors have been
                  //used to color the n vertices.
    write(x[1:n]);
    else mcoloring(k+1);
} until (false);
}
```

for example: Given this map, find out its chromatic number.



Soln: Chromatic number = 4



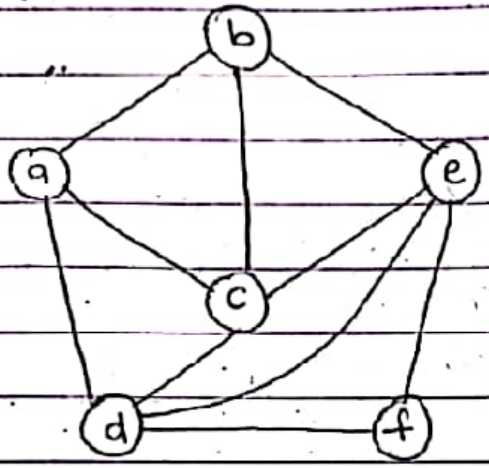
A map and its planar graph representation

5.5 Hamiltonian Cycles:

Given a graph $G = (V, E)$, we have to find the Hamiltonian circuit using Backtracking approach. We start our search from any arbitrary vertex, say 'a'. This vertex 'a' becomes the root of our implicit tree. The first element of our partial solution is the first immediate vertex of the Hamiltonian cycle that is to be constructed. The next adjacent vertex is selected on the basis of an alphabetical (or numerical) order. If at any stage any arbitrary vertex makes a cycle with any vertex other than vertex 'a' then we say that dead end is reached. In this case, we backtrack one step, and again the search begins by selecting another vertex and backtrack the element from the partial solution must be removed. The search using backtracking is successful if a Hamiltonian cycle is obtained.

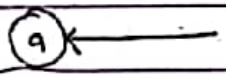
Example:

Consider a graph $G=(V,E)$ shown below and find out a Hamiltonian circuit using backtracking method.

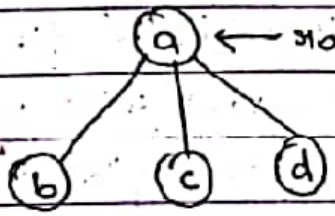


Soln:

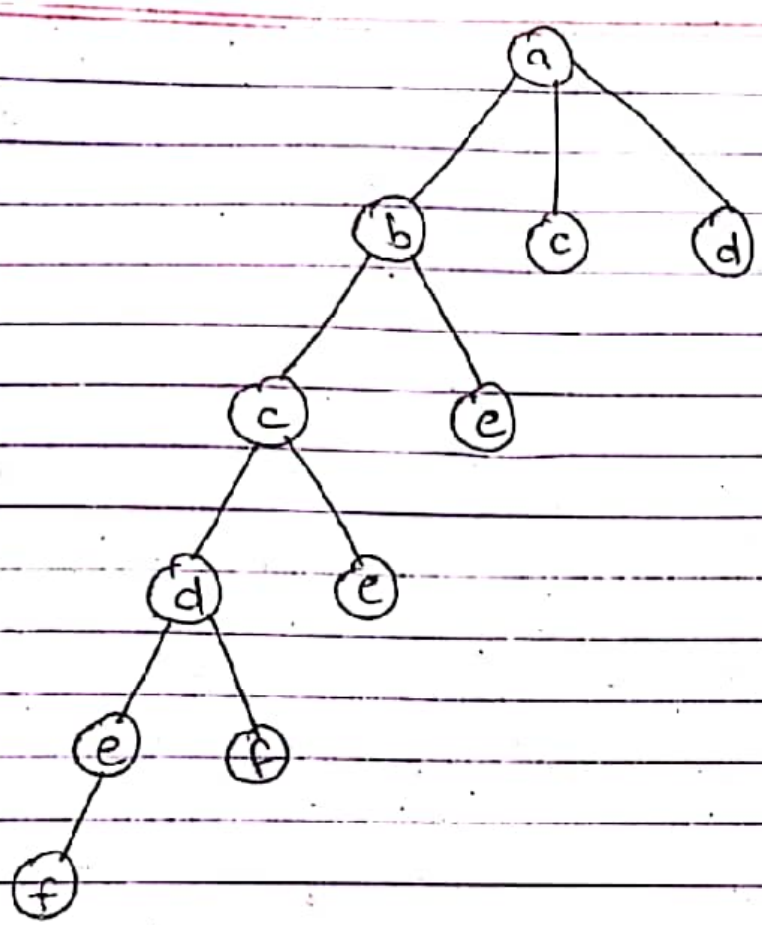
Firstly, we start our search with vertex 'a'. This vertex 'a' becomes the root of our implicit tree.



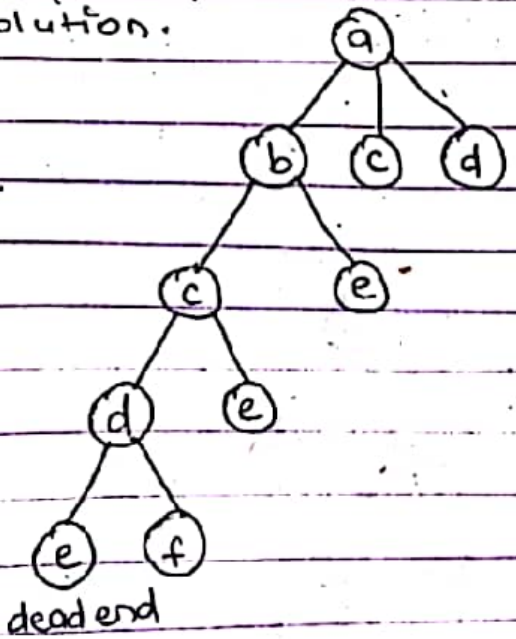
Next we choose vertex 'b' adjacent to a in lexicographical order (b, c, d).



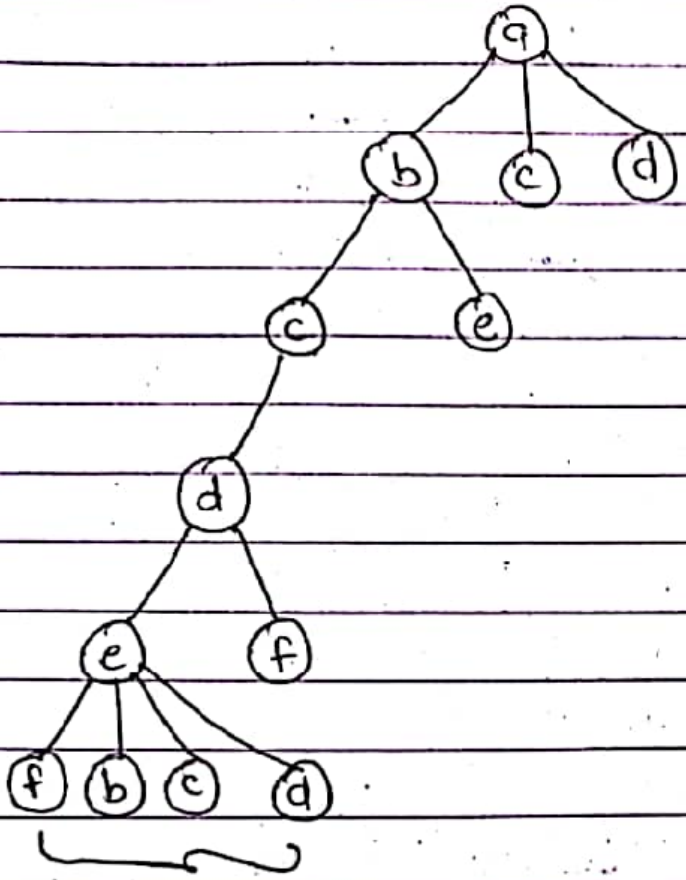
Next, we select 'c' adjacent to 'b' and then we select 'd' adjacent to 'c'.



The vertex adjacent to 'f' and 'd' and 'e' but they have been already visited. Thus, we get the deadend and we backtrack one step and remove the vertex 'f' from partial solution.

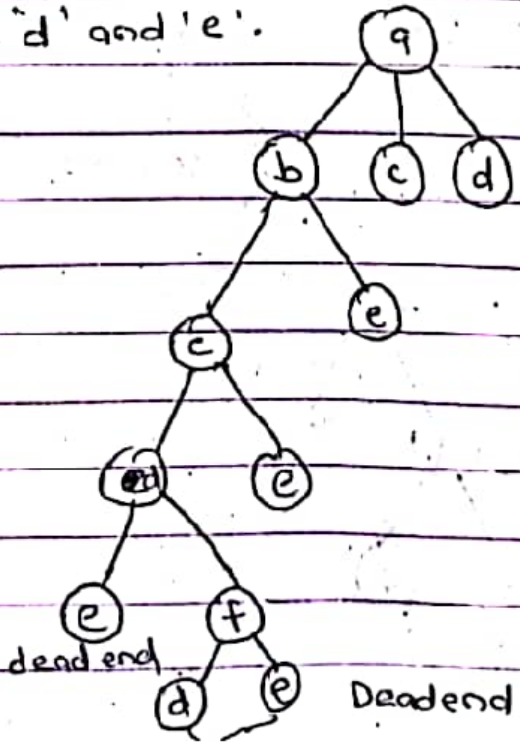


From backtracking, the vertex adjacent to 'e' are b, c d, f from which vertex 'f' has already been checked and 'b', 'c', 'd' have already visited.

