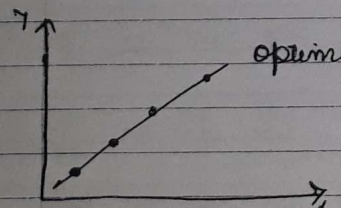
DELTA Pg No. Date / /

Unit II
Dynamic Programming

- Dynamic programming →
  - It solves the problem which has optimization
  - It is just like divide & conquer technique which solves the problem into sub problem and then combines the solution of subprogram into a single solution.
  - Optimization problem are those problem that have many possible sol^n known as fasible sol^n. but we need to choose the best out of all the possible sol^n i.e known as optimal sol^n.



- Steps of Dynamic programming →

Step I → Characterize the structure of optimize sol^n

Step II → Recursively define the value of an optimal sol^n

Step III → Compute the value of an optimal sol^n in bottom-up fashion.

Step IV → Construct an optimal sol^n from the computed information

V. Imp. 1. Longest common subsequence (L.C.S) →

$X = \{2, 3, 4, 5, 6, 7, 8, 9, 10\}$

$Y = \{3, 4, 6, 10, 12, 13\}$

$Z = \{3, 4, 12, 13\}$

- Subsequence → In the order no's are there in main subsequence should be ~~in~~ there in same sequence in subsequence.

like $X = \{2, 3, 4, 5\}$

$Y = \{3, 4, 2\}$ ×

$Y = \{3, 4, 5\}$ ✓

By logical application of

- Biological application often need to compare DNA of two or more diff organism
- So to compute the similarity of two strands of DNA we can apply the sol^n of LCS.

In the longest common subsequence problem we are given two sequences X & Y.

$X = \{x_1, x_2 \cdots x_m\}$

$y = \{y_1, y_2 \cdots y_n\}$

and the sol^n will be to find the LCS of X & Y

- Algorithm LCS - length (X, Y)

1. $m := \text{length}[X]$
2. $n := \text{length}[Y]$
3. let $b[1:m, 1:n]$ & $c[0:m, 0:n]$ are new tables
4. for $i := 1$ to m (row)
5. do $c[i, 0] = 0$
6. for $j = 0$ to n (column)
7. do $c[0, j] = 0$

```
8.  for (i=1 to m)
9.  for (j=1 to n)
10. if (xi = yj)
11  c[i,j] := c[i-1,j-1]+1
12  b[i,j] := " ↖ "
13. else if c[i-1,j] >= c[i,j-1]
14.  c[i,j] := c[i-1,j]
15.  b[i,j] = " ↑ "
16. else c[i,j] = c[i,j-1]
17  b[i,j] = " ← "
18. return c & b
```

```
Print-LCS (b,x,i,j)
1. if i=0 or j=0
2. exit
3. if b[i,j] = " ↑ "
4.  print xi
5.  PRINT-LCS (b,x,[i-1],j-1)
6.  else if b[i,j] = " ↑ "
7.  PRINT-LCS (b,x,i-1,j)
8.  else PRINT-LCS (b,x,i,j-1)
```

O(mn)

X = {A, B, C, B, D, A, B}

Y = {B, D, C, A, B, A}

		Y	B (1)	D (2)	C (3)	A (4)	B (5)	A (6)
0	X	0	0	0	0	0	0	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↑	4↖	4←	4↖
7	B	0	1↖	2↑	4↑	4↑	4←	4←

m=7, n=6, i=7

		Y	B	D	C	A	B	A
	X	0	0	0	0	0	0	0
	A	0	0↑	0↑	0↑	1↖	←1	1↖
	B	0	1↖	←1	←1	1↑	2↖	←2
	C	0	1↑	1↑	2↖	←2	2↑	2↑
	B	0	1↖	1↑	2↑	2↑	3↖	←3
	D	0	1↑	2↖	2↑	2↑	3↑	3↑
	A	0	1↑	2↑	2↑	3↖	3↑	4↖
	B	0	1↖	2↑	2↑	3↑	4↖	4↑

↓ length of LCS

BCBA ⇒ LCS.

Questions $X = \{1, 0, 0, 1, 0, 1, 0, 1\}$

$Y = \{0, 1, 0, 1, 1, 0, 1, 1, 0\}$

	Y	0	1	0	1	1	0	1	1	0
X	0	0	0	0	0	0	0	0	0	0
1	0	0↑	1↑	1←	1↑	1↑	1←	1↑	1↑	1←
0	0	1↑	1↑	2↑	2←	2←	2↑	2←	2←	2↑
0	0	1↑	1↑	2↑	2↑	2↑	3↑	3←	3←	3↑
1	0	1↑	2↑	2↑	3↑	3↑	3↑	4↑	4↑	4←
0	0	1↑	2↑	3↑	3↑	3↑	4↑	4↑	4↑	5↑
1	0	1↑	2↑	3↑	4↑	4↑	4↑	5↑	5↑	5↑
0	0	1↑	2↑	3↑	4↑	4↑	5↑	5↑	5↑	6↑
1	0	1↑	2↑	3↑	4↑	5↑	5↑	6↑	6↑	6↑

↓ length of LCS

Answer = 1 0 0 1 1 0

(1) X

2. Matrix chain multiplication (M.C.M)

- In this problem a sequence of n matrices are to be multiplied and the production is computed
- We multiply two matrices if & only if they are compatible i.e the column of 1st matrix is equal to the row of 2nd matrix.

