Dynamic Programming:

→ It is typically applied to optimization problem.

→ In such problems there are many possible solutions.

→ Each solution has a value, and we wish to find a sol^n. with the optimal value (max or min value).

→ We call such a sol^n. an optimal sol^n. to the problem, since there may be several solutions that achieve the optimal value.

⇒ The development of dynamic programming algo is broken into 4 steps:-

→ Characterize the structure of optimal solution

→ Recursively define the value of optimal sol^n.
Like divide & conquer, divide the problem into two or more optimal parts, recursively call them & find the sol.^{n}.

→ Compute the value of optimal solution in a bottom-up fashion

→ Construct the optimal sol^n. from the computed value of smaller subproblems.

⇒ Dynamic programming is best applied to the problem having these two characteristics.

→ Optimal substructure.

→ Overlapping subproblems.

Ingredients/Characteristics of dynamic programming -

→ Builts on previous subproblems

→ It only calculates enough sub problems to get to the next step.

→ Every subproblem you solve involves a decision.

→ Decisions in subproblems are not based on decisions in previous subproblems.

→ Optimal solution to a subproblem is extended from previous optimal solutions.

Dynamic programming:	Divide & conquer
1. It guarantees an optimal solution.	1. It may or may not provide an optimal solution.
2. It uses bottom-up approach.	2. It uses top-down approach.
3. ~~Binary search algorithm~~ Floyd-Warshall algo. uses dynamic programming	3. Binary search algorithm follows divide & conquer approach.

4. DP is more efficient	4. Divide & conquer is less efficient.
5. Dynamic programming is a iterative method.	5. Divide & conquer is a recursive method.
6. Dynamic programming is needed when subproblems are dependent	6. Divide & conquer works best when all subproblems are independent.

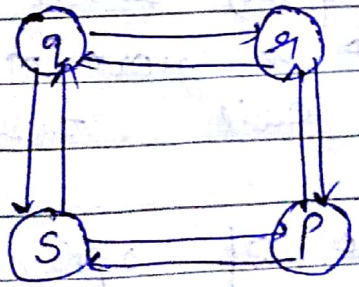
* Emphasis on optimal substructure & overlapping substructure

Optimal substructure

1. A problem is said to have optimal substructure if an optimal sol.^{n} can be constructed efficiently from optimal sol^{n}s. of its subproblems.

2. A greedy algorithm is used to solve a problem with optimal substructure.

3. The shortest path problem follows optimal substructure property.



Overlapping substructure

A problem is said to have overlapping substructure if the problem can be broken down into subproblem which are reused several times.

A dynamic programming is used to solve a problem with overlapping substructure.

3. The problem of computing the fibonacci sequence exhibits overlapping subproblem.

```
int fib(int n)
{
  if (n <= 1)
  return;
  return fib(n-1) + fib(n-2);
}
```

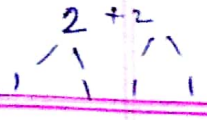
Memorization

Apart from bottom-up approch fashion, dynamic programming can be implemented using memorization.

Memorization is a optimization technique used primarily to speed up the computer program by the storing the results and return the stored result when the same computation is needed again.

Using memorization we implement an algorithm recursively, but we also keep track of all of the substitution. If we answer a subproblem that we have seen, then we look up for the solution in the stored table. If we encounter a subproblem that we have not seen then each subsequent time that the subproblem is encountered the value stored in the table is simply looked up and returned.

Memorization offers the efficiency of dynamic programming. It maintains the top down recursive strategy.



Complexity = O(n)

Algorithm

At every step i, f(i) performs the following steps:

[a] Checks whether table [i] is NIL or not.

[b] If its not NIL, f(i) returns the value of table [i].

[c] If its NIL and i satisfies the base condition, we update the lookup table with the base value and return the same.

[d] If its NIL and i does not satisfies the base condition, then f(i) splits the problem 'i' into subproblems and recursively calls itself to solve them.

[e] After the recursion call returns, f(i) combins the solution to subproblems, update the lookup table and returns the solution for problem i.

Matrix - Chain Multiplication

→ Matrix chain multiplication is an optimization problem that can be solved using dynamic programing

In this a sequence of matrix is given, we want to find the most efficient way to multiply these matrices together.

The problem is not actually to perform the multiplication, but to decide in which order to perform the multiplication.