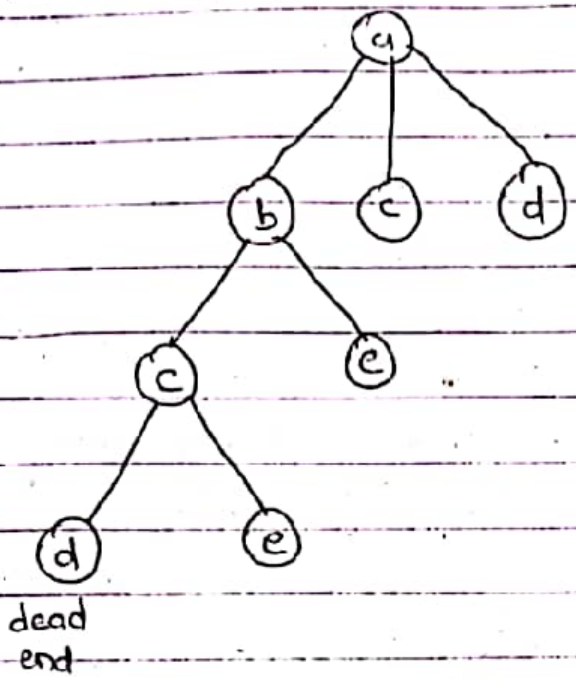


dead ends

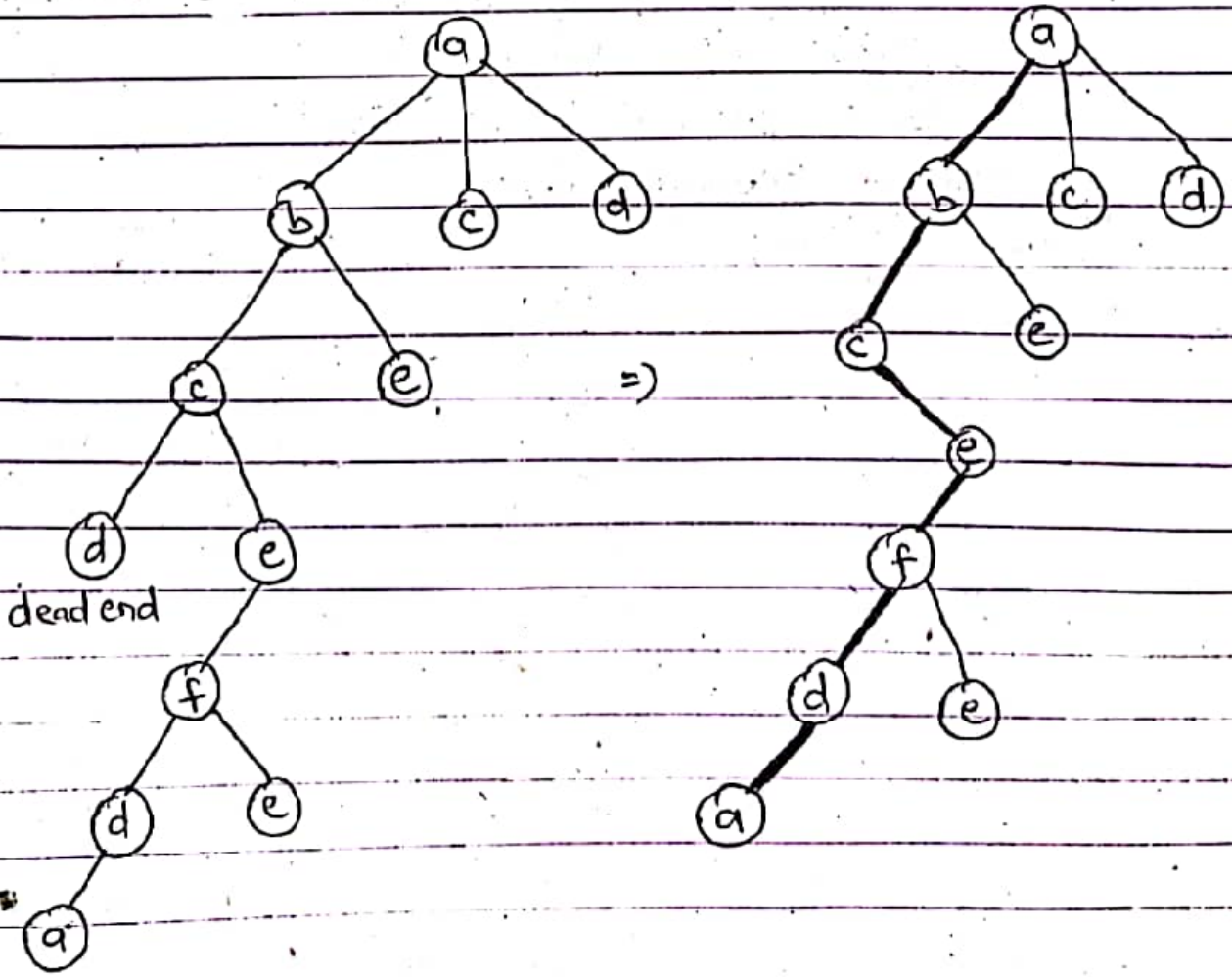
So, again we backtrack one step. Now, the vertex adjacent to 'd' are 'e', 'f' from which 'e' has already been checked and adjacent of 'f' are 'd' and 'e'.



If 'e' vertex visited then again we get dead state. So again we backtrack.



Now, adjacent to C is 'e' and adjacent to 'e' is 'f' and adjacent to 'd' is 'a'.



Here, we get the Hamiltonian Circuit or cycle as all the vertex other than start vertex 'a' is visited only once.

a → b → c → e → f → d → a is Hamiltonian cycle.

```
Algorithm Hamiltonian(k)
// This algorithm uses the recursive formulation of backtracking
// to find all the Hamiltonian cycles of a graph. The graph is
// stored as an adjacency matrix G[1:n, 1:n]. All cycles to
// begin at node 1.
{
  Repeat
  {// Generate values for x[k].
      NextValue(k); //Assign a legal value to x[k].
      if (x[k] = 0) then return;
      if (k=n) then write (x[1:n]);
      else Hamiltonian(k+1);
  } until (false);
}
```

Chapter-4
Dynamic programming

4.1 The General Method:

Dynamic programming is an algorithm design method that can be viewed as the result of a sequence of decisions. Dynamic programming is a powerful technique that can be used to solve many problems in time $O(n^2)$ or $O(n^3)$ for which a naive approach would take exponential time. Dynamic programming solutions have a polynomial complexity which assures a much faster running time than other techniques like backtracking, etc.

Eg:1 knapsack

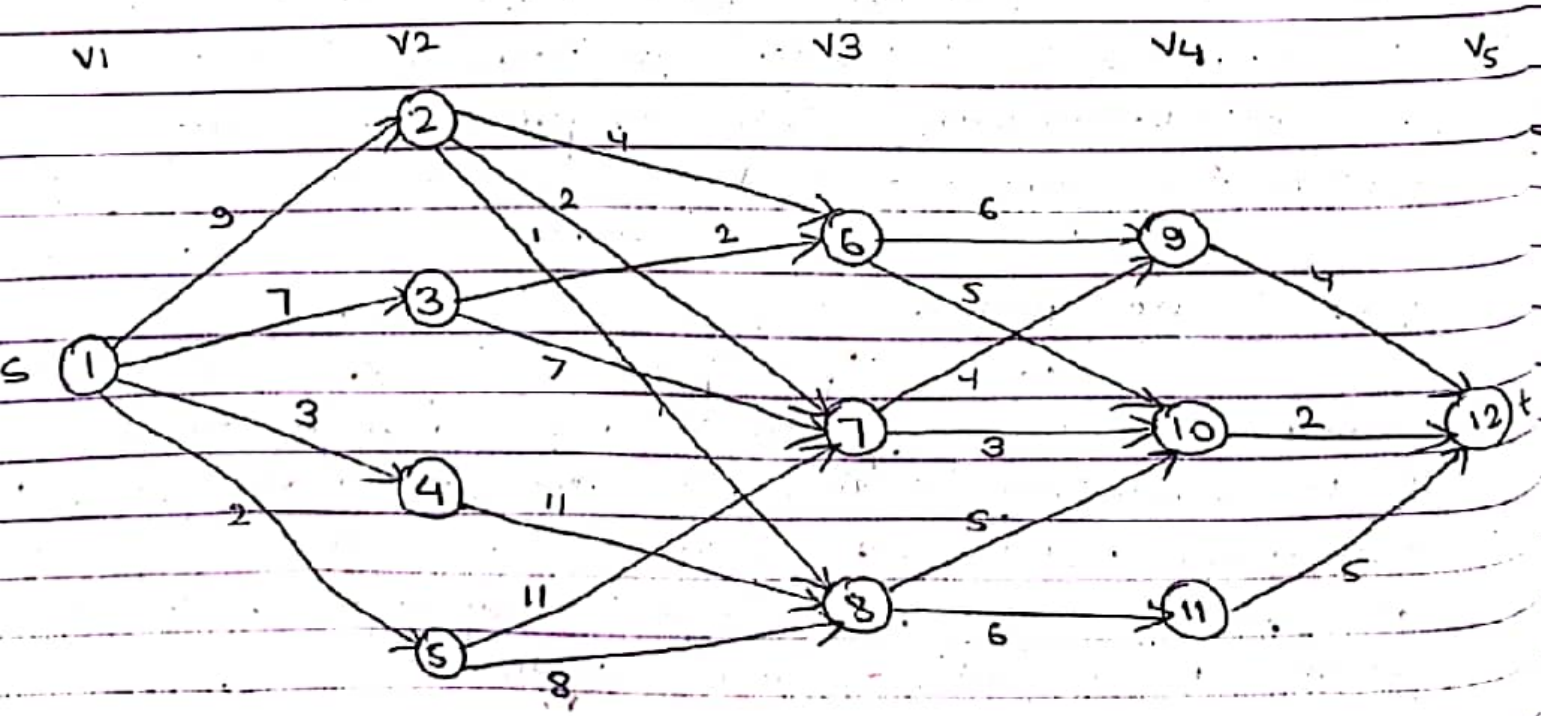
The solution to the knapsack problem can be viewed as the result of a sequence of decisions. We have to decide the values of x_i, $1 \le i \le n$. First we make a decision on x_1, then on x_2, then on x_3 and so on. An optimal sequence of decisions maximizes the objective function $\sum p_i x_i$ satisfying the constraints $\sum w_i x_i \le m$ and $0 \le x_i \le 1$.