$(A_1, A_2, A_3 \cdots A_n)$

- Example →

Consider a product of a chain A_1, A_2, A_3

$A_1 : 10 \times 100$ $A_2 : 100 \times 5$ $A_3 : 5 \times 50$

$(A_1 A_2) A_3)$ → No. of scalar multiplication $i \times k \times j \rightarrow s$ (i: 10, k: common 100, j)

$\therefore (A_1 A_2)$ N.S.M $= 10 \times 100 \times 5 = 5000$

and $(A_1 A_2)_{size} = 10 \times 5$ (First Row × second Column)

$((A_1 A_2) A_3) = 10 \times 5 \times 50$

$= 7500$

$\therefore 5000 + 2500 = 7500$

$(A_1 (A_2 A_3))$ No. of scalar multiplication $(A_2 A_3) = 100 \times 5 \times 50$

$= 25000$

$(A_2 A_3)_{size} = 100 \times 50$

$\therefore (A_1 (A_2 A_3)) = 10 \times 100 \times 50$

$= 50000$

$\therefore 25000 + 50000 = 75000$

As dynamic Programming is based on optimization so we have to take N.S.M as minimum.

- MCM problem can be stated as →
given a chain of n matrices $A_1, A_2 \cdots A_n$ where matrix A_i has dimension $P_{i-1} \times P_i$, fully parenthesis the product of $A_1 A_2 \cdots A_n$ in such a way so that no of scalar multiplicat^n are minimum

- Steps of MCM →

(i) The structure of an optimal parenthesization →
The first step is to find the optimum structure & then use it to construct a optimal sol^n which is optimal parenthesizat^n as it can have major impact on the cost of evaluating the product.

(ii) A recursive sol^n →
The cost of an optimal sol^n can be recursively defined in terms of optimal sol^n to the sub problems.

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

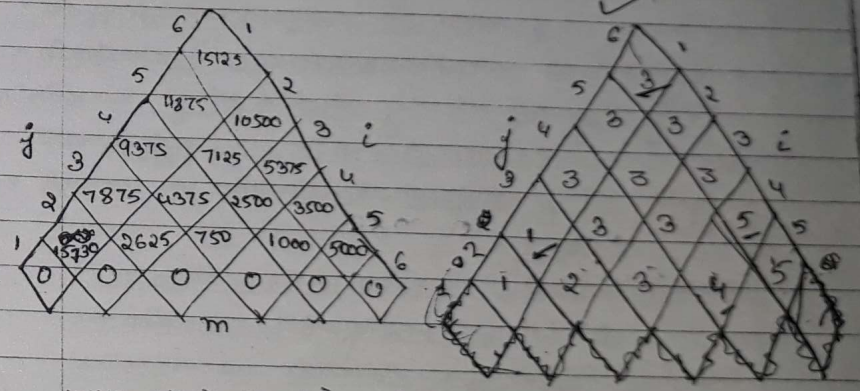
(iii) Computing the optimal cost →

MATRIX-CHAIN-ORDER (p)

1. $n \leftarrow$ length[p] − 1
2. for $i \leftarrow 1$ to n
3. do $m[i,i] \leftarrow 0$
4. for $l \leftarrow 2$ to n // l is length of chain
5. for $i \leftarrow 1$ to $n - l + 1$
6. do $j \leftarrow i + l - 1$
7. $m[i,j] \leftarrow \infty$
8. for $k \leftarrow i$ to $j - 1$
9. do $q \leftarrow m[i,k] + m[k+1,j] + P_{i-1} P_k P_j$
10. If $q < m[i,j]$
11. Then $m[i,j] \leftarrow q$
12. $s[i,j] \leftarrow k$ $\quad \Theta(n^2)$
13. return m & s

Example →

Matrix	A_1	A_2	A_3	A_4	A_5	A_6
Dimension	30×35	35×15	15×5	5×10	10×20	20×25



Here up is the dimension of the matrix & m-l's are the two tables used to store the no of scalar multiplications & k respectively.

$A_i := P_{i-1} \times P_i$

$A_1 = 30 \times 35$

$P_0 = 30$
$P_1 = 35$
$P_2 = 15$
$P_3 = 5$
$P_4 = 10$
$P_5 = 20$
$P_6 = 25$

$\therefore n = 6$ (length of p)

