UNIT-2
Dynamic Programming.

Dynamic Programming :- It is a designing technique that solves the prbm by partitioning the problem into destroying subproblems, solving the subproblem recursively & then combining the subproblems solution to solve the original prbm

This technique is applied to optimization problems that have many possible sol^n. The task is to find a sol^n i.e globally optimal

Elements of dynamic programming :→ While solving a dynamic programming problem algorithm, the following sequence of elements is follow :-

1) Characterized the structure of an optimal sol^n :→ In this, all the possible sol^n of the problem are considered & are checked.

2) Recursively define the value of an optimal sol^n :- In this, the possible sol^n strategy is proposed for the given prbm.

3) Compute the value of an optimal sol^n in botten-up fashion :- In this, the proposed sol^n is checked for all the possible sol^n.

4) Construct an optimal sol^n from computed information :- In this step, the best sol^n out of all the possible sol^n is chosen to be the final global optimal sol^n.

Examples:- LCS:→ Least Common Subsequence
MCM:- Matrix Chain Multiplication.

* LCS:→

Subsequence:- A subsequence of a given sequence is ~~a list of~~ that consist of one or more ~~elements~~ of the sequence in the same order.

LCS:- In LCS problem, we are given two sequences
$X = \{x_1, x_2, x_3 \dots\dots x_m\}$
$Y = \{y_1, y_2, y_3 \dots\dots y_n\}$.

The aim of this prob, is to find a maximum length common subsequence of x & y. This prob is applicable in DNA matching in which we need to find out how similar are two strands of DNA or how closely related two people are?

Algorithm of LCS:-

Algorithm LCS_length(X, Y)

1. m := length[X] , n := length[Y] 1
2. Let b[1....m, 1....n] & c[0....m, 0....n] are two new tables
3. for {i = 1 to m] m+1
4. do c[i, 0] = 0]
5. for j = 0 to n] n+2
6. do c[0, j] = 0]
7. for i = 1 to m m+1
8. for j = 1 to n m(n+1)

```
9.  If (xi = yj)
10. c[i,j] := c[i-1, j-1]
11  b[i,j] = "↖"
12  else if c[i-1,j] ≥ [i,j-1]
13  c[i,j] = c[i-1,j]
14  b[i,j] = "↑"
15  else c[i,j] := c[i,j-1]
16  b[i,j] = "←"
17  return c & b
```

mn

$T(n) = O(mn)$

```
PRINT-LCS (b,X,i,j)

1. If (i=0) or (j=0)
2  return
3  If b[i,j] = "↖"
4  PRINT LCS (b, x, i-1, j-1)
5  print xj
6  else if b[i,j] := "↑"
7  PRINT LCS (b, x, i-1, j)
8  else PRINT-LCS (b, x, i, j-1)
```

In this algorithm, x & y are two sequences of length m & n respectively. C is the table i.e used to maintain the length of LCS & b is the table that will maintain the direction of arrows.

X = A, B, C, B, D, A, B.

Y = B, D, C, A, B, A.

		(0)	(1)	(2)	(3)	(4)	(5)	(6)
		Y	B	D	C	A	B	A
(0)	Yi	0	0	0	0+1	0	0+1	0
(1)	A	0↑	0↑	0↑	0↑	1↑	1←	1↑
(2)	B	0	1↖	1←	1←	1↑	2↑	2←
(3)	C	0	1↑	1↑	2↖	2←	2↑	2↑
(4)	B	0	1↖	1↑	2↑	2↑	3↑	3←
(5)	D	0	1↑	2↖	2↑	2↑	3↑	3↑
(6)	A	0	1↑	2↑	2↑	3↑	3↑	4↖
(7)	B	0	1↖	2↑	2↑	3↑	4↑	4↑ → Length of LCS

BCBA

0,1
1,0

X = 1,0,0,1,0,1,0,1

Y = 0,1,0,1,1,0,1,1,0

UNIT-II

DAA - Lecture

MCM:- In MCM problem, a sequence of n matrix ($A_1, A_2 \ldots A_n$) for ($i = 1, 2, 3 \ldots n$) is given where each matrix A[i] has the dimension $A_i : p_{i-1} * p_i$

We wish to fully parenthesized the product $A_1, A_2, \ldots A_n$ in such a way so that no. of scaler multiplication (intermidate) are minimized

Consider a product of chain (A_1, A_2, A_3) such that :-

$A_1 = 10 * 100$ $\quad A_2 = 100 \times 5$ $\quad A_3 : 5 * 50$

$(A_1 A_2) A_3 \qquad A_1 (A_2 A_3)$

$(A_1 A_2)_{NSM} = 10 \times 1000 \times 5 = 5000$

$(A_1 A_2) : 10 * 5 \quad A_3 : 5 * 50$

$((A_1 A_2) A_3)_{NSM} = 10 * 5 * 50 = 2500$

$\underline{7500} \rightarrow$ Total no. of Scaler multiplication

$(A_2 A_3)_{NSM} = 100 * 5 * 50 = 25000$

$(A_2 A_3) : 100 * 50 \quad A_1 : 10 * 100$

$(A_1 (A_2 A_3)) = 10 * 100 * 50 = 50000$

$\underline{75000} \rightarrow$ Total NSM

Steps of MCM:-

Step 1:→ The structure of an optimal parantheziation:- The first step is to find the optimum structure & when use it to constract an optimal sol^n which is optima

parenthesization as it will have major impact on the cost of evaluating the product.

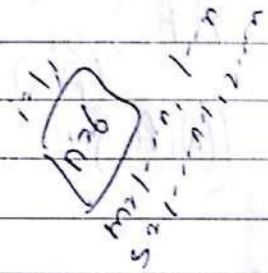
Step2:- A recursive sol":- The cost of an optimal solution can be recursively defined in terms of optimal sol" with the following recursive definition:-

$$m[i,j] \begin{cases} 0 & \text{if } i=j \\ \min_{i \le k < j} \{ m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \} & i<j \end{cases}$$

Step 3:- Computing the optimal cost:- A recursive algorithm will be defined to compute the minimum cost $m[1,n]$ for multiplying a set of matrices $[A_1, A_2 \ldots A_n]$

Algorithm of MCM

Algorithm Matrix-chain Order (p)



```
1  n := length[p] - 1
2  Let m[1...n, 1...n] & s(1...n-1, 2...n)
       be new tables
3  for i := 1 to n
4  m[i, i] = 0
5  for l := 2 to n // l is the chain length
6  for i = 1 to n-l+1
7     j = i + l - 1
8     m[i,j] = ∞
9     for k = i to j - 1
10    q := m[i, k] + m[k+1, j] + p_{i-1} p_k p_j
```

11 if $q < m[i,j]$
12 $m[i,j] := q$
13 $s[i,j] := k$
14 return m & s.