Eg2: Shortest path:

One way to find a shortest path from vertex i to vertex j in a directed graph G is to decide which vertex should be the second vertex, which the third, which the fourth and so on, until vertex j is reached. An optimal sequence of decision is one that results in a path of least length.

4.2 Multistage Graph:

A multistage graph $G = (V, E)$ is a directed graph in which the vertices are partitioned into $k \geq 2$ disjoint sets V_i, $1 \leq i \leq k$. Such that if (u, v) is an edge in E, then $u \in V_i$ and $v \in V_{i+1}$ for some i, $1 \leq i \leq k$. The sets V_1 and V_k are such that $|V_1| = |V_k| = 1$. Let s and t, respectively be the vertices in V_1 and V_k. (The vertex s is the source, and t is the sink. Let $c(i, j)$ be the cost of edge (i, j). The multi-stage problem is to find a minimum cost path from s to t. Each set V_i defines a stage in the graph. The cost of a path from s to t is the sum of costs of the edges on the path. 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 and so on, and eventually terminates in stage k.)



Above fig. shows a five-stage graph. A minimum cost

s to t is indicated by the broken edges.

Using forward approach:

$$cost(i,j) = \min\{c(j,l) + cost(i+1,l)\}$$

Stage 4

$cost(4,9) = c(9,12) = 4$ via (9-12)

(stage → 4, node → 9)

$cost(4,10) = c(10,12) = 2$ via (10-12)

$cost(4,11) = c(11,12) = 5$ via (11-12)

Stage 3

$cost(3,6) = \min\{6 + cost(4,9), 5 + cost(4,10)\}$

$= \min\{6+4, 5+2\}$

$= \min\{10, 7\}$

$= 7$ (via 6-10-12)

$cost(3,7) = \min\{4 + cost(4,9), 3 + cost(4,10)\}$

$= \min\{4+4, 3+2\}$

$= \min\{8, 5\}$

$= 5$ (via 7-10-12)

$cost(3,8) = \min\{5 + cost(4,10), 6 + cost(4,11)\}$

$= \min\{5+2, 6+5\}$

$= \min\{7, 11\}$

$= 7$ (via 8-10-12)

s to t is indicated by the broken edges.

Using forward approach:

$$Cost(i,j) = \min\{c(j,l) + cost(i+1,l)\}$$

Stage 4

$Cost(4,9) = c(9,12) = 4$ via (9-12)

(Stage / node)

$Cost(4,10) = c(10,12) = 2$ via (10-12)

$Cost(4,11) = c(11,12) = 5$ via (11-12)

Stage 3

$Cost(3,6) = \min\{6 + cost(4,9), 5 + cost(4,10)\}$

$= \min\{6+4, 5+2\}$

$= \min\{10,7\}$

$= 7$ (via 6-10-12)

$Cost(3,7) = \min\{4 + cost(4,9), 3 + cost(4,10)\}$

$= \min\{4+4, 3+2\}$

$= \min\{8,5\}$

$= 5$ (via 7-10-12)

$Cost(3,8) = \min\{5 + cost(4,10), 6 + cost(4,11)\}$

$= \min\{5+2, 6+5\}$

$= \min\{7, 11\}$

$= 7$ (via 8-10-12)

Stage 2

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 }
= 7 (via 2-7-10-12)

Cost(2,3) = min { 2 + cost(3,6), 7 + cost(3,7) }
= min { 2+7, 7+5 }
= min { 9, 12 }
= 9 (via 3-6-10-12)

cost(2,4) = min { 11 + cost(3,8) }
= min { 11+7 }
= 18 (via 4-8-10-12)

Cost (2,5) = min { 8 + cost(3,8), 11 + cost(3,7) }
= min { 8+7, 11+5 }
= min { 15, 16 }
= 15 (via 5-8-10-12)

Stage 1

cost(1,1) = { 7 + cost(2,3), 9 + cost(2,2), 3 + cost(2,4),
2 + cost(2,5) }
= min { 7+9, 9+7, 3+18, 2+15 }
= min { 16, 16, 21, 17 }
= 16 (via 1-2-7-10-12)
(via 1-3-6-10-12)

so, shortest path is 1-2-7-10-12 or
1-3-6-10-12

Using backward approach:

Stage 2

Cost(2,2) = C(1,2) = 9 Via 1-2

Cost(2,3) = C(1,3) = 7 Via 1-3

Cost(2,4) = C(1,4) = 3 Via 1-4

Cost(2,5) = C(1,5) = 2 Via 1-5

Stage 3

Cost(3,6) = min{ 2 + cost(2,3), 4 + cost(2,2) }

= min{ 2+7, 4+9 }

= min{ 9, 13 }

= 9 (via 1-3-6)

Cost(3,7) = min{ 7 + cost(2,3), 2 + cost(2,2), 11 + cost(2,5)

= min{ 7+7, 2+9, 11+2 }

= min{ 14, 11, 13 }

= 11 (via 1-2-7)

cost(3,8) = min{ 1 + cost(2,2), 11 + cost(2,4), 8 + cost(2,5) }

= min{ 1+9, 11+3, 8+2 }

= min{ 10, 14, 10 }

= 10 (via 1-2-8) (via 1-5-8)

Stage 4

cost(4,9) = min{ 6 + cost(3,6), 4 + cost(3,7) }

= min{ 6+9, 4+11 }

= min{ 15, 15 }

(via 1-3-6-9, via 1-2-7-9)

cost(4,10) = min { 5+cost(3,6), 3+cost(3,7), 5+cost(3,8) }

= min { 5+9, 3+11, 5+10 }

= min { 14, 14, 15 }

= 14

(via 1-3-6-10) (via 1-2-7-10)

cost(4,11) = min { 6 + cost(3,8) }

= min { 6+10 }

= 16

(via 1-2-8-11) or (1-5-8-11)

stage 5

cost(5,12) = min { 4+cost(4,9), 2+cost(4,10), 5+cost(4,11) }

= min { 4+15, 2+14, 5+16 }

= min { 19, 16, 21 }

= 16 (via 1-3-6-10-12)

or (via 1-2-7-10-12)

So, the shortest path sequence is 1-3-6-10-12 or

1-2-7-10-12

4.3 0/1 knapsack problem:

In knapsack problem, a solution to the problem can be obtained by making a sequence of decisions on the variables $x_1, x_2, \ldots x_n$. A decision on variable x_i involves determining which of the values 0 or 1 is to be assigned to it. In 0/1 knapsack problem 0/1 indicates either to take the whole or not to take at all ie. no fraction is allowed.

In knapsack problem, a thief robbing a store and carry a maximum weight 'w' into their knapsack. There are 'n' items and i^{th} item weighs w_i and is worth v_i dollars. The problem is that what items should thief takes? There are two types of knapsack problem:

→ 0/1 knapsack problem
→ fractional knapsack problem

In 0/1 knapsack problem, the items may not be broken into smaller pieces, so thief may decide either to take an item or to leave it (binary choice) but may not take a fraction of an item. Since 0/1 knapsack problem exhibits optimal sub-structure property, therefore only dynamic programming algo-rithm exists.