length of 0 to 6 = 7

$\therefore 7 - 1 = 6$

→ Step 1 →

L=2 , i = 1 to 6-2+1 = 1 to 5

(a) i=1 j = 1+2-1 = 2 K = 1 to 2-1 = 1

q = m[1,1] + m[2,2] + p0p1p2

= 0+0+30×35×15 = 15,750

(b) i=2 j = 2+2-1 = 3 K = 2 to 3-1 = 2

q = m[2,2] + m[3,3] + p1p2p3

= 0+0+ 35×15×5 = 2625

(c) i=3 j = 3+2-1 = 4 K = 3 to 4-1 = 3

q = m[3,3] + m[3,3] + p2p3p4

= 0+0+0 + 15×5×10 = 750

(d) i=4 j = 4+2-1 = 5 K = 4 to 5-1 = 4

q = m[4,4] + m[4,4] + p3p4p5

0+0+0+5×10×20 = 1000

(e) i=5 j = 5+2-1 = 6 K = 5 to 6-1 = 5

q = m[5,5] + m[5,5] + p4p5p6

= 0+0+0 + 10×20×25 = 5000

→ Step II →

L=3 , i = 1 to 6-3+1 = 1 to 4

(a) i=1 , j = 1+3-1 = 3 K = 1 to 3-1 = 1 to 2

(i) i=1, j=3 K=1

q = m[1,1] + m[2,3] + p0p1p3 = 0+2625 + 30×35×5 = 5250

(ii) i=1 , j=3 K=2

q = m[1,2] + m[3,3] + p0p2p3 = 1575+0+ 30×15×5 = 2250

2625 = 7875

15750 = 18000

(b) i=2 , j = 2+3-1 = 4 K = 1 to 4-1 = 2 to 3

(i) i=2 , j=4 , K=2

q = m[2,2] + m[3,4] + 35×15×10 = 0+750+5250 = 6000

(ii) i=2 , j=4 , K=3 = m[2,3] + m[4,4] + 35×5×10

(c) i=3 , j=5 K=3 to 4

(i) i=3, j=5, K=3

q = m[3,3] + m[4,5] + 15×5×20 = 0+1000+1500 = 2500

(ii) i=3 , j=5 , K=4

q = m[3,4] + m[5,5] + 15×10×20 = 750+0+3000 = 3750

(d) i=4 , j=6 , K=4 to 5

K=4 q = m[4,4] + m[5,6] + 5×10×25 = 6250

K=5 q = m[4,5] + m[6,6] + 5×20×25 = 3500

→ Step III →

L=4 i = 1 to 3

(a) i=1 , j = 1+4-1 = 4 K = 1 to 4-1 = 1 to 3

K=1 q = m[1,1] + m[2,4] + 30×35×10 = 14875

K=2 q = m[1,2] + m[3,4] + 30×15×10 = 15750+750+4500

= 21000

K=3 q = m[1,3] + m[4,4] + 30×5×10 = 9375

(b) i=2 , j = 1+5-1 = 5 K = 2 to 4

K=2 = 13000

K=3 = 7125

K=4 = 11375

(c) i=3 , j = 3+4-1 = 6 K = 3 to 5

K=3 q = m[3,3] + m[4,6] + 15×5×25 = 5375

K=4 q = m[3,4] + m[5,6] + 15×10×25 = 9500

K=5 q = m[3,5] + m[6,6] + 5×20×25 = 10000

→ Step IV →

i=5 i=2 K = 1 to 2-1 = 1 to 2

(a) i=1 , j = 1+5-1 = 5 K = 1 to 5-1 = 1 to 4

K=1 q = m[1,1] + m[2,5] + 30×35×20 = 28125

K=2 = 27250

K=3 = 11875

K=4 = 15375

(b) i=2, j= 2+5-1=6 K=2 to 5

K=2 = 16625

K=3 = 10500

K=4 = 18125

K=5 = 24625

→ ~~Step-II~~ →

L=6 i=6-6+1=1 to 1 K=1 to 6-1=1 to 5

(a)(i) K=1 = 36750

(ii) K=2 = 32375

(iii) K=3 = 15125

(iv) K=4 = 9375+5000+7500 = 21875

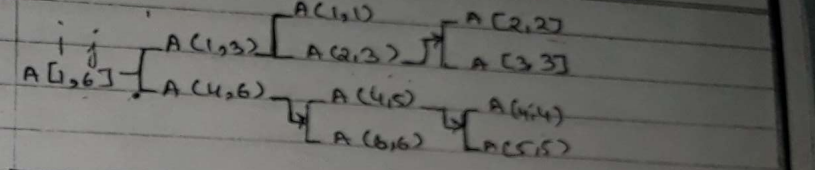
(v) K=5 = 11875

~~(b) i=2 j=6~~

(iv) ~~Construct & Optimal Solution~~ →

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

1. If i=j
2. then print Ai
3. else print "("
4. PRINT-OPTIMAL-PARENS (S, i, S[i,j])
5. PRINT-OPTIMAL-PARENS (S, S[i,j]+1, j)
6. PRINT ")"



((A1(A2A3))((A4A5)A6)) ⇒ Answer

3# Optimal Polygon Triangulation →

- A polygon can be described as P=(v0,v1,v2...vn-1) where V denotes the vertices.
- It is a piecewise linear closed curve in the plane.
- The pieces are known as side.
- A point joining two consecutive its known as vertex
- The set of the points on the polygon formed its boundary.
- The set of points surrounding the polygon formed its exterior
- OPT is very much similar to MCM & is based upon convex polygon.
- A polygon is convex if given any two points on its boundary or interior i.e all the points on the line segment drawn b/w them are contained in polygon boundary or interior.
- Chord →

It is line segment that join two non adjacent vertices & they should never intersect each other.

Algo of mcm only

DELTA Pg No. Date / /

- Triangulation →

It is the set of code that divides the polygon P into non-overlapping triangles i.e all the codes should not intersect with each other.

It is technique you solving numerical problem that involves dividing a i/p shape into simple element such as triangles.

For n no of vertices n-3 chords ✓

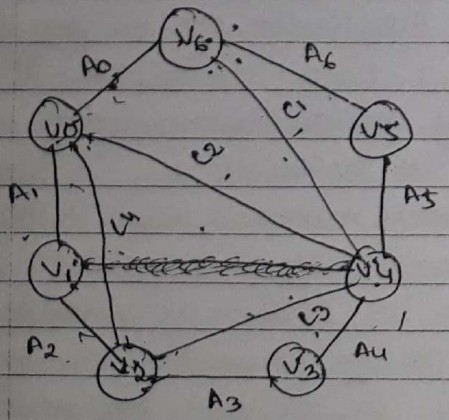
n-2 triangles are formed.

Example → Draw a polygon with 7 vertices.

$V_0, V_1, V_2, V_3, V_4, V_5, V_6$

Step I →

In this polygon sides are labled with the name of matrices & dimensions will be denoted by the vertices of the corresponding matrix.



Step II → Now draw the chords on above polygon with formula for n vertices as n-

For on the no of chords will be 7-3=4

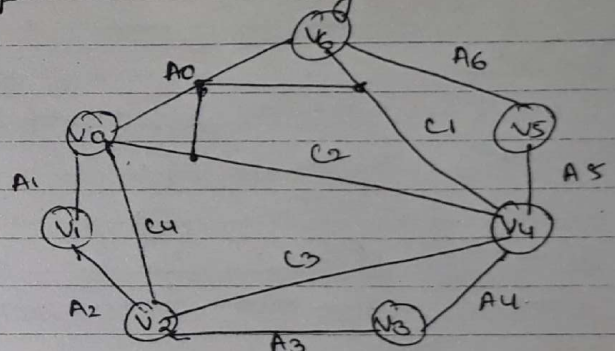
The chord will be drawn in such a way that a line will join two non adjacent vertices and main vertices of polygon are covered.