$A_1 : 30 \times 35$ (p_0 p_1)
$A_2 : 35 \times 15$ (p_1 p_2)
$A_3 : 15 \times 5$ (p_2 p_3)
$A_4 : 5 \times 10$ (p_3 p_4)
$A_5 : 10 \times 20$ (p_4 p_5)
$A_6 : 20 \times 25$ (p_5 p_6)

Here p: is the order of matrix
m: is the table to maintain the scaler multiplication
s: " " that is used to maintain the value of k

$l=1$ $n=6$ $i \leftarrow 1$ to 6.
$l=2$
$i \leftarrow 1$ to $6-2+1 = 1$ to 5

(i) $i=1, j=1+2-1=2, k=1$ to $2-1=1$
$q = m[1,1] + m[2,2] + p_0 p_1 p_2$
$= 0+0+30\times35\times15 = 15,750$

(ii) $i=2, j=2+2-1=3$ $k=2$ to $3-1=2$
$q = m[2,2] + m[3,3] + p_1 p_2 p_3$
$= 2625$

(iii) $i=3, j=3+2-1=4$
$k=3$
$q \leftarrow m[3,3]+m[4,4]+p_2 p_3 p_7$
$= 750$

(iv) $i=4, j=5, k=4$
$q \leftarrow m[4,4]+m[5,5]+p_3 p_4 p_5$
1000

(v) $i=5, j=6, k=5$
$q \leftarrow m[5,5]+m[6,6]+p_4 p_5 p_6 = 5000$

Step 2 $l=3$
$i = 1 \text{ to } n-l+1 = 1 \text{ to } 6-3+1 = 1 \text{ to } 4$

(i) $i=1, j=1+3-1=3, k=1 \text{ to } 3-1$
$= 1 \text{ to } 2$

(a) $i=1, j=3, k=2$
$q \leftarrow m[1,1]+m[2,3]+p_0 p_1 p_3 = 7875$

(b) $i=1, j=3, k=2$
$q \leftarrow m[1,2]+m[3,3]+p_0 p_2 p_3 = 18000$

(ii) $i=2, j=4, k=2 \text{ to } 4-1 \Rightarrow 2, 3$

(a) $i=2, j=4, k=2$
$q \leftarrow m[2,2]+m[3,4]+p_1 p_2 p_4 = 2625+750+5250 = 8625$

(b) $i=2, j=4, k=3$
$q \leftarrow m[2,3]+m[4,4]+p_1 p_4 p_3 = 2625+1750$
$= 4375$

(iii) $i=3, j=5, k=3$ to 4

(a) $i=3, j=5, k=3$

$q \leftarrow m[3,3] + m[4,5] + p_2 p_3 p_5 = 2500$

(b) $i=3, j=5, k=4$

$q \leftarrow m[3,4] + m[5,5] + p_2 p_4 p_5 = 3750$

(iv) $i=4, j=6, k=4$ to 5

(a) $i=4, j=6, k=4$

$q \leftarrow m[4,4] + m[5,6] + p_3 p_4 p_6 = 6250$

(b) $i=4, j=6, k=5$

$q \leftarrow m[4,5] + m[6,6] + p_3 p_5 p_6 = 3500$

(v) $i=1, j=4, k=1$ to 3

(a) $i=1, j=4, k=1$

$q \leftarrow m[1,1] + m[2,4] + p_0 p_1 p_4$

$q \leftarrow 0 + 4375 + 30 * 35 * 10$

$q \leftarrow 14875$

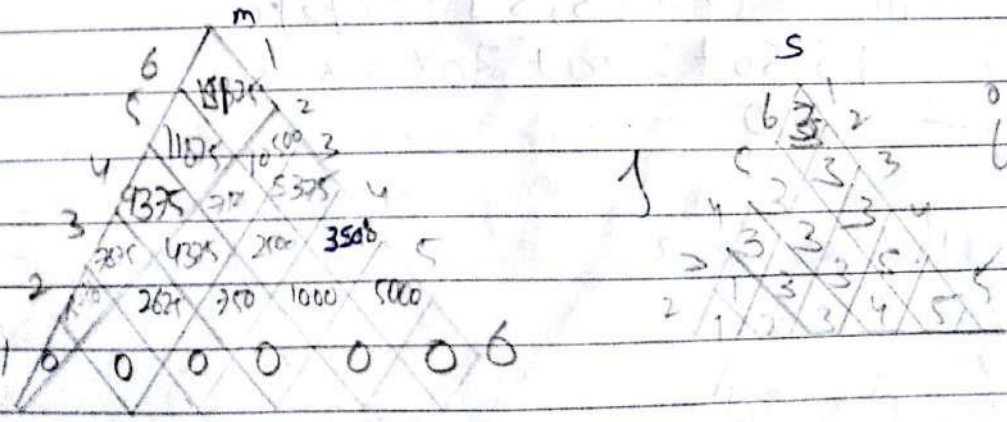
(b) $i=1, j=4, k=2$

$q \leftarrow m[1,2] + m[3,4] + p_0 p_2 p_4$

$q \leftarrow 15750 + 750 + 30 * 15 * 10$

$q \leftarrow 21000$

(c) $q \leftarrow 9375$



i=2 j=5 k=2 to 4

(a) i=2, j=5, k=2
q ← m[2,2] + m[3,5] + P1P2P5
q ←
q ← 13000

(b) i=2, j=5, k=3
q ← m[2,3] + m[4,5] + P1P3P5
q ←
q ← 7125

(c) i=2, j=5, R=4
q ← m[2,4] + m[5,5] + P1P4P5
q ← 11375

S-4 l=5 i ← 1 to 6-5+1 = 1 to 2, j=5, k=1 to 4

(a) i=1, j=5, k=1
q ← m[1,1] + m[2,5] + P0P1P5
q ← 0 + 7125 + 30*35+20
q ← 28125

(b) i=1, j=5, k=2
q ← m[1,2] + m[3,5] + P0P2P5
q ← 15750 + 2500 + 30*15*20
q ← 27250