Let i be the highest-numbered item in an optimal solution. We can express this fact in the following formula:
Let $c[i, w]$ to be the solution for items $1, 2, \ldots i$ and maximum weight w. Then,

$$c[i,w] = \begin{cases} 0 & \text{if } i=0 \text{ or } w=0 \\ [i-1, w] & \text{if } w_i \geq w \\ \max\{[v_i + c[i-1, w-w_i], c[i-1,w]\} & \text{if } i>0 \text{ and } w \geq w_i \end{cases}$$

Time complexity of DP = $O(2^n)$ but it will not take that much time.

This says that the value of the solution to 'i' items either include ith item, in which case it is v_i plus a subproblem solution for (i-1) items and weight excluding w_i, or does not including ith item and weight excluding w_i, or does not in which case it is subproblem's solution for (i-1) items and the same weight. That is, if the thief picks item i, thief takes v_i value, and thief can choose from items w-w_i, and get c[i-1, w-w_i] additional value. On the other hand, if thief decides not to take item i, thief can choose from item 1, 2, ---i-1 upto the weight limit. w and get c[i-1,w] value. The better of these two choices should be made.

The algorithm takes as input the max. weight w, the number of items n, and the two sequences $v = \langle v_1, v_2, \ldots v_n \rangle$ and $w = \langle w_1, w_2, \ldots w_n \rangle$. It stores the c[i, j] values in the table, i.e. a two dimensional array c[0...n, 0...w] whose entries are computed in a a major order. That is the first row of c is filled in from left to right, then the second row, and so on. At the end of computation c[n, w] contains the maximum value that can be picked into the knapsack.

$v_i \rightarrow b_k$

$i \longrightarrow k$

$c \longrightarrow B$.

Dynamic problem is used for solving optimization problems
i.e. either max. or min.

=) There are many possible soln $\Rightarrow 2^n$ = Total soln

Algorithm

$$B[k,w] = \begin{cases} 0 & \text{if } k=0 \text{ or } w=0 \\ B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1, w-w_k] + b_k\} & \text{if } i>0 \text{ and } w \geq w_i \end{cases}$$

```
KNAPSACK (n, W)
for w = 0 to W
    B[0, w] = 0
for i = 1 to n
    B[i, 0] = 0
for i = 0 to n
for w = 0 to W
if wi ≤ w    // item i can be part of the solution
if bi + B[i-1, w-wi] > B[i-1, w]
    B[i, w] = bi + B[i-1, w-wi]
else
    B[i, w] = B[i-1, w]
else
    B[i, w] = B[i-1, w]   // wi > w
```

i \ w	0	1	2	3	4	5
0						
1						
2						
3						
4						

Example:

Find the optimal solution for the knapsack 0/1 problem b using dynamic problem approach?

Consider $n=4$, $W=5$

$(w_1, w_2, w_3, w_4) = (2, 3, 4, 5)$ and

$(b_1, b_2, b_3, b_4) = (3, 4, 5, 6)$

Soln:

item →	1	2	3	4
b_i	3	4	5	6
w_i	2	3	4	5

step1: for $w \leftarrow 0$ to W

$B[0, w] = 0$

i.e.

$B[\,][\,]_1 =$

	0	1	2	3	4	5
i \ w	0	0	0	0	0	0
0						
1						
2						
3						
4						

step2:

for $i \leftarrow 1$ to n

$B[i, 0] = 0$

i \ w	0	1	2	3	4	5
0	0	0	6	0	0	0
1	0					
2	0					
3	0					
4	0					

Step 3: for item 1 w_1 b_1

2	3

Now for B[1, 1]

↓ i ↓ w

if $[w_1 \leq w]$ ⇒ false

So, $B[i, w] = B[i-1, w]$

$B[1, 1] = B[1-1, w]$

$= B[0, 1]$

$= 0$

i\w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0				
2	0					
3	0					
4	0					

for [1, 2]

↓ i ↓ w

now if $(w_1 \leq W)$ ⇒ true

and if $(b_i + B[i-1, w-w_i] > B[i-1, w])$

if $(3 + B[1-1, 2-2] > B[0, 2])$

if $(3 + B[0, 0] > B[0, 2])$

if $(3 + 0 > 0)$

if $(3 > 0$ ⇒ true.)

then, $B[i, w] = b_i + B[i-1, w-w_i]$

$B[1, 2] = 3 + B[0, 0] = 3$

i \ w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3			
2	0					
3	0					
4	0					

for B[1,3]

i w

if $(w_i < w)$ => true

and if $(b_i + B[i-1, w-w_i]) > B[i-1, w]$

if $(3 + B[0,1] > B[0,3])$

if $(3+0 > 0)$

if $(3 > 0$ => true)

then,

$B[i,w] = b_i + B[i-1, w-w_i]$

$B[1,3] = 3 + B[0,1]$

$= 3+0 = 3$

finally,

i \ w	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

Chapter-7

NP-Hard and NP-Complete problems

7.1 Basic concepts:

In this chapter we are concerned with the distinction between problems that can be solved by a polynomial time algorithm and problems for which no polynomial time algorithm is known.

P-class problem:

The P in the P-class problem stands for polynomial time. The class P consists of those problems that are solvable in polynomial time. More specifically the problems that can be solved in time $O(n^k)$ for some constant k, where n is the size of input to the problems.

Time complexity $= O(n^k)$ ← constant

↓ size of input to an algorithm.

It is deterministic.

NP-class problem:

The class NP (not called non-polynomial but it is called non-deterministic polynomial). Non-deterministic means data may or may not be present in array. It consists of those problems that are "verifiable" in polynomial time. Verifiable means we are given a 'certificate' of a solution, then we could verify that certificate is correct in time polynomial in the size of the input.

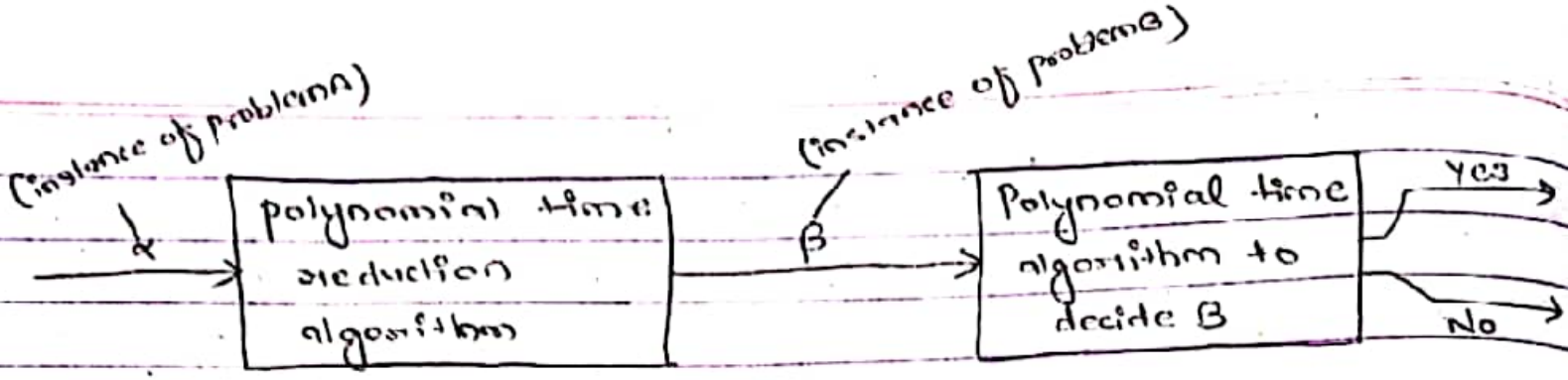


fig: Polynomial time algorithm to decide A.

In polynomial time, we transform an instance α of A into an instance β of B, we can solve B in polynomial time, and we can use the answer for β as the answer for α, so, we can use answer for β as the answer for α.

Eg: knapsack decision problem ~~kdp~~ ie. DKP:

This algorithm is a non-deterministic polynomial time algorithm for the knapsack decision problem

```
Algorithm Dkp (p, w, n, m, x, x)
{
    w:=0 ; p:=0;
    for i:=1 to n do
    {
        x[i] := choice (0,1);
        w:= w+ x[i] * w[j];
        p:= p + x[i]* p[i];
    }
    If ((w>m) or (p<x)) then failure();
    else success();
}
```

Eg: Satisfiability:

Let $x_1, x_2 \dots x_n$ denotes boolean variables. Let $\bar{x}_i$ denote the negation of x_i. A literal is either a variable or its negation. Examples of such formula are $(x_1 \wedge x_2) \vee (x_3 \wedge \bar{x}_4)$ and $(x_3 \vee \bar{x}_4) \wedge (x_1 \vee \bar{x}_2)$. The symbol $\vee$ denotes "or" and $\wedge$ denotes "and". The formula is in conjuctive normal form (CNF) if and only if it is represented as $\bigvee_i \bigwedge_k = c_j$. Thus $(x_1 \wedge x_2) \vee (x_3 \wedge \bar{x}_4)$ is in DNF.