7-3=4 chords

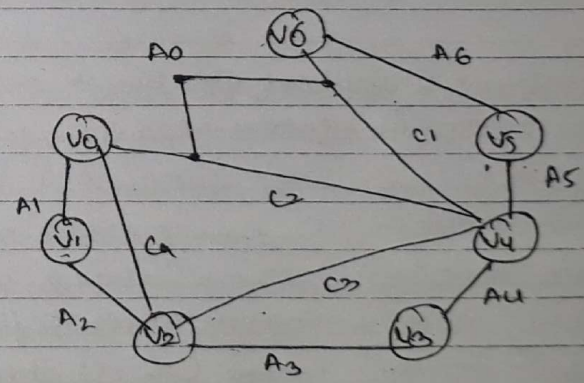
7-3=5 triangle

$(V_6\ V_5\ V_4)$ (V_0, V_6, V_4) (V_0, V_1, V_2) (V_2, V_3, V_4)

Step III → Now mark a bisecting points on the edge A0 that will bisect two chords C1 & C2 now remove the line A0 on which the bisecting point lies initially.

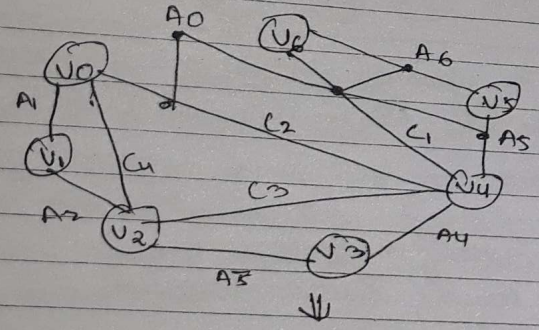


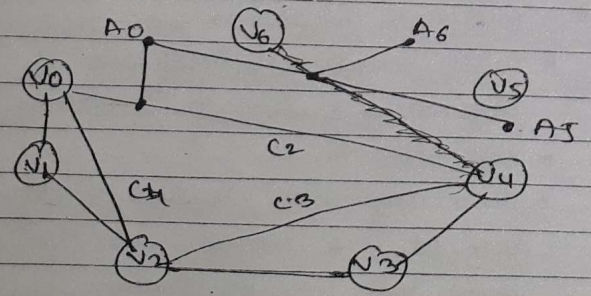
then



Step IV → Now again mark the intersecting points from the chord C_1 to the two edges of polygon i.e A_5 & A_6 & then remove all the edges on which the intersecting point lies.

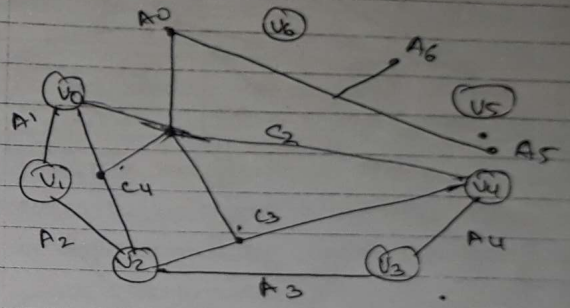
Bcz the intersecting point on the edge A_5 & A_6 dont intersect further bcz there are no more edges in that direct^n so the edge line A_5 & A_6 & the code C_1 is remove.

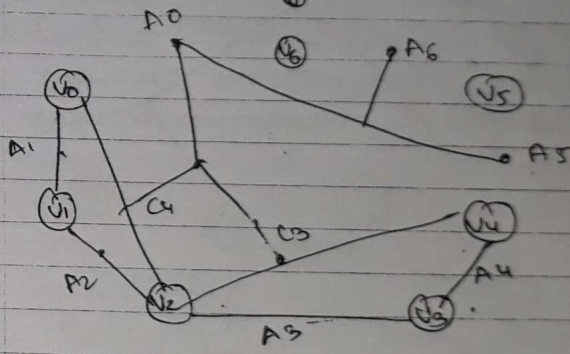




Step V →

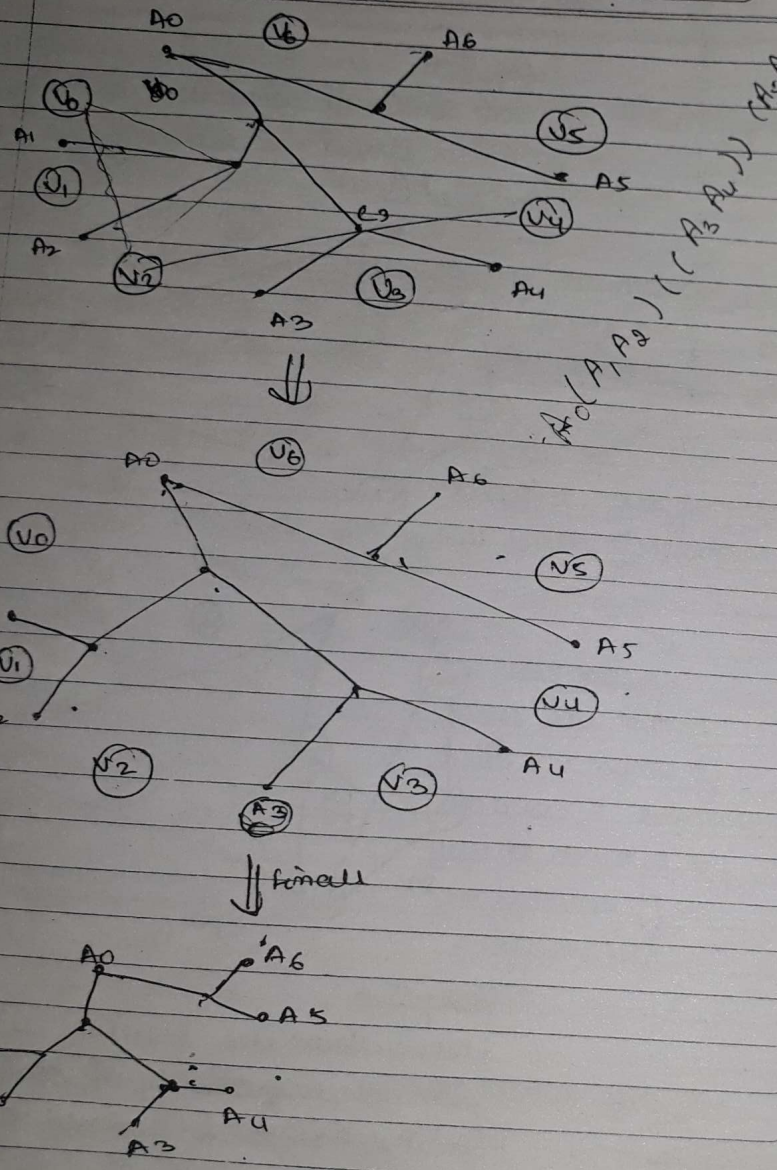
Now Similarly draw the bisecting line on the chord C_4 & C_3 from the bisecting line C_2 & then remove the code C_2.