(c) i=1, j=5, k=3
q ←
q ←
q ←

do i=1, j=5, k=4
q ←
q ←
q ←

s=5 l=6 i ← 1 to 6-6+1 = 1 to 1
= r=6 k = 1 to 5

i=1, j=6, k=1
q ← 36750

i=2, j=6, k=2
q ← 32375

i=1, j=6, k=3
q ← 15125

i=1, j=6, k=4
q ← 21875

i=1, j=6, k=5
q ← 26875 = m[1,5] + m[6,6] + $p_0 p_5 p_6$
= 11875 + 0 + 30*20*25
= 26875

Step 4:→ Constructing an optimal sol^n:- We have determined the optimal no. of scalar multiplications that are required to compute a Matrix Chain product.

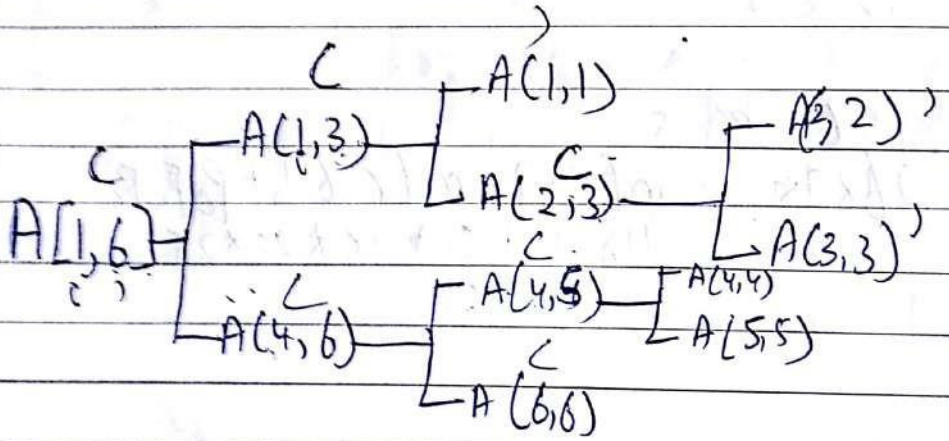
Each entry of variable k in the table s[i,j]

determines the optimal parantheization of the matrices which can be obtained with the help of following algorithm:-

PRINT-OPTIMAL-PARENTHESIZATION (S, i, j)

1. if (i=j)
2. Print "A_i".
3. else print "("
4. PRINT-OPTIMAL-PARENTHESIZATION (S, i, S[i,j])
5. " " " (S, S[i,j]+1, j)
6. print ")"

Here i = the ~~order~~ value of first matrix.
j = " " " last "
& S = is the table in which values of k are stored.



$((A_1(A_2A_3))((A_4A_5)A_6))$

$((A_1(A_2A_3))((A_4A_5)A_6))$

UNIT-II

DAA - Lecture.

Greedy Algorithms :→ These algorithms are those algorithm that make choice on the basis of a sol^n that looks best at that particular moment. The local optimal choice is made in such a way so that it will lead to global optimal sol^n. However, it may not be possible in every case.

Elements of Greedy Algorithm :- A greedy algorithm obtains an optical sol^n to a problem by making a sequence of choices. At each decision point the algorithm makes choice on the basis of best available option which may or may not lead to an optimal sol^n globally. The following elements define the structure of greedy algorithm :-

1. Greedy Choice Property :- It is the main step while defining a greedy algorithm bcoz the best sol^n is chosen at the moment & then the subproblem sol^n are solved. Basically, in this algorithm, we proceed in a top-down fashion in which the greedy choice property is selected first & then the sol^n are computed.

2) Optimal substructure :- A prbm is said to have optimal substructure if an optimal sol^n to the prbm is also an optimal sol^n to its subproblem.

~~Huffman Cod~~

Methods in GA :- ① Huffman Codes
② Knapsack Problem

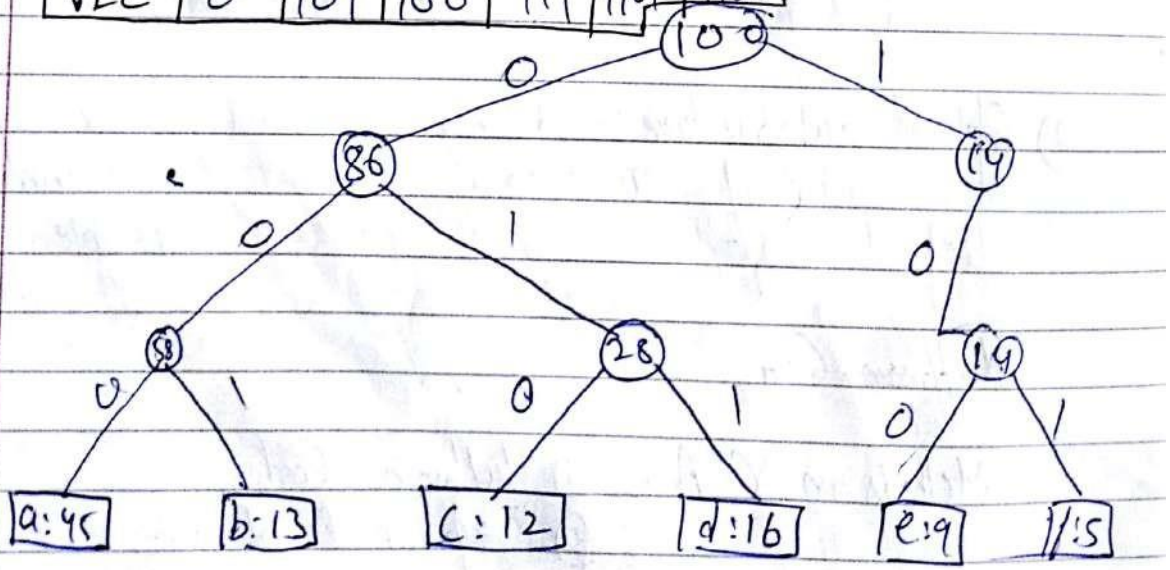
(3) Activity Selection Problem
(4) Travelling Salesman "
(5) Task Scheduling "

Huffman Codes:- These codes are used to compress the data from 20-90% depending on characteristics of data that are being compressed.