```
Algorithm Eval(E,n)
// Determine whether the propositional formula E is
// satisfiable. The variables are x1, x2 --- xn.
{
    for i:= 1 to n do
        xi = choice(false,true);
    if E(x1, ----, xn) then success(1);
    else failure();
}
```

fig: Non-deterministic satisfiability

The algorithm having the property that the result of every operation is uniquely defined is known as deterministic algorithms.

Algorithms that contain operations whose outcomes are not uniquely defined but are limited to specified sets of possibilities is called non-deterministic algorithm.

i) choice(s) arbitrarily chooses one of the element of set s.

ii) failure() signals as unsuccessful completion.

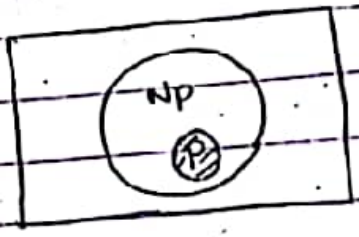
iii) success() signals a successful completion.

Eg: consider the problem of searching for an element x in a given set of elements $A[1:n]$, $n \geq 1$. We are required to determine an index j such that $A[j] = x$ or $j := 0$ if x is not in A.

```
Algorithm search
{
    j := choice(1, n)
    if A[j] := x then
    {
        write (j);
        success();
    }
    write(0);
    failure();
}
```

fig: Non-deterministic search

P and NP-problems:



$P \subseteq NP$

P is a subset of NP. Every P problem is also a NP problem but every NP problem is not a P-problem. In NP-hard problem we are making a statement about how hard it is not how about how easy it is. we are not trying to prove the existence of an efficient algorithm but rather no efficient.

P is the set of all decisions solvable by deterministic algorithms in polynomial time.. Np is the set of all decision problem solvable by non-deterministic algorithms in polynomial time..

Since, deterministic algorithms are just a special case of non-deterministic ones, we can conclude $P \subseteq Np$

Definition of Np-hard and Np-Complete:

A problem L is Np hard if and only if satisfiability reduces to L.

A problem L is Np-complete if and only if L is Np-hard and Np.

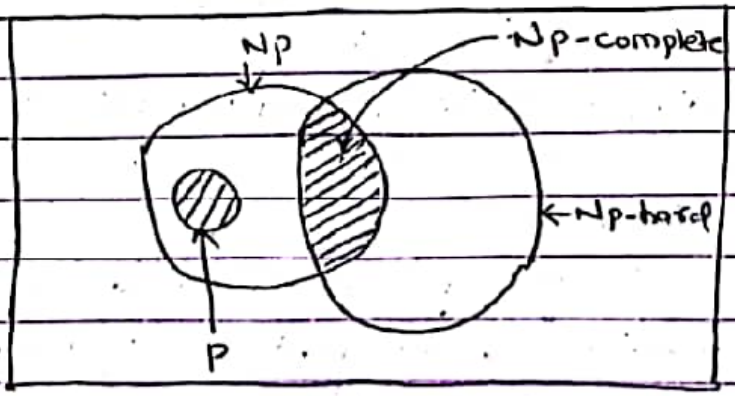
A language $L \subseteq \{0,1\}$ is Np complete if

i. $L' \leq L$ for every $L' \in Np$ and

ii. $L \in Np$

Here $L' \leq_p L$ means L' is reducable in to L in polynomial time.

The commonly believed relationship among P, Np, Np-complete and Np-hard problem is shown below.



It is easy to see that there are Np-hard problems that are not Np-complete. Only a decision problem can be Np-complete. However, an optimization problem can be Np-hard.

Optimization problems can not be Np-complete. There also exist Np-hard decision problems that are not Np-complete.

7.2 Np-hard graph problems:

The strategy we adopt to show that a problem L_2 is Np-hard is:

i. pick a problem L_1 already known to be Np-hard.
ii. show how to obtain (in polynomial deterministic time) an instance I' of L_2 from any instance I of L_1 such that from the solution of I' we can determine the solution of instance I of L_1.
iii. Conclude from step (ii) that $L_1 \propto L_2$.
iv. Conclude from step (i) and (iii) and transitivity of $\propto$ that L_2 is Np-hard.

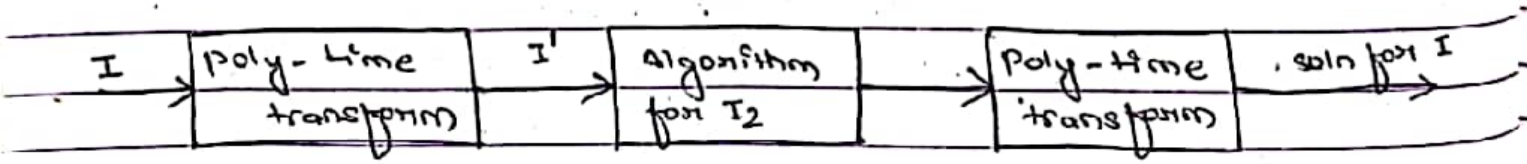


fig: Reduction of L_1 to L_2.

The virtue of conceiving of non-deterministic algorithms is that often what would be very complex to write deterministically. In fact it is very easy to obtain polynomial time non-deterministic algorithm for many problems

that can be deterministically solved by a systematic search of a solution space of exponential size.

Np-hard problem:

→ A problem is said to be Np hard if it cannot be verified that it is solvable in polynomial time.

→ A problem is said to be Np-hard if it is at least as hard as any Np-problem.

The first Np-hard problem is the circuit satisfiability problem, or simply SAT.. Given, a boolean formula, its can be implemented using logic gates, and its truth table can be built.

p	q	r	output
0	0	0	
0	0	1	1
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

If output is 1 for at least one combinations, then it is satisfiable.

SAT problem is to find out time taken to know which of the so many (2^n for n input variables) values of variables i.e., one row in the truth table, the output evaluates to 1

* Prove that a decision problem L_1, is a Np-hard problem.

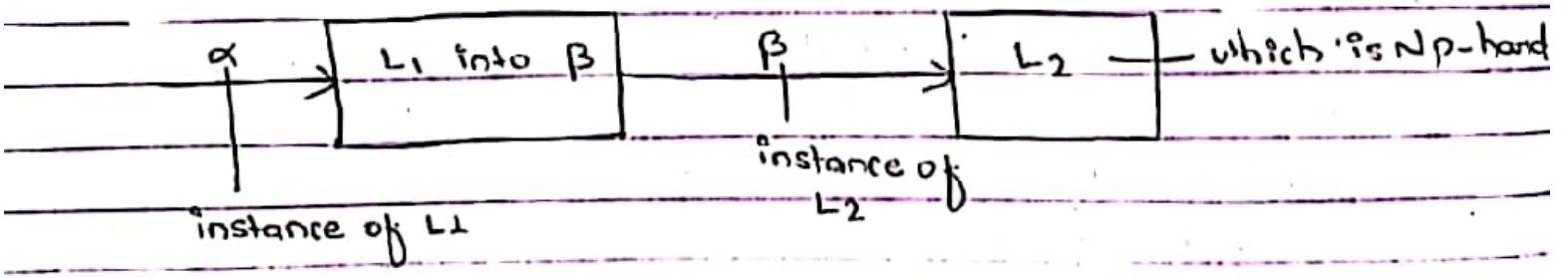
Step-1: Take a problem which is already known to be Np-hard. (L_2)

Step-2: Either transform L_1 into L_2 in polynomial time or vice-versa.

i.e., L_1 is reducible to L_2 in polynomial time.

Hence, answers of L_2 are also answers of L_1.

Step 3: So, L_1 is a Np-hard problem.



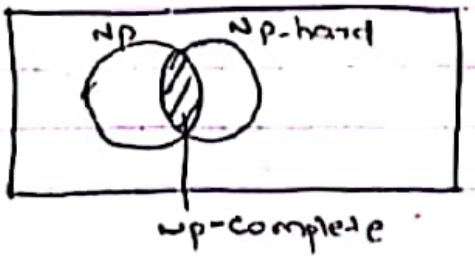
SAT is the first Np-hard problem known.

Np-complete:

It is also known that SAT problem is Np-problem.

Moreover, SAT is Np-problem as well as Np-hard problem

So, it is Np-complete problem.



* Proving a problem to be Np-complete;

Step i) Prove that it is Np-problem.

ii) Prove that it is Np-hard problem.

iii) Since, it is Np and Np-hard so, it is complete Np.

Proving a problem is Np-complete;

If L_1 is Np-complete, then L_2 is also Np-Complete if an instance of L_1 is reducible to a instance of L_2 or vice versa in polynomial time.