Step VI →

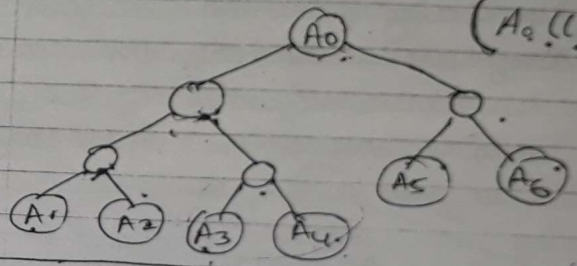
Now draw the bisecting line on chord C_3 & C_4 to A_3, A_4 & A_1, A_2 respectively & then remove the side A_1, A_2, A_3, A_4 & chords C_3 & C_4 this way.



$A_0(A_1A_2)((A_3A_4))(A_5A_6)$

Step III → The obtained structures of the bisecting can be converted into a tree like structure in which A_0 is root & A_1 A_2 A_3 A_4 are part of left subtree & A_5 A_6 are part of right subtree

$(A_0((A_1A_2)(A_3A_4))(A_5A_6))$



The optimal sol^n of this problem is
$A_0((A_0((A_1A_2)(A_3A_4))(A_5A_6))$ → Answer.

Greedy Algorithm

1. Activity selection Problem
2. Huffman Codes
3. Knapsack Problem
4. Task - scheduling Problem
5. Travelling Salesman Problem.

- Definition →
- These algorithm always makes a choice that look best at the moment i.e they always go for the local best possible solution with a chance that it may lead to global optimal solution.
- They may not always produce the optimal sol^n but it is quite powerful for many problems they can

produce optimal soln

- Greedy technique →
- These algo solve the problem by making the choice that seems best at particular moment. Many optimization problem can be solved using a Greedy algorithm.
- It is mainly used you the problem have no efficient soln but these algorithm may provide the soln that is close to optimal soln.

- Elements →

1. Choice of property →
A global optimal soln can be arrived at by making global optimal soln, which can be made by making different greedy choices
En → greedy you min cost, max profit, min distance etc.

2. Optimal substructure →
optimal soln contains optimal sub soln ie if soln of the sub problem are optimal then it may lead to a global optimal soln.

★ Huffman Codes →

- It is a compression technique
- These codes are used to compress the data efficiently, they can compress 20-90% of data depending on the characteristics of data beg being compressed.

Example → Suppose we are have 100,000 character file and we wish to store it compactly. there are 6 diff characters a-f & the following table describes the frequency of letter occurring in that file

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

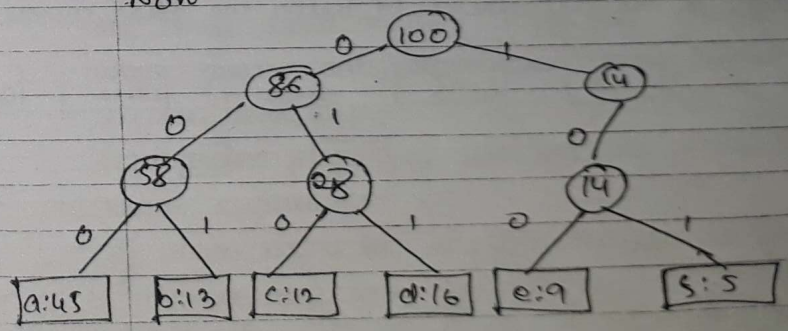
Answer → We have to do encoding.

1. Fixed length code → In this technique each letter is represented by an equal no of bits with a fixed length code.

a:45	b:13	c:12	d:16	e:9	f:5

then add the frequency of two character & make there parent & give 0 on left & 1 on right.

Now



Total storage space taken by character using FLC →
= (45×3 + 13×3 + 12×3 + 16×3 + 9×3 + 5×3) × 1000

$(135 + 39 + 36 + 48 + 27 + 15) \times 1000$

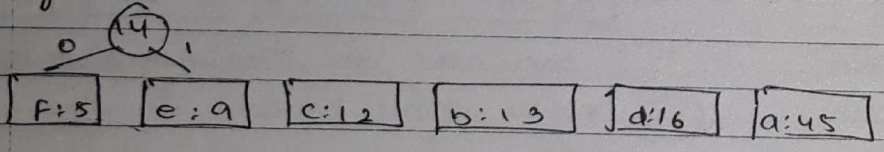
$= 300 \times 1000$

$= 3,00,000$ bits

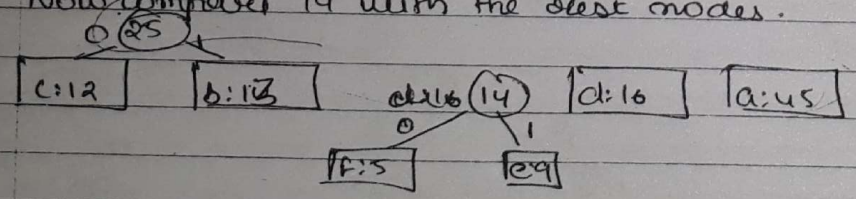
2. Variable length code (VLC) →

- This technique uses prefix codes it is considerable a better technique than a FLC bcz it assigns the frequent letters shorter codes & vice-versa.
- It is based upon the following length of $\frac{1}{\text{Frequency}}$.