Here, data is consider to be a sequence of characters. Data is represented in the form of a table that describes its frequency i.e how many times the character is used. There are two ways to encode the data:-

1) Fixed-length Code (FLC):- Each character is represented using a unique binary string which is known as its codeword. Their is a file of one-lakh (1,00,000) characters that we wish to store compactly.

characters	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
FLC	000	001	010	011	100	101
VLC	0	101	100	111	1101	1100



Total storage space :- $(45*3 + 13*3 + 12*3 + 16*3 + 9*3 + 5*3) * 1000$

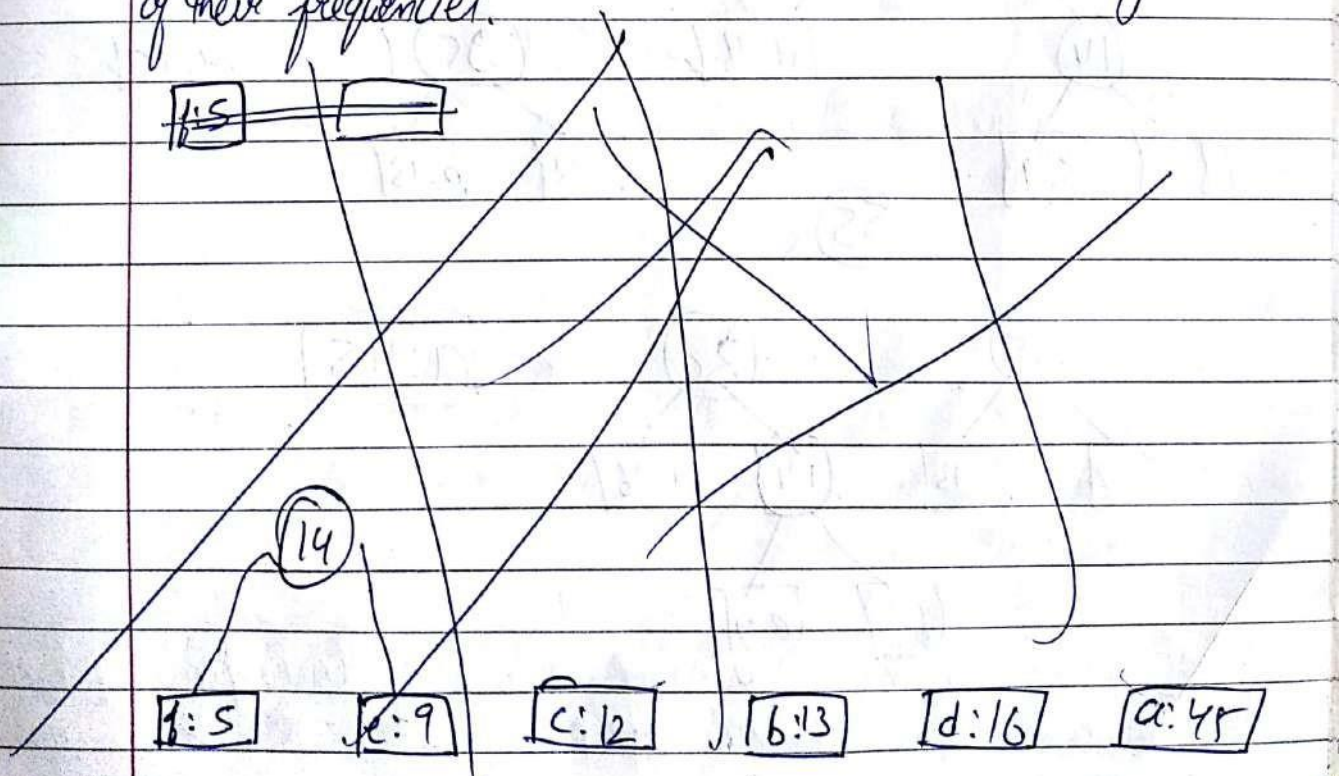
$= (100*3) * 1000 = 3,00,000$ bytes.

Variable-Length Code :- Since, in FLC, all the characters were represented by equal no. of bits. So, the storage space is considerably high. Huffman gave the concept of V-L C in which the characters are represented or stored depending on their frequency i.e the frequent characters were given short code-word & infrequent characters were given long code word i.e.