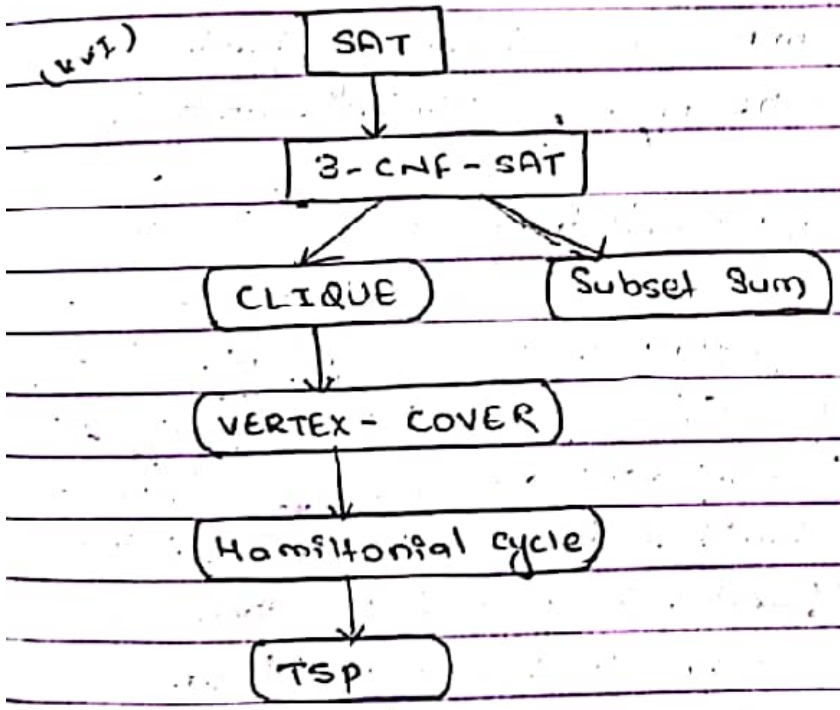


fig: The structure of Np-complete cycle / Proof

Chapter - 6
Branch and Bound

6.1 - Introduction :

Branch and bound is a systematic method for solving optimization problems. Branch and bound is a rather general optimization technique that applies where the greedy method and dynamic programming fail. However, it is much slower. Indeed, it often leads to exponential time complexities in the worst case.

Branch and bound requires two tools. The first one is a way of covering the feasible region by several smaller feasible sub-regions. This is called branching. Another tool is bounding, which is a fast way of finding upper and lower bounds for the optimal solution within a feasible sub-region.

The branch and bound technique explore the implicit graph and deals with the optimal solution to a given problem. In this technique, at each stage, we calculate the bound for a particular node and check whether this bound will be able to give the solution or not. If we find at any node the solution so obtained is appropriate but the remaining solution is not leading to a best case then we leave this node without exploring. In branch and bound terminology, a BFS like state space search will be called FIFO search and DFS like state space search will be called LIFO.

Solution is represented as state space tree and it is used for solving only minimization problem.

Example: [4 queens]

Let us see how a FIFO branch and bound algorithm would search the state space tree for the 4-queens problem.

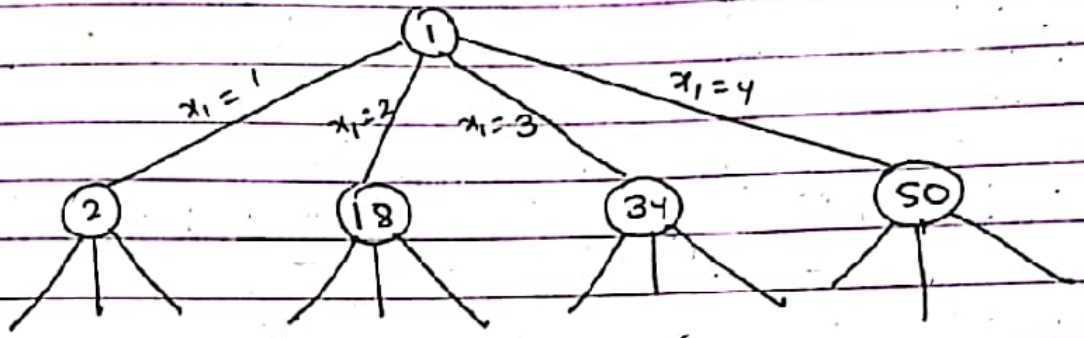


fig: Tree organization of the 4-queens problem

Initially, there is only one live node, node 1. This is the case when no queen has been placed in chessboard. Node 1 becomes E-node and nodes 2, 18, 34, 50 are generated. This represents a chessboard with queen 1 in row 1 and column 1, 2, 3 and 4 respectively. The live nodes are now 2, 18, 34 & 50 If nodes are generated in this order, next E-node is 2. It is expanded and nodes 3, 8, and 13 are generated Node 3 is immediately killed using the bounding function. Nodes ~~there~~ that are killed as a result of the bounding functions have a "B" under them.

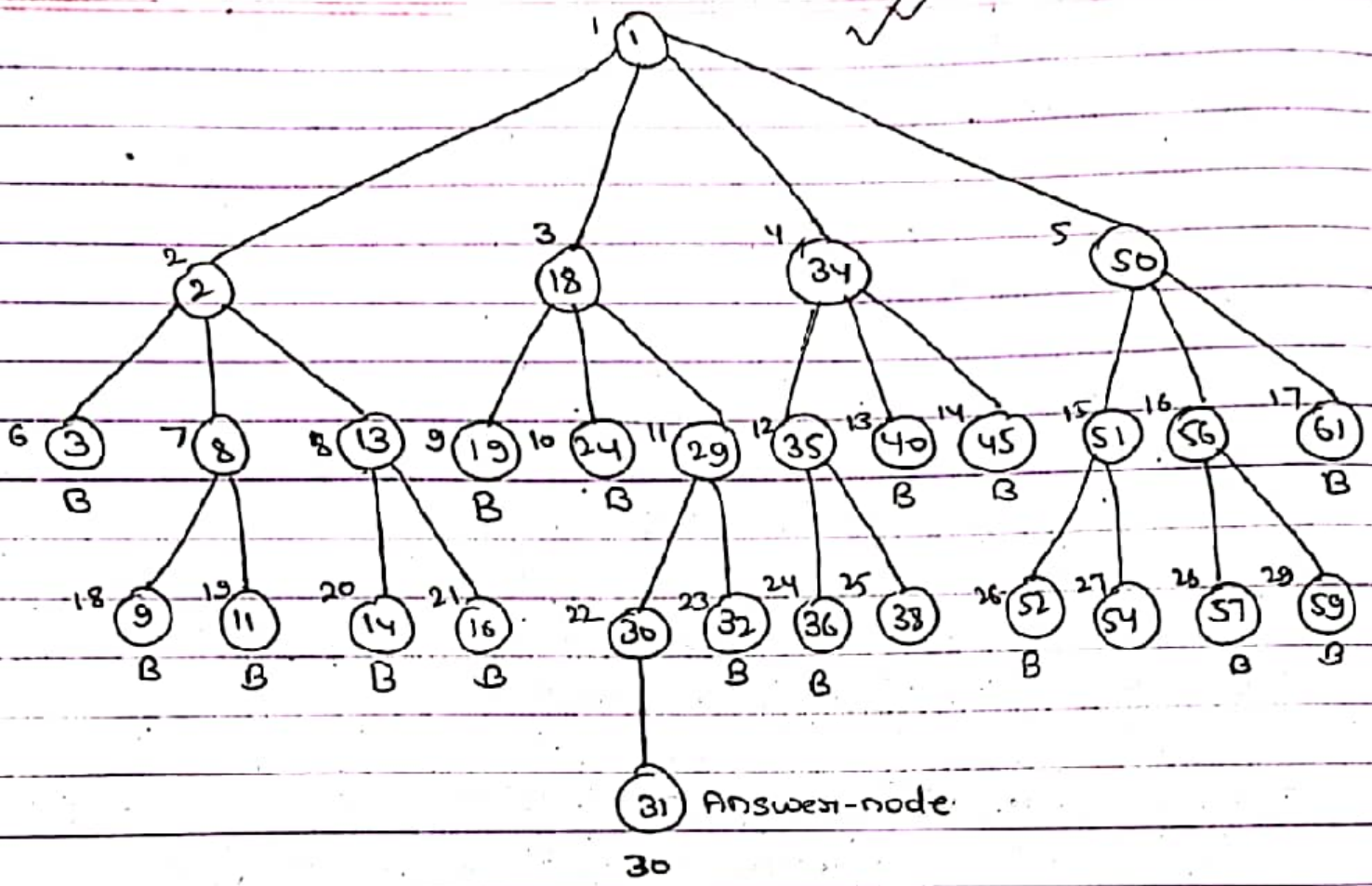


fig: portion of 4-queens state space tree generated by FIFO branch and bound.

6.2 0/1 knapsack problem:

In knapsack problem, we have to fill the knapsack of capacity W, with a given set of items $I_1, I_2 \cdots I_n$ having weights $w_1, w_2, \cdots w_n$ in such a manner that the total weight of the items should not exceed the knapsack capacity and the maxm possible value can be obtained.

By using branch and bound technique we have a bound that none of the items have total sum more than the capacity of knapsack and must give maximum possible value. The implicit tree for this problem is constructed

as a binary tree, where the left branch signifies the inclusion of item and the right branch signifies exclusion. A node in the state space tree contains three parts; the first part indicate the total weight of the item, the 2nd part indicates the value of the current item and the third part indicates the upper bound for the node.