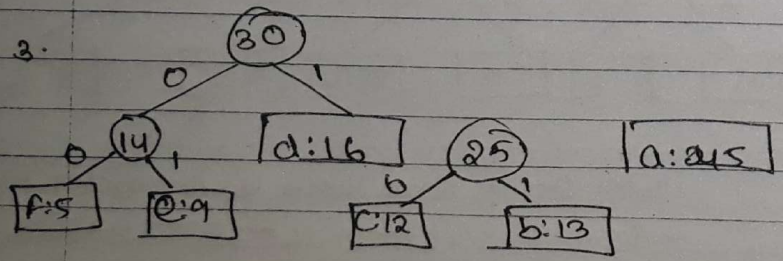
Step 1. arrange all the node in ↑ order of their frequencies.



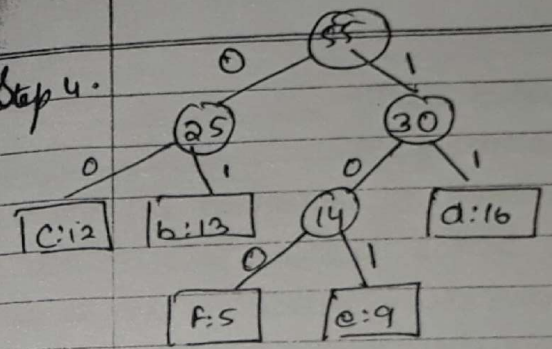
Step 2. Now compare 14 with the next nodes.



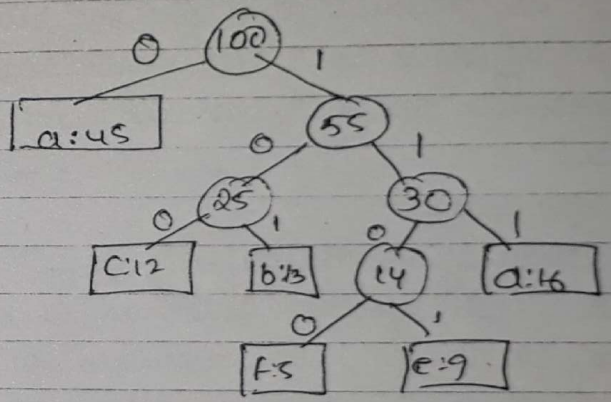
Step 3.



Step 4.



Step 5.



Total storage space →

$= (45 \times 1 + 13 \times 3 + 12 \times 3 + 16 \times 3 + 9 \times 4 + 5 \times 4) 1000$

$= (45 + 39 + 36 + 48 + 36 + 20) 1000$

$= 224,000$ bits

NOW $\frac{224000}{300000} \times 100$

74.6%.

- ~~Algorithm~~ →
- C = Set of n characters & each character as an attribute frequency denoting the no of occurances of corresponding alphabet.
- Q = Min priority Queue that arranges all the letters in ↑ order
- Extract-min retrieves the min element from the queue.

DELTA Pg No. Date / /

HUFMAN (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 = EXTRACT-MIN(Q)
7. freq[z] = freq[x] + freq[y]
8. INSERT (Q, z)
9. return EXTRACT-MIN(Q) // return root of tree

O(n log n)

★ Travelling Salesman Problem (TSP) →

In this problem a salesman needs to visit n cities in such a manner that all the cities must be visited only once and in the end he returns to the city from where he started with the minimum cost.

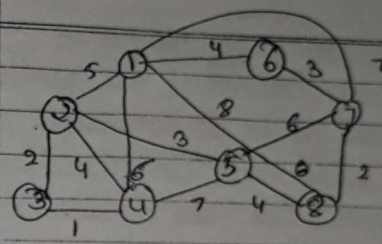
Let the city denoted by = $x_1, x_2 \dots x_n$

Cost of traveling from city i to city j = c_{ij}

So the TSP is to find the least starting & ending at x_1 that will take in all the cities with the minimum cost.

Example →

DELTA Pg No. Date / /



- Step I → Draw cost - adjacency matrix.

Source \ Destination	I	II	III	IV	V	VI	VII	VIII
1	0	5	–	6	–	(4)	7	8
2	5	0	(2)	4	3	–	–	–
3	–	2	0	(1)	–	–	–	–
4	(6)	4	1	0	7	–	–	–
5	–	(3)	–	7	0	–	6	4
6	4	–	–	–	–	0	(3)	–
7	7	–	–	–	6	3	0	(2)
8	8	–	–	–	(4)	–	2	0

- Step II → Starting with the source city 1 choose the road having min cost except 0 & then make this destination node the source node in the next step.

Repeat the procedure till all the cities have been explored.

If there is a tie in two cities we can choose any one randomly.

$1 \xrightarrow{4} 6 \xrightarrow{3} 7 \xrightarrow{2} 8 \xrightarrow{4} 5 \xrightarrow{3} 2 \xrightarrow{2} 3 \xrightarrow{1} 4 \xrightarrow{6} 1$

Total Cost = 25

* Task scheduling problem →

- Principles do minimize the penalty
- In this we have to optimally schedule the unit time task on a single processor where each task has a deadline & the penalty that must be paid if the dead line is missed.
- A unit time task is a job on a program that is to be run on a comp that require exactly one unit of time to complete.
- This problem has the following i/p.