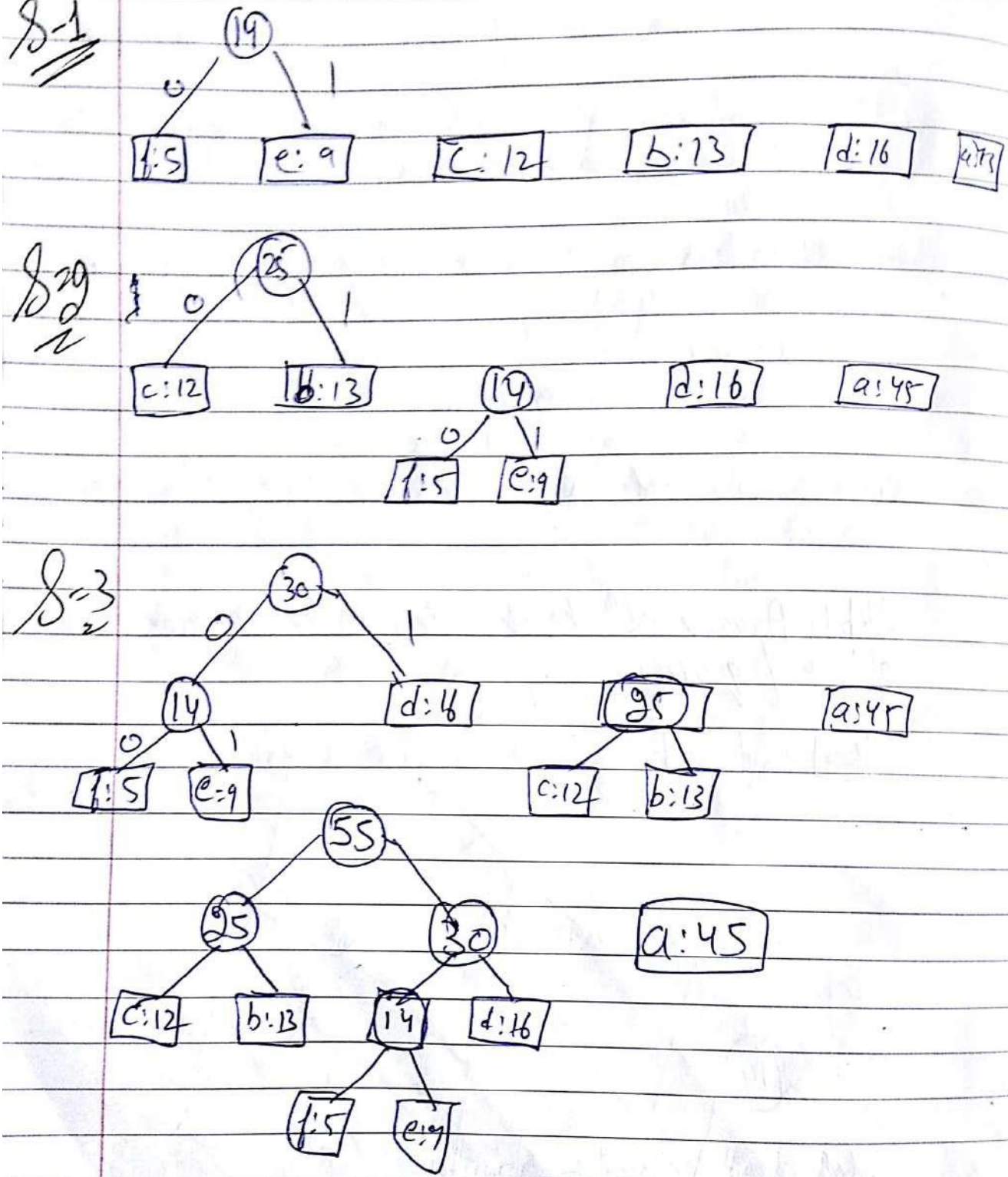
$$\text{Frequency} \propto \frac{1}{\text{Prefix code}}$$

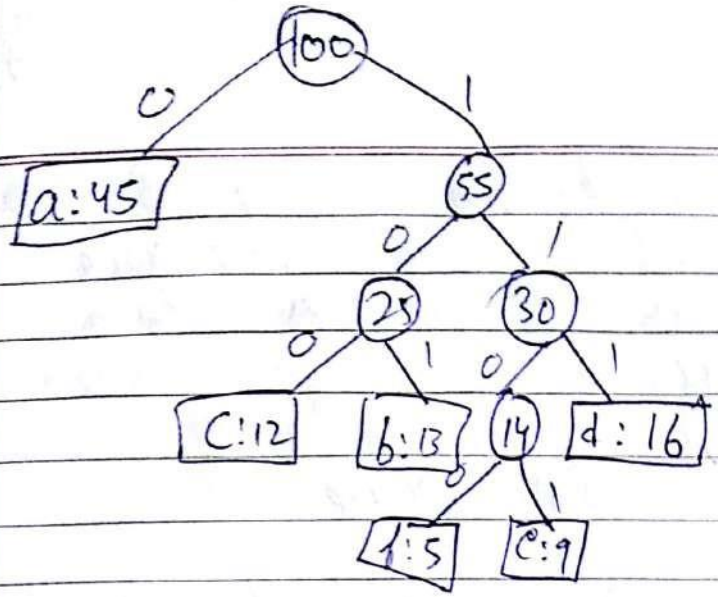
where Prefix code represents the encoding of respective characters

Step 1 :- Arrange all the characters in the increasing order of their frequencies.



Step 3:- Combine the sum of 1st two nodes & rearrange all the characters in increasing order. However, the leaf node will be ~~compared~~ compared with root node for comparison.





Total no. of bits req. using VLC.

$(45*1 + 13*3 + 12*3 + 16*3 + 9*4 + 5*4) * 1000$

2,24,000 bits

Algorithm Huffman :→

Algorithm Huffman (C)

1. n := |C|
2. Q = C
3. for i := 1 to n-1
4. { Allocate a new node 'z'
5. Left[z] = x = EXTRACT-MIN (Q)
6. RIGHT[z] = y = " " "
7. freq[z] = freq[x] + freq[y]
8. INSERT (Q, z) }
9. return EXTRACT-MIN (Q) // return the root of tree

In this algorithm, C is a sequence of characters, n is the length of the sequence.
EXTRACT-MIN :- is the procedure to retrieve the min

value from the sequence of characters. & insert is the procedure i.e used to insert the value in the queue. Its running time is $O(n \log n)$ where n is no. of characters & $\log n$ is the height of tree.

UNIT-II

DAA-Lecture

Knapsack Problem:- This greedy algorithm is based on the following principles:→

We are given a knapsack or a bag & n no. of objects. Object i has weight w_i & profit p_i & the capacity of knapsack is 'm' units. If a fraction x_i such that $0 \leq x_i \leq 1$ is placed in knapsack when a profit of $p_i x_i$ units is earned. So, the objective is to fill up the knapsack in such a way that will maximize the total profit but the weight of all the objects can be equal to almost m. An knapsack prbm can be calculated as:-

$$\text{Maximize} \sum_i p_i x_i$$

$$\text{Subject to} \sum_i w_i x_i \leq m$$

$$\text{where } 0 \leq x_i \leq 1 \text{ & } 1 \leq i \leq n$$

Knapsack prbm is of 2 types:-

1) Fractional KP.
2) Zero/One KP.