wt.	value
ub	
node	

The upper bound ub of the node can be computed as:

$$ub = V + (W - w)(V_{i+1} / w_{i+1})$$

where V is the value of the current node
W is the total weight i.e. knapsack capacity.
w is the wt. of current node.

```
Algorithm UBound (cp, cw, k, m)
{
    b := cp ; c := cw
    for i := k+1 to n do
    {
        if (c + w[i] <= m) then
        {
            c := c + w[i] ; b := b - p[i];
        }
    }
    return b;
}
```

Example:

Solve the following instance of the knapsack problem by branch and bound algorithm.

Item (I)	Weight (w_i)	Value (v_i)	Value v_i/w_i
I_1	5	$ 50	10
I_2	8	$ 48	6
I_3	6	$ 30	5
I_4	4	$ 16	4

the knapsack capacity is 12

Soln:

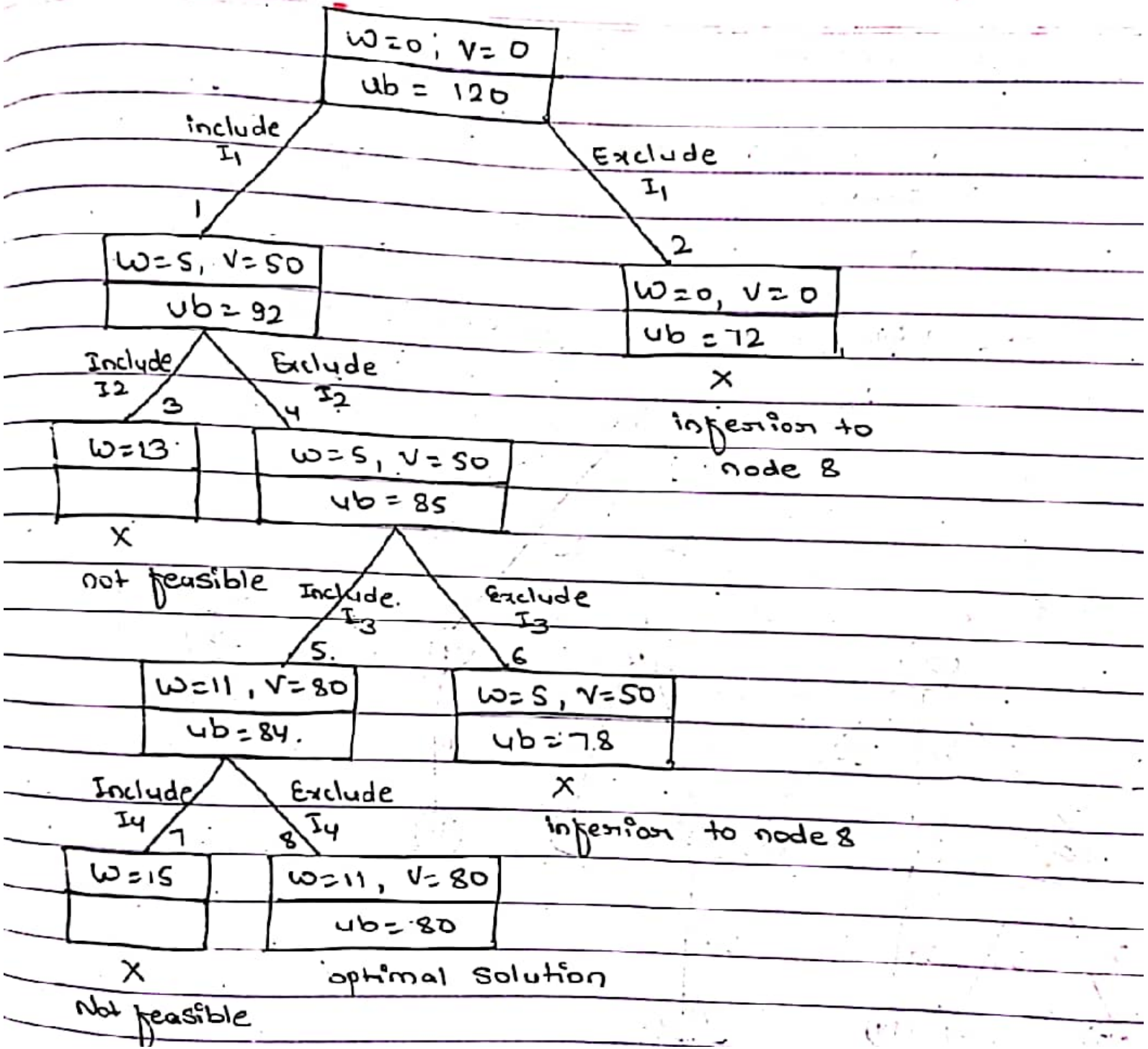
We arrange the table according to its descending value per weight ratio.

Now, we start with root node; the upper bound for the root can be computed as,

$$ub = v + (W - w)(v_{i+1}/w_{i+1})$$
$$= 0 + (12 - 0) \times 10$$
$$= 120$$

Now, we include item I_1, and exclude item I_1 which are indicated by the left branch and right branch respectively.

$$ub = v + (W - w)(v_{i+1}/w_{i+1})$$


w=0; v=0
ub = 120
include I1
Exclude I1
1
2
w=5, v=50
ub=92
w=0, v=0
ub=72
X
inferior to node 8
Include I2
Exclude I2
3
4
w=13
w=5, v=50
ub=85
X
not feasible
Include I3
Exclude I3
5
6
w=11, v=80
ub=84.
w=5, v=50
ub=78
X
inferior to node 8
Include I4
Exclude I4
7
8
w=15
w=11, v=80
ub=80
X
Not feasible
optimal Solution

Branch & Bound - BFS
Backtracking - DFS

Eg of branch and bound Job sequencing

Breath first search.

Jobs = {J1, J2, J3, J4}

P = {10, 5, 8, 3}

d = {1, 2, 1, 2}

BFS → Queue is used so it is called FIFO-BB

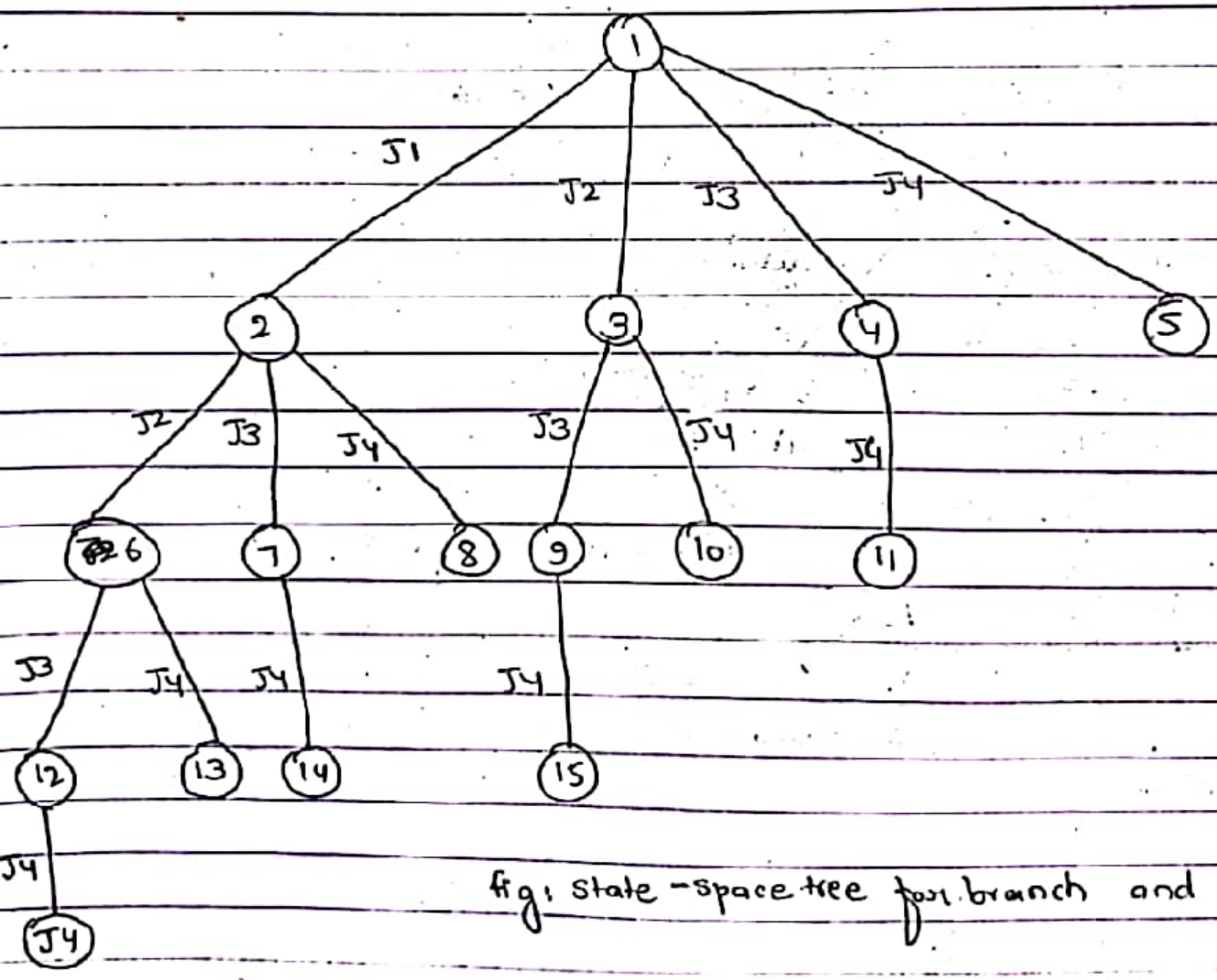


fig: State-space tree for branch and bound

Longest Common Subsequence (LCS)

Sequence B will be subsequence of A if it appears in same relative order, but not necessarily continuous.