⇒ Decide optimal substructure to minimize multiplication cost.

$$c[i][j] = \min_{i \le k < j} \left[C[i][k] + C[k+1][j] + (P_{i-1} * P_k * P_j) \right]$$

Algorithm

Matrix Chain (p(0..n), n)

```
1.  for i = 1 to n
2.      C[i][i] = 0
3.  End For          // length
4.  // length
5.  for length = 1 to n-1
6.      For i = 1 to n-length
7.          j = i + length
            C[i][j] = Min [C[i][k] + C[k+1][j] + (P_{i-1} * P_k * P_j)]
                      i<=k<j
```

for P

8. s[i][j] = k // K is at min of c[i][j]
9. End for
10. End for
11. Return c[i][j]

print Optimal (s[1....n] i, j)
1. if i = j then print "A" i
2. Else
3. Call Print Optimal (s, i, s[i][j])
4. Call Print Optimal (s, s[i][j] + 1, j)

Complexity - $O(n^3)$

Ques Consider the following four matrices. Show how to multiply this matrix chain optimally.

$A_1 \times A_2 \times A_3 \times A_4$
$(30 \times 1) \quad (1 \times 40) \quad (40 \times 10) \quad (10 \times 25$

Sol" The dimensions of four matrices are represented by the vector p_i, for i = 0 to 4

$p_0 = 30$
$p_1 = 1$
$p_2 = 40$
$p_3 = 10$
$p_4 = 25$

When length = 0

$$c[i][j] = \min\left[c[i][k] + c[k+1][j] + (p_{i-1} \times p_k \times p_j)\right]$$

$s[i][j] = k$ at minimum of $c[i][i]$

$c[1][1] = 0$ $\quad s[1][1] = N_i'l$

$c[2][2] = 0$ $\quad s[2][2] = N_i'l$

$c[3][3] = 0$ $\quad s[3][3] = N_i'l$

$c[4][4] = 0$ $\quad s[4][4] = N_i'l$

} x

When length 1

→ $c[1][2] = c[1][1] + c[2][2] + p_0 \times p_1 \times p_2$

$= 0 + 0 + 30 \times 1 \times 40$

$= 1200$

$s[1][2] = 1$

→ $c[2][3] = c[2][2] + c[3][3] + p_1 \times p_2 \times p_3$

$= 0 + 0 + 1 \times 40 \times 10$

$= 400$

$s[2][3] = 2$

→ $c[3][4] = c[3][3] + c[4][4] + p_2 \times p_3 \times p_4$

$= 0 + 0 + 40 \times 10 \times 25$

$= 10,000$

$s[3][4] = 3$

When length = 2

→ $C[1][3] = \min$ { when k = 1 : $C[1][1] + C[2][3] + p_0 \times p_1 \times p_3$ }

$= 0 + 400 + 30 \times 1 \times 10$

$= 400 + 300$

$= 700$

when k = 2

$\min\ C[1][2] + C[3][3] + p_0 \times p_2 \times p_3$

$= 1200 + 0 + 30 \times 40 \times 10$

$= 1200 + 12000$

$= 13200$

As 700 is min 180, $s[1][3] = 1$

→ $C[2][4]$ → (k = 2, 3)

when k = 2

$C[2][4] = \min [C[2][3] + C[3][4] + p_1 \times p_2 \times p_4]$

$= 0 + 10000 + 1 \times 40 \times 25$

$= 10000 + 1000$

$= 11000$

when k = 3

$C[2][4] = \min [C[2][3] + C[4][4] + p_1 \times p_3 \times p_4]$

$= 400 + 0 + 1 \times 10 \times 25$

$= 400 + 250$

$= 650$

As $C[2][4] = 650$ is min when $s[2][4] = 3$

→ When length = 3

$C[1][4]$ = min K = 1, 2, 3

When K=1

$C[1][4] = \min [C[1][1] + C[2][4] + P_0 * P_1 * P_4]$

$= 0 + 650 + 30 \times 1 \times 25$

$= 650 + 750$

$= 1400$

When K=2

$C[1][4] = \min [C[1][2] + C[3][4] + P_0 * P_2 * P_4]$

$= 1200 + 10000 + 30 \times 40 \times 25$

$= 11200 + 30000$

$= 41200$

When K=3

$C[1][4] = \min [C[1][3] + C[4][4] + P_0 * P_3 * P_4]$

$= 700 + 0 + 30 \times 10 \times 25$

$= 700 + 7500$

$= 8200$

As $C[1][4] = 1400$ is min at $S[1][4] = 1$

C[i][j]

i\j	1	2	3	4
1	0	1200	700	1400
2		0	400	650
3			0	10000
4				0

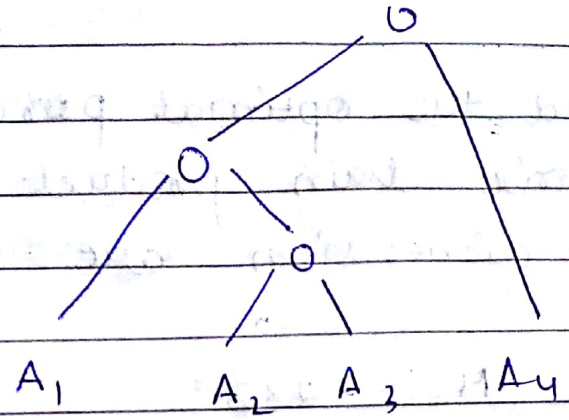
s[i][i]