Fractional→ Example:- Consider the following instance of the knapsack prbm where $n=3$, $m=20$, $(p_1 p_2 p_3) = (25, 24, 15)$ & (w_1, w_2, w_3)

$= (15, 15, 10)$ ③

1. Greedy for maximum profit:-

x_1	x_2	x_3
1	$\frac{2}{15}$	0

fraction of Hrshiya which is Pizza

$\sum w_i x_i = 18*1 + 15*\frac{2}{15} = \boxed{20}$

$\sum p_i x_i \quad 25*1 + 24*\frac{2}{15} = 25 + \frac{48}{15} = 25 + 3.2 = 28.2$

2) Greedy for minimum weight:-

x_1	x_2	x_3
0	$\frac{10}{15}$	1

$\sum w_i x_i = 10*1 + 15*\frac{10}{15} = 20$

$\sum p_i x_i = 15*1 + 24*\frac{10}{15}$

$= 15 + \frac{240}{15}$

$= 15 + 16 = 31$

3) Greedy for profit per unit weight :-

$\frac{p_1}{w_1} = \frac{25}{18} = 1.3 \qquad \frac{p_2}{w_2} = \frac{24}{15} = 1.6 \qquad \frac{p_3}{w_3} = \frac{15}{10} = 1.5$

x_1	x_2	x_3
0	1	5/10

$\sum w_i x_i = 15*1 + 10*\frac{5}{10} = 20$

$\sum p_i x_i = 24*1 + 15*\frac{5}{10} = 24 + 7.5 = 31.5$

Zero/One KP:-

1. Greedy for maximum profit :-

x_1	x_2	x_3
1	0	0

$\sum w_i x_i = 18 * 1 = 18$

$\sum p_i x_i = 25 * 1 = 25$

2. Greedy for minimum weight :-

x_1	x_2	x_3
0	0	1

$\sum w_i x_i = 10 * 1 = 10$

$\sum p_i x_i = 15 * 1 = 15$

3. Greedy for profit per unit weight :-

$\frac{p_1}{w_1} = 1.3$ $\quad \frac{p_2}{w_2} = 1.6$ $\quad \frac{p_3}{w_3} = 1.5$

x_1	x_2	x_3
0	1	0

$\sum w_i x_i = 15 * 1 = 15$

$\sum p_i x_i = 24 * 1 = 24$

Algorithm for fractional knapsack problem:-

Algorithm Fractional Greedy Knapsack (m, n)

```
{
1. for i=1 to n
2. do x[i] = 0.0   // initialize 'x'.
3.  U = m
4. for i:= 1 to n do
5. {
6.  If (w[i] > U) then break;
7.  x[i] = 1.0, U = U - w[i];
8. }
9. If (i<n) then x[i]:= U/w[i];
10 }
```

In this algorithm;→

m = is the capacity of the knapsack
n = total no. of objects.
x[i] = is the solution vector.
p[1:n] & w[1:n] are profit & weight vector respectively which are ordered in such a way, so that

$$\frac{p[i]}{w[i]} \geq \frac{p[i+1]}{w[i+1]} \geq \frac{p[i+2]}{w[i+2]} + \text{-----}$$

~~Order~~ Running Time $O(n)$.

In fractional knapsack problem, the object may be picked in fraction that can lead to the filling of the knapsack to its fullest. while in 0/1 KP, the entire object either needs to be pick or left. i.e value of x[i] can be ~~given~~ either 0 or 1 & the knapsack may not be filled completely

Ques: $n=7 \quad m=15$

$(P_1, P_2 \dots P_7) = (10, 5, 15, 7, 6, 18, 3)$

$(w_1, w_2 \dots w_7) = (2, 3, 5, 7, 1, 4, 1)$

UNIT-II

DAA - Lecture

Activity Selection Problem :-

Algorithm Activity - Selection (s, f)

```
1  n := length [s]
2  A = {a_1}
3  k := 1
4  for m := 2 to n                    Θ(n)
5  If (s[m] ≥ f[k])
6  A := A ∪ {a_m}
7  k := m
8  return A
```

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

This greedy algorithm is based on the selection of maximum size subset of mutually compatible activities.

We are having a set S of n activities from $S = \{a_1, \dots a_n\}$ that wish to use a common resource. Each activity a_i has a start time s_i & finish time f_i.

where $0 \leq s_i < f_i < \infty$. If selected activity a_i takes place, the selected resource is occupied for $\{s_i, f_i\}$ interval.

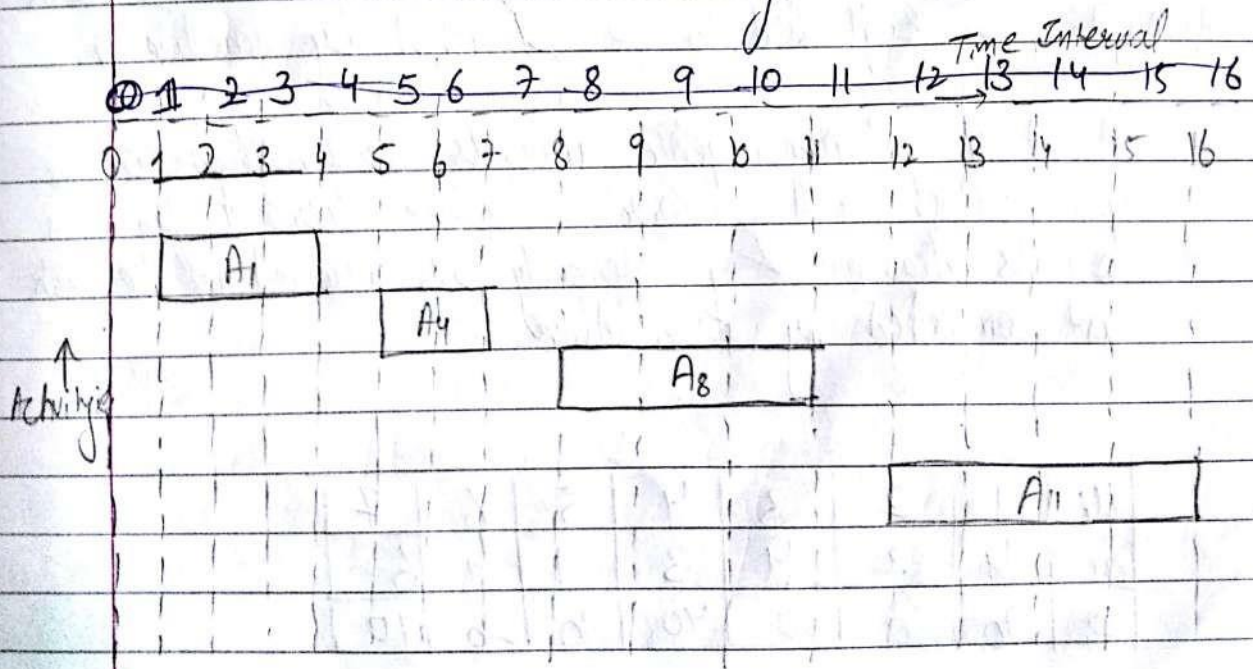
Activities a_i & a_j are said to be compatible if the interval $\{s_i, f_i\}$ & $\{s_j, f_j\}$ do not overlap i.e if :-

$s_i \geq f_j$
$s_j \geq f_i$.