A = a b c d e f g h

B = p q b r c e u f

b c e f

Here B is subsequence of A.

A = b e f g p q

B = e p r l m q

e p q

Q. Find the longest Common Subsequence (LCS) between abcdaf and acbcf.

Soln:

		a	b	c	d	a	f
Row	0	0	0	0	0	0	0
a	0	0+1	1	1	1	1	1
c	0	1	1	2	2	2	2
b	0	1	2	2	2	2	2
c	0	1	2	3	3	3	3
f	0	1	2	3	3	3	4

L C Sub-sequence = a b c f

length = 4

Least Common Subsequence (LCS)

Sequence B will be subsequence of A if it appears in same relative order, but not necessarily continuous

A = a b c d e f g h

B = p q b r c e u f

b c e f

Here B is subsequence of A.

A = b e f g p q

B = e p r l m q

e p q

Q. Find the longest Common Subsequence (LCS) between abcdaf and acbcf.

Soln:

		a	b	c	d	a	f
Row	0	0	0	0	0	0	0
a	0	0+1	1	1	1	1	1
c	0	1	1	1	2	2	2
b	0	1	2	2	2	2	2
c	0	1	2	3	3	3	3
f	0	1	2	3	3	3	4

L C Sub-sequence = a b c f

Length = 4

Q. Find the longest common subsequence (LCS) between "X1MJYAUZ" and "MZJAWXU".

Soln:

	col	X	M	J	Y	A	U	Z
Row	0	0	0	0	0	0	0	0
M	0	0	1	1	1	1	1	1
Z	0	0	1	1	1	1	1	2
J	0	0	1	2	2	2	2	2
A	0	0	1	2	2	3	3	3
W	0	0	1	2	2	3	3	3
X	0	1	1	2	2	3	3	3
U	0	1	1	2	2	3	4	4

Least common sub-sequence = MJAU

Length of least common sub-sequence = 4

Dynamic programming (optimization problems)

sequence of decisions

4.4 0/1 knapsack:

$m = 3$ $\quad P = \{1, 2, 5, 6\}$

$n = 4$ $\quad w = \{2, 3, 4, 5\}$

Soln:

$x_i = 0/1$ not in fraction i.e. either zero or one.

$\sum p_i x_i = \max$.

$\sum w_i x_i \leq m$

2^n = Solutions

Time Complexity $O(2^n)$

Tabulation method:

		V / weights									
		n	0	1	2	3	4	5	6	7	8	$=m$
P_i	w_i	0	0	0	0	0	0	0	0	0	0	→ objects
1	2	1	0	0	1	1	1	1	1	1	1	
2	3	← 2	0	0	1	1	2	3	3	3	3	
5	4	← 3	0	0	1	2	5	5	6	7	7	
6	5	4	0	0	1	2	5	6	6	7	8	

$V[i, w] = \max\{V[i-1, w], V[i-1, w-w_i] + P[i]\}$

(i ↓ row, w ↓ wt)

$V[4,1] = \max\{V[3,1], V[3, 1-5] + 6\}$

$= \max\{V[3,1], V[3,-4] + 6\}$

$= 0$

$V[4,5] = \max\{V[3,5], V[3, 5-5] + 6\}$

$= \max\{5, 0+6\}$

$= 6$

$V[4,6] = \max\{V[3,6], V[3, 6-5]+6\}$

$= \max\{6, 0+6\}$

$= 6$

$V[4,7] = \max\{V[3,7], V[3, 7-5]+6\}$

$= \max\{7, 7\}$

$= 7$

$V[4,8] = \max\{V[3,8], V[3, 8-5]+6\}$

$= \max\{7, 8\}$

$= 8$

$(x_1 \quad x_2 \quad x_3 \quad x_4)$

$(0 \quad 1 \quad 0 \quad 1)$

$P = 8 - 6 - 2 - 2 = 0$

Total profit $= \sum p_i x_i$

$= p_1x_1 + p_2x_2 + p_3x_3 + p_4x_4$

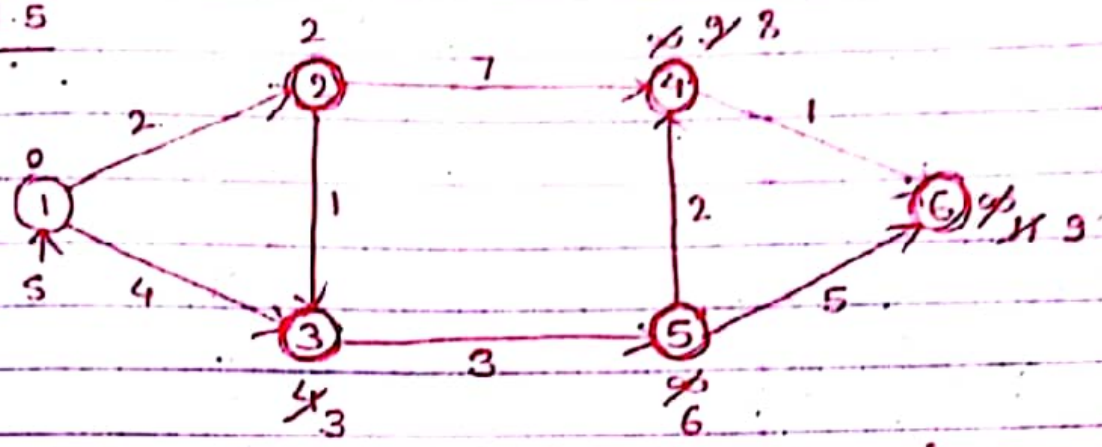
$= 0 + 1 \times 2 + 0 + 6 \times 1$

$= 2 + 6$

$= 8$

Dijkstra Algorithm (Greedy method)

3.5



Relaxation:

if $(d[u] + c(u,v) < d[v])$

$d[v] = d[u] + c(u,v)$

Smallest so select it first

1st → {2, 4, ∞, ∞, ∞}

2nd step = {3, ∞, 9, ∞}

3rd step = {6, 9, ∞}

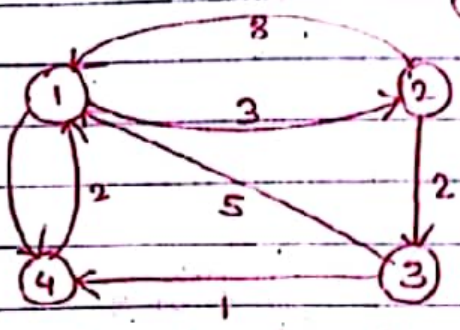
4th step = {8, 11}

5th = 9

v	d[v]
2	2
3	3
4	8
5	6
6	9

Time complexity = $n \times n = \theta(n^2)$ → worst case

1.3 All pairs shortest path (Dynamic programming)



A^0 =	1	2	3	4
1	0	3	∞	7
2	8	0	2	∞
3	5	∞	0	1
4	2	∞	∞	0

Complexity : $O(n^3)$

Soln :

A^1 =	1	2	3	4
1	0	3	∞	7
2	8	0	2	15
3	5	8	0	1
4	2	5	∞	0

Similarly, A^2 =

0	3	5	7
8	0	2	15
5	8	0	1
2	5	7	0

$A^0[2,3]$ - $A^0[2,1] + A^0[1,3]$

$2 < 8 + \infty$

$A^0[2,4]$ $A^0[2,1] + A^0[1,4]$

$\infty > 8 + 7$

$A^0[3,2]$ $A^0[3,1] + A^0[1,2]$

$\infty > 5 + 3$

$A^0[3,4]$ $A^0[3,1] + A^0[1,4]$

$1 < 5 + 7$

$A^0[4,2]$ $A^0[4,1] + A^0[1,2]$

$\infty > 2 + 3$

$A^0[4,3]$ $A^0[4,1] + A^0[1,3]$

$0 < 2 + \infty$.

A^3 =

0	3	5	6
7	0	3	3
5	8	0	1
2	5	7	0

A^4 =

0	3	5	6
5	0	2	3
3	6	0	1
2	5	7	0