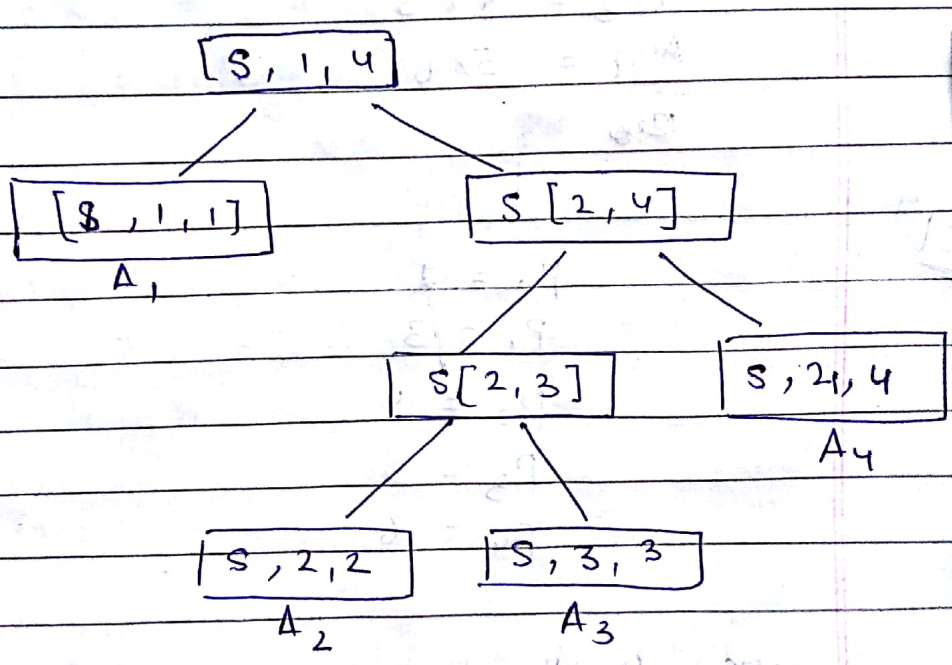
i\j	1	2	3	4
1	0	1	1	1
2		0	2	3
3			0	3
4				0

$A_1 A_2 A_3 A_4$.

$(A_1)(A_2 A_3 A_4)$

$A_1 (A_2 A_3) A_4$.





Cost = $A_2 \times A_3$ = (1 x 40) X (40 X 10)

= 1 x 40 X 10

= 400

$= (A_2 \times A_3) A_4$ = (1 x 40 x 10) x 10 x 25

= 1 x 40 x 10 x 25

= 250

$A_1 \cdot ((A_2 \times A_3) A_4)$ = (30 x 1) (1 x 40 x 10) (10 x 25)

= (30 x 1) (1 x 10 x 25)

= 30 x 1 x 25

Total number of multiplications

$= 400 + 250 + 750$

$= 1400$

Ques Find the optimal parenthesization of matrix chain product whose sequence of dimension are $<2, 3, 4, 5, 6>$

$M_1 = 2 \times 3$

$M_2 = 3 \times 4$

$M_3 = 4 \times 5$

$M_4 = 5 \times 6$

Sol^n

$p_0 = 2$

$p_1 = 3$

$p_2 = 4$

$p_3 = 5$

$p_4 = 6$

When length = 0

$c[i][j] = \min c[i][k] + c[k+1][j] + p_{i-1} * p_k * p_j$

$s[i][j] = k$ at min of $c[i][j]$

$c[1][1] = 0$

$c[2][2] = 0$

$c[3][3] = 0$

$c[4][4] = 0$

When length = 1

$$C[1][2] = \min[C[1][1] + C[2][2] + p_0 \times p_1 \times p_2]$$
$$= 0 + 0 + 2 \times 3 \times 4$$
$$= 24$$

$S[1][2] = 1$

$$C[2][3] = \min[C[2][2] + C[3][3] + p_1 \times p_2 \times p_3]$$
$$= 0 + 0 + 3 \times 4 \times 5$$
$$= 60$$