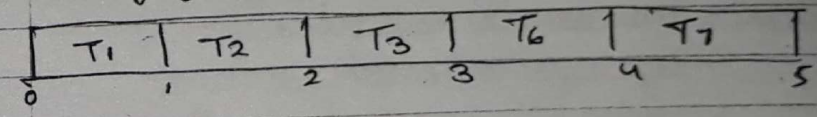
1. S = Set given 1-n = {1, 2, 3 ... n} of n unit time task.
2. A set of 'n' integer deadline $d_1, d_2 \cdots d_n$ and each task i is supposed to be finished by deadline d_i "$i \to d_i$".
3. A set of n non-ve weights or penalty $w_1, w_2 \cdots w_n$ such that a penalty w_i is incurred if task i is not finished by time d_i & no penalty is incurred if a task is finished in its deadline.

Example →

Question: Find the optimal solution for the following task for given penalty & deadline.
All these task are unit task & are to be run on a single processor. having 5 time slots starting from 0.

Task	1	2	3	4	5	6	7
deadline	4	2	4	3	1	4	6
Penalty	70	60	50	40	30	20	10

1. Greedy for maximum penalty →

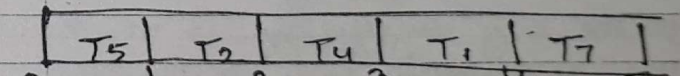


Total Penalty = T4+T5+T6+T7 + T1+T5
= 40 + 30
= 70.

If two has same penalty given the one with minimum deadline.

2. Greedy for minimum Penalties deadline →
In this approach we are executing the task on the basis of penalty that is the task having max penalty will be executed first.

In this approach we execute the task on the basis of the deadline that is the task having minimum deadline will be executed first.



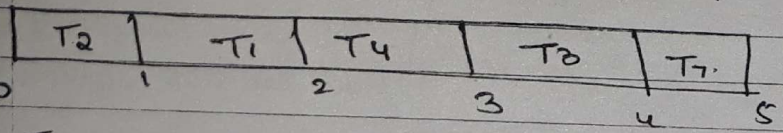
$T_1 = 4 \quad T_3 = 4 \quad T_6 = 4$

∴ The dead line is same we will execute the one with max penalty.

Total Penalty = $T_3 + T_6$
= 50 + 20
= 70.

* Greedy for minimum deadline & maximum penalty

In this approach our motive is to choose the task having minimum deadline as well as maximum penalty.



Total penalty = T5 + T6
= 30 + 20
= 50

* Knapsack problem →

- Consider a knapsack or bag having 'n' no of objects & each object 'i' has weight wi & profit 'pi' & the max capacity of knapsack is 'm'
- If a fraction xi such that $0 \le x_i \le 1$ of object i is placed into the knapsack then profit $p_i x_i$ is earned.
- So the objective is to obtain the filling of knapsack in such a way that the total profit earned maximum
- So the problem can be stated as

$$\text{Maximize} \sum_{1 \le i \le n} p_i x_i$$

$$\text{Subject to} \sum_{1 \le i \le n} w_i x_i \le m$$

and $0 \le x_i \le 1$ & $1 \le i \le n$

- It is of two type →
1. Fractional knapsack Problem
2. 0/1 knapsack Problem

Example / difference →
either we can pick object or cannot in the case of 0/1 knapsack Problem.
like 10, 15, 20

& m = 20 (m: max capacity of knapsack) if we pick 10 we cannot pick 20 or 15 (if remaining object have weight more than 10) and if we use fractional knapsack the we can take fraction.

If m = 20 $w = 10 + \frac{10}{15}^{2}(15)^{5}_{3}$
$= 10 + 10$
$= 20$.
we can pick fraction of second object.

1. Fractional knapsack problem →
In this problem it is possible to extract the fractional part of an object.
2. 0/1 knapsack problem →
In this problem either the complete object is selected or reject i.e it is not possible to extract any fraction out of the object.
* If it is given then use any one & if not use fraction*

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

1. Greedy for max profit

x_1	x_2	x_3 (fraction)
1	2/15	0

$\sum_i w_i x_i = 18 \times 1 + \frac{2}{15} \times 15 = 20$

$\sum_i p_i x_i = 25 \times 1 + \frac{2}{15} \times 24 = 25 + 3.2$

$= \boxed{28.2}$

2. Greedy for minimum weight →

x_1	x_2	x_3 (fraction)
10/15	1	

$\sum_i w_i x_i = 10 \times 1 + \frac{10}{15} \times 15 = 20$

$\sum_i p_i x_i = 15 \times 1 + \frac{\overset{2}{10}}{\underset{3}{15}} \times \overset{8}{24} = 15 + 16 = \boxed{31}$

3. Greedy for profit per unit weight →

x_1 x_2 x_3

$\frac{p_1}{w_1} = \frac{25}{18}$ $\frac{p_2}{w_2} = \frac{\overset{8}{24}}{\underset{5}{15}}$ $\frac{p_3}{} = \frac{15}{10}$

$= 1.3$ $= \boxed{1.6}$ $p_3 = 1.5$

x_1	x_2	x_3 (fraction)
1	5/10	

$\sum_i w_i x_i = 15 \times 1 + \frac{5}{10} \times 10 = 20$

$\sum_i p_i x_i = 24 \times 1 + \frac{5}{10} \times 15 = 24 + 7.5 = \boxed{31.5}$

- Algorithm →

Algorithm greedy knapsack (m, n)

// p[1:n] & w[1:n] contain profits & weights respectively of the 'n' objects ordered such that $\frac{p[i]}{w[i]} \geq \frac{p[i+1]}{w[i+1]}$

{ Arrange in descending order 1.6 > 1.5 > 1.3 } 'm' is the knapsack size & x[1:n] is solution vector.

```
1. For i := 1 to n
2. do x[i] := 0   // initialize x
3. U := m;
4. for i := 1 to n do   (any variable)
5. {
6. If (w[i] > U) then break;
7. x[i] := 1 , U = U - w[i];
8. }
9. if (i ≤ n) then x[i] := U/w[i]
10. }
```

O(n)

★ Activity selection problem →

- There is a set S of n proposed activity such that $S = \{a_1, a_2, a_3 \cdots a_n\}$
- These activity a_i has start time 's_i' & finish time f_i, where $0 \leq s_i < f_i < \infty$
- Activities a_i & a_j are compatible if the intervals (s_i, f_i) & (s_j, f_j) do not overlap. i.e if either $s_i \geq f_j$ or $s_j \geq f_i$

• In activity selection problem we wish to select max size subset of mutually compatible activity.