The following is the set of activities which are arranged in increasing order of finishing time. i.e

$$f_1 \leq f_2 \leq f_3 \cdots\cdots f_n$$

In this algorithm S is the starting time.
f " " finishing "
A " " set of given activities.
A " " " " mutually compatible activities
a " " activity.



$A = \{A_1, A_4, A_8, A_{11}\}$

Task Scheduling Problem:- This greedy algorithm is based upon the principle of minimizing the penalty that will incurr/chargeable if the scheduled task is not completed on time. In this algorithm, we have to optimally scheduled the unit time task on a single processor where each task has a deadline along with a penalty to be paid if the task misses its deadline.

A unit time task is a job or a program to be run on a computer that requires exactly one unit of time to complete. The problem of scheduling unit time task with deadline & penalty for a single processor as following inputs:-

1) A set of $S = \{a_1, a_2 \dots a_n\}$ of 'n' unit time tasks.

2) A set of 'n' integer deadlines $d_1, d_2 \dots d_n$ where each d_i satisfies the following condition & task a_i is supposed to be finished by deadline d_i. $(1 \le d_i \le n)$

A set of 'n' non-negative weights or penalties $(w_1, w_2 \dots w_n)$ which will incurr if a task does not finishes by its deadline & no penalty is incurred if the task is completed by its deadline.

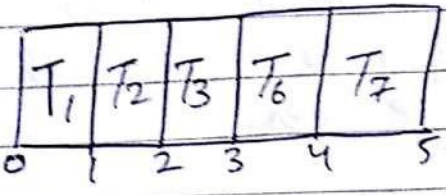
a_i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

If two task are having the same deadline, then the task will maximum penalty is chosen first. Similarly, if two task have same penalty, then the one having minimum deadline is chosen first.

Page No. 79
Date :

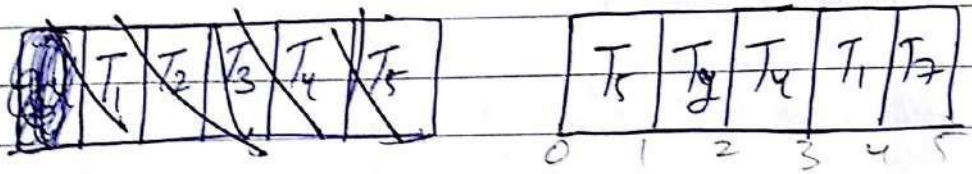
Find the optimal sol^n for the following task for given penalty & deadline where all these task have unit time & have to be run on a single processor having 5 time slots. & the slots starts from 0.

1. Greedy for maximum penalty:- In this approach, the task having maximum penalty are chosen at priority order to have those task that have minimum penalty.



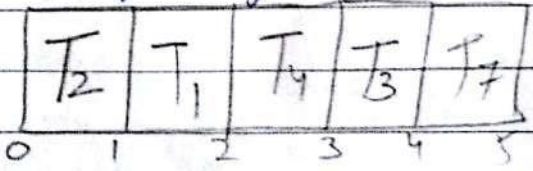
Penalty incurred = $T_4 + T_5$
$= 40 + 30 = 70$

2. Greedy for minimum deadline:- In this approach, the task are chosen on the basis of minimum deadline, so that the crucial task have minimum deadline could be executed at the earliest.



Penalty incurred = $T_3 + T_6 = 50 + 20 = 70$

3. Greedy for maximum penalty & minimum deadline:- In this approach, the task is chosen on the basis of both the criteria i.e maximum penalty & minimum deadline This approach generally provides optimal sol^n.



Penalty incurred:
$T_5 + T_6 = 30 + 20 = 7$

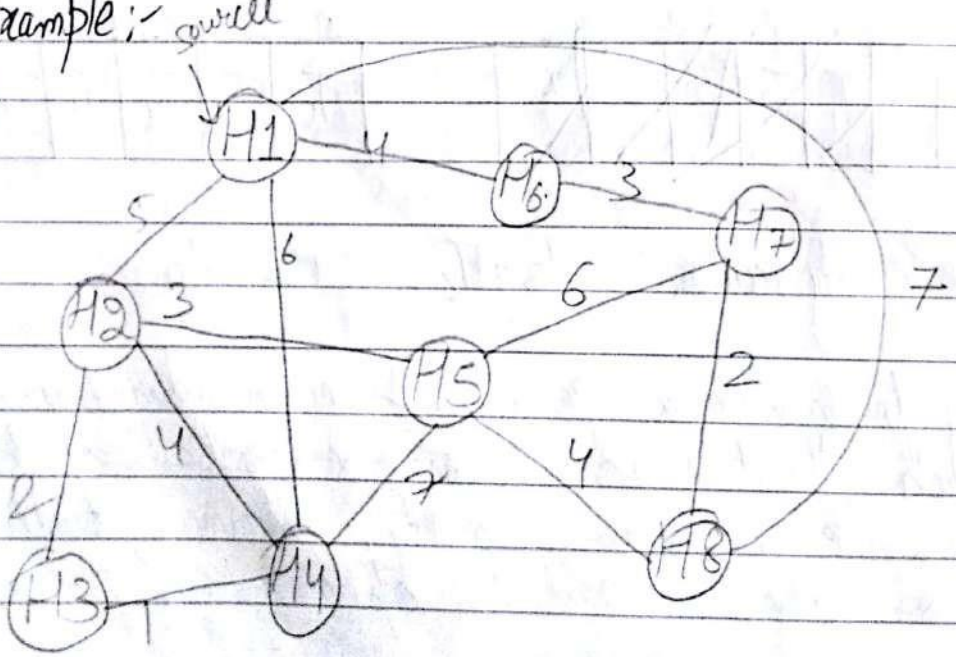
24137
12437

DAA-Lecture

Travelling Salesman Problem:- In this prbm, a salesman need to visit 'n' cities in such a way that all the cities must be visited exactly once & in the end, he has to return to the city from where the travelling was started. Suppose, the cities are numbered as $(x_1, x_2, \ldots, x_n)$ & C_{ij} denotes the cost of travelling from city i to city j. So, the aim of the travelling salesman prbm is to find the route starting & ending from x_1 covering all the cities.