$S[2][3] = 2$

$$C[3][4] = \min[C[3][3] + C[3][4] + p_2 \times p_3 \times p_4]$$
$$= 0 + 0 + 4 \times 5 \times 6$$
$$= 120$$

$S[3][4] = 3$

When length = 2

$C[1][3] = k, 1, 2$

When $k = 1$

$$C[1][3] = \min\ C[1][1] + C[2][3] + p_0 \times p_1 \times p_3$$
$$= 0 + 60 + 2 \times 3 \times 5$$
$$= 60 + 30$$
$$= 90$$

When $k = 2$

$$C[1][3] = \min[C[1][2] + C[2][3] + p_0 \times p_2 \times p_3]$$
$$= 24 + 60 + (2 \times 4 \times 5)$$
$$= 84 + 40$$
$$= 124$$

As c[1][3] = 90 is min when s[1][3] = 1

c[2][4] ⇒ k = 2, 3

when k = 2

c[2][4] = min [c[2][2] + c[3][4] + p1 x p2 x p4

= 0 + 120 + 3 x 4 x 6

= 120 + 72

= 192

when k = 3

c[2][4] = min [c[2][3] + c[4][4] + p1 x p3 x p4

= 60 + 0 + 3 x 5 x 6

= 60 + 90

= 150

As c[2][4] = 150 is min when

s[2][4] = 3.

when length = 3

k = 1, 2, 3

c[1][4] = min [c[1][1] + c[2][4] + p0 x p1 x p4

= 0 + 150 x 2 x 3 x 6

= 150 x 36

= 540

c[1][4] = min [c[1][2] + c[3][4] + p0 x p2 x p4

= 24 + 120 + 2 x 4 x 6

= 144 + 48

= 192

$c[1][4] = \min [c[1][3] + c[4][4] + p_0 \times p_3 \times p_4$

$= 90 + 0 + 2 \times 5 \times 6$

$= 90 + 60$

$= 150$

As $c[1][4] = 150$ is min so $s[1][4] = 3$.

c[i][j]

i \ j	1	2	3	4
1	0	24	90	150
2		0	60	150
3			0	120
4				0

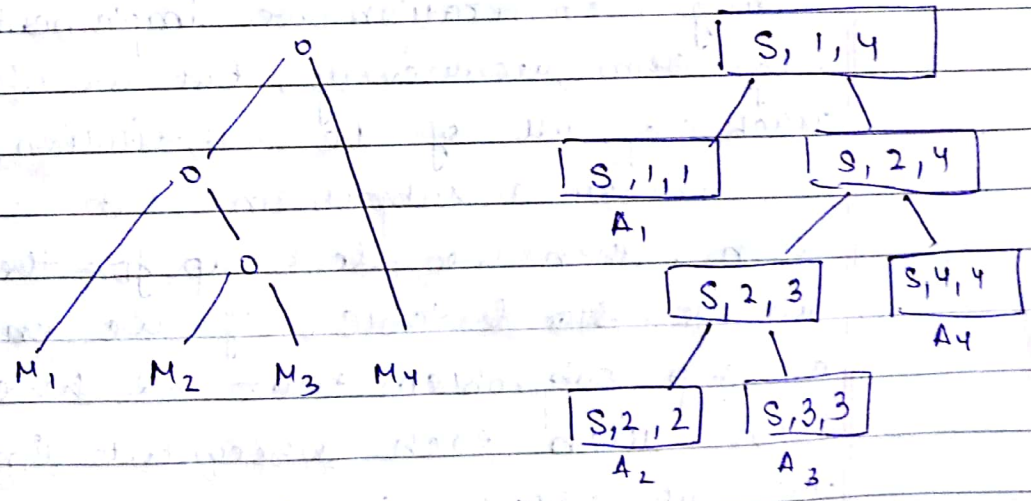
s[i][j]

i \ j	1	2	3	4
1	0	1	1	3
2		0	2	3
3			0	3
4				0

$M_1 M_2 M_3 M_4$.

$(M_1 M_2 M_3) M_4$

$((M_1)(M_2 M_3) M_4)$



Cost: $M_2 M_3 = (3 \times 4) \times (4 \times 5)$

$= 3 \times 4 \times 5 = 60$

$M_1 (M_2 M_3) = (2 \times 3)(3 \times 4 \times 5)$

$= 2 \times 3 \times 5 = 30$

$M_1 (M_2 M_3) M_4 = (2 \times 3 \times 5)(5 \times 6)$

$= 2 \times 5 \times 6$

$= 60$

Total cost = $60 + 30 + 60 = 150$

Longest Common Subsequences.