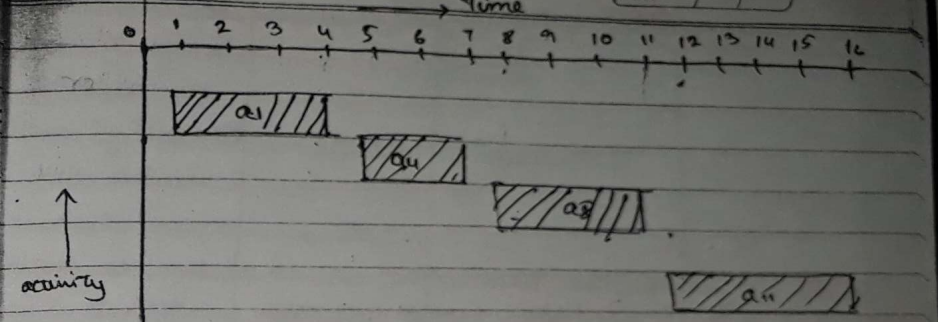
Recursive activity - selector (s, f, k, n) → total no of activity
- s → starting time of activity
- f → finishing time of activity
- variables initialized to 0

```
1 m := k+1
2 while m ≤ n & s[m] < f[k] // find the first activity in set to finish.
3 m := m+1
4 if m ≤ n
5 return {am} ∪ Recursive activity selector
6 else return ∅
```

Greedy - activity - selector (s, f)

```
1 n := length [s]
2 A := {a1}
3 k := 1
4 for m := 2 to n,
5 if s[m] ≥ f[k]
6 A = A ∪ {am}
7 k = m
8 return A
```

fg i	1	2	3	4	5	6	7	8	9	10	11
si	1	3	0	5	3	5	6	8	8	2	12
fi	4	5	6	7	9	9	10	11	12	14	16



Divide & Conquer	Dynamic programming	Greedy Method
1. This technique is executed in 3 levels of recursion (a) Divide the problem into no of subproblem (b) Conquer the subproblem by solving it recursively if the size of subproblem is small enough then solve it directly (c) combine the sol^n of subproblems into sol^n of original problem	The development of DP algorithm can be broken into sequence of 4 steps (a) Characterized the structure of an optimal sol^n (b) Recursively define the value of optimal sol^n (c) compute the value of an optimal sol^n in a bottom up fashion (d) construct an optimal sol^n from the computed information.	This technique has five components. (a) a candidate set from which sol^n is created (b) A select^n f^n which chooses the best candidate that is to be added to sol^n (c) A feasibility f^n that is use to determine if a candidate can be use to contribute to the sol^n (d) an objective f^n which assign

	Divide & Conquer	DP	Greedy method.
			The value is a sol^n or partial sol^n ⇒ a sol^n y^n which will indicate when we have discovered a complete optimal sol^n
2.	In this technique problem is divided into small subproblem & these subproblem can solved in independay & finally all sol^n of subproblem are collected together to get sol^n of the given problem.	In this many decision sequences are generated & all the over-lapping circum-stances are considered	In this it is not necessary to divide the problem into subproblem to get the sol^n.
3.	In this the duplication in ~~subpro~~ subsol^n are neglected that is duplicate subsol^n may be obtain.	Here computed duplicat^n in sol^n are avoided totally	There are very less chances of duplicat^n because generally problem doesn't share subproblem.
4.	It is less efficient bcz of remover on sol^n	It is more efficient than DAC	It is also efficient bcz optimum select^n is done without revising previously

	Divide & Conquer	Dynamic Programming	Greedy Method
			generated sol^n
5.	It uses top down approach of problem solving i.e recursive method.	It uses bottom-up approach of problem solving i.e iterative method.	It also uses top down approach.
6.	It splits the i/p at specific deterministic point usually in the middle	It splits its i/p at every possible split point rather than at a particular point after trying all the possible split point it determines which split point is optimal	In this a set of feasible sol^n is generated & then optimal sol^n is picked.
7.	They call themselves recursive bcz one or more time we need to deal with closely related subproblem.	It is not recursive & it considers all the possible sequences in order to obtain a final sol^n	In this a set of feasible sol^n is generated & the sol^n maximizing or minimizing the opti objective sol^n is considered as optimal sol^n.

	Divide & Conquer	Dynamic Programming	Greedy method
8.	All the subproblem are independent of each other therefore an optimum sol^n can be local or global	It is guaranteed that it will generate the optimal sol^n using the principle of optimality	Only local optimizat^n is achieved.
9.	It does more work on subprob-lem & hence does more time consumption bcz subproblems further share sub-subproblem.	It solves the subproblem only once & then store it in the table hence time consumption is less	Time consume is less than both the other techniques bcz there is no sharing of sub-subproblem
10.	Ex→ • Quick sort • Binary search. • merge sort	Ex→ • MCM • LCS • OPT etc.	Ex→ • Knapsack problem activity select^n problem • Travelling salesman etc.

Dynamic Programming | Greedy method
It uses bottom-up approach | It also uses top-down approach.

DELTA Pg No. Date

★ Graph →
A graph
vertices
or vertex

On the basis
direction
edges
weights
→ Directed
→ Undirected

On the basis
connection
edges
→ Connected
→ Disconnected

all the
The ed
Called