Process of TSP:- The process is started source destination x_1 then in the next step, the minimum cost city is chosen from x_1 & this process is repeated at every step until all the cities are visited & the final distance obtained is said to be the shortest path obtained from this greedy method.

Example:-



Step 1:- Draw the cost adjacency matrix for the given graph.

Source \ Destination	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	(4)	0	7
H_2	5	0	(2)	4	3	0	0	0
H_3	0	2	0	(1)	0	0	0	0
H_4	(6)	4	1	0	7	0	0	0
H_5	0	(3)	0	7	0	0	6	4
H_6	4	0	0	0	0	0	(3)	0
H_7	0	0	0	0	6	3	0	(2)
H_8	7	0	0	0	(4)	(0)	2	0

in which direct distance will be mentioned & the nodes which are unreachable, their distance is noted by 0 or Infinity.

Distance of a node from itself will be also counted as 0

Step 2:- Now choose the minimum distance from H_1 & mark this its distance in the covered route & the corresponding city will be selected as the next source.

Total distance travelled & route covered:-

$H_1 \xrightarrow{4} H_6$, $H_6 \xrightarrow{3} H_7$, $H_7 \xrightarrow{2} H_8$,

$H_5 \rightarrow$

$H_1 \xrightarrow{4} H_6 \xrightarrow{3} H_7 \xrightarrow{2} H_8 \xrightarrow{4} H_5$

$H_1 \xleftarrow{6} H_4 \xleftarrow{1} H_3 \xleftarrow{2} H_2 \xleftarrow{3}$

= 25

~~Backtracking Approach:-~~

Backtracking :→ It is a designing technique in which there is no need to traverse the complete solution space if the corresponding solⁿ is not leading to optimal solⁿ. Let m_i is the size of set 'S' & there are 'n' no of tuples i.e (m_1, m_2 ---- m_n). These all are possible candidates to satisfy a function. In any of the conventional approach, we have to evaluate all the possible values of 'n' to yield an optimal solⁿ. However, using backtracking approach the solⁿ may be obtained using fewer values of 'n'. i.e if it is realise that the solⁿ vector (m_1, m_2 ----- m_i) where $1 \le i \le m$ is not leading to an optimal solⁿ, then (m_{i+1}, m_{i+2}, ------ m_n) possible vectors can be ignored completely.

All the tuples that will satisfy the explicit constraints define the ~~all the~~ possible solⁿ space for the particular

DAA-Tut :-

Strassen's Matrix Multiplication :- Let a & b are two $n*n$ matrices, the product matrix $C = a*b$ will also be an $n*n$ matrix & is computed by :-

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

where all i & j lying b/w 1 to n.

The product of two $n*n$ matrices can be computed by partitioning them into 4^2 square submatrices, where each submatrix dimension $\frac{n}{2} * \frac{n}{2}$.

The product $a*b$ for 2 by 2 matrix can be compute as :-

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

The running time of normal matrix is $\Theta(n^3)$

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

$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$$

$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$$

$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$$

To compute a & b using conventional method, multiplication of $\frac{n}{2} * \frac{n}{2}$ matrices is performed & for that 8 multiplications & 4 additions operations are req. So, the resultant tym in the form of recurrence relation can be obtained as:-

$$T(n) \begin{cases} b & \text{if } n \leq 2 \\ 8T\left(\frac{n}{2}\right) + cn^2 & \text{if } n > 2 \end{cases}$$

where b & c are constant.

$a = 8$, $b = 2$ $\quad f(n) = n^2$

1. Compute $n^{\log_b a} = n^{\log_2 8} = n^{\log_2 2^3} = n^{3\log_2 2} = n^3$

2. Since, $n^{\log_b a - \epsilon} = f(n)$

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n^3)$$

Since, matrix multiplications are more ~~not~~ expensive than matrix addition or subtraction, Volker Strassen as computed a new way to define C_{ij}. This method involves computing 7 $\frac{n}{2} * \frac{n}{2}$ matrices & then the resultant product matrices is computed by following 7 parameters of Strassen:-

$$P = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$$

$$Q = (A_{21} + A_{22}) \cdot B_{11}$$

$R = A_{11}(B_{12} - B_{22})$

$S = A_{22}(B_{21} - B_{11})$

$T = (A_{11} + A_{12}) \cdot B_{22}$

$U = (A_{21} - A_{11}) \cdot (B_{11} + B_{12})$

$V = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$

$C_1 = P + S - T + V$

$C_2 = R + T$

$C_{21} = Q + S$

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

Using Strassen parameters, 7 multiplication & 18 addition-subtraction are required to compute the value of C_{ij}. So, the resultant time complexity recurrence relation can be obtained as:-

$$T(n) = \begin{cases} a, & \text{if } n \leq 2 \\ 7T\left(\frac{n}{2}\right) + bn^2, & \text{if } n > 2 \end{cases}$$

where a & b are constant.

Compute $n^{\log_2 7} = n^{\log_2 7}$

≈ 2.81

n

2. Since, $n^{\log_b a} = c = f(n)$

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n^{2.81})$$

Since, tym complexity obtained by using strassen's multiplication is significantly smaller than the conventional multipl~ This method is an effective way to minimize the no. of scaler multiplications

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

$$\begin{bmatrix} 2 & 3 \\ 4 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 \\ 3 & 0 \end{bmatrix} = \begin{bmatrix} 11 & 4 \\ 7 & 8 \end{bmatrix}$$

$P = (2+1).(1+0) = 3$

$Q = (4+1).1 = 5$

$R = 2.(2-0) = 4$

$S = 1(3-1) = 2$

$T = (2+3).0 = 0$

$U = (4-2).(1+2) = 6$

$V = (3-1).(3+0) = 6$

$C_{11} = 3+2-0+6 = 11$

$C_{12} = R+T = 4+0 = 4$

$C_{21} = S+2 = 7$

$C_{22} = 3+4-5+6 = 8.$