Longest Common Subsequences.

If a set of sequences are given the L.C.S problem is to find out the common subsequence of all the sequences that is of maximal length.

The LCS problem are classic computer science problem, the basis of data compaision programs and has application in bioinformatics

Naive Method.

Let x be a subsequence of length m and Y be a subsequence of length n check for every subsequence of x whether it is a subsequence of Y and return LCS found.

So complexity of naive algorithm method take $O(n2^m)$ times.

Complexity = Time complexity = $O(m+n)$
Space = $O(mn$.

```
LCS (X, Y)
1.  m ← length[X]
2.  n ← length[Y]
3.  for i ← 1 to m
4.      do c[i,0] ← 0
5.  for j ← 1 to n
6.      do c[0,j] ← 0
7.  for i ← 1 to m
8.  do for j ← 1 to m
9.  do if xi = yj
10.         then c[i,j] ← c[i-1, j-1] + 1
11.              b[i,j] ← "↖"
12.         else if c[i-1, j] ≥ c[i, j-1]
13.              then 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 and b.

PRINT LCS (b, xi, i, j)
1.  if i = 0 and j = 0
2.      then return
3.  if b[i, j] = "↖"
4.      then PRINT LCS (b, xi, i-1, j-1)
5.           print xi
6.  else if b[i, j] = "↑"
7.      then PRINT LCS (b, xi, i-1, j)
8.      else PRINT LCS (b, xi, i-1, j-1)
```

Ques

A = a b c b b a c

B = b c b a a b

i / j		a		b		c		b		b		a		c
		0		0		0		0		0		0		0
b	0	↑ 0	↖	1	←	1	↖	1	↖	1	←	1	←	1
c	0	↑ 0		1 ↑	↖	2	←	2	←	2	←	2	↖	2
b	0	↑ 0	↖	1		2 ↑	↖	3	↖	3	←	3	←	3
a	0	↖ 1		1 ↑		2 ↑		3 ↑		3 ↑	↖	4	←	4
a	0	↖ 1		1 ↑		2 ↑		3 ↑		3 ↑	↖	4		4 ↑
b	0	1 ↑	↖	2		2 ↑	↖	3	↖	4		4 ↑		4 ↑

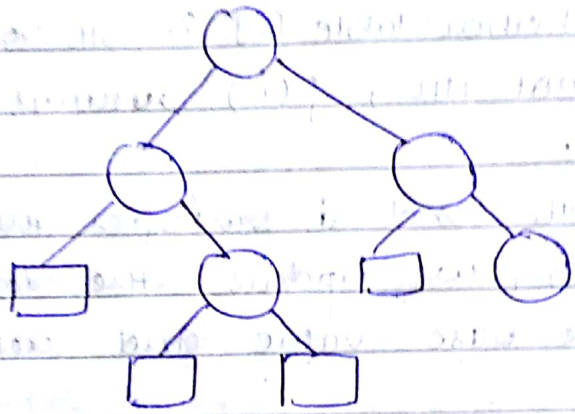
b c b a

longest common sub-sequence = bcba.

of length = 4.

Optimal Binary search trees.

Optimal Binary search tree is a binary search tree for which the nodes are arranged on levels such that the tree cost is minimum.



The square represent the terminal nodes and also known as external nodes. These terminals represents the unsuccessful searches of the tree.

The round nodes represent the internal nodes, these are the successful searches of the tree that stored the actual key in the tree.

Internal nodes represents by n
External nodes represents by n-1

Complexity — $o(n^3)$

Given

Q

$n = 4$

$(a_1, a_2, a_3, a_4) = (do, if, int, while)$

$(P_1, P_2, P_3, P_4) = (3, 3, 1, 1)$

$(q_0, q_1, q_2, q_3, q_4) = (2, 3, 1, 1, 1)$

$0 \le i \le 4$

Formulas :

$w(i,i) = q(i)$

$c(i,i) = 0$

$r(i,i) = 0$

$$c(i,j) = \min_{i<k\le j} \{ c[i,k-1] + c[k,j] + w(i,j) \}$$

$w(i,j) = P(j) + q(j) + w(i,j-1)$

$r(i,j) = k$, $r(i,i+1) = i+1$

	0	1	2	3	4
0	$w_{00}=2$ $c_{00}=0$ $r_{00}=0$	$w_{11}=3$ $c_{11}=0$ $r_{11}=0$	$w_{22}=1$ $c_{22}=0$ $r_{22}=0$	$w_{33}=1$ $c_{33}=0$ $r_{33}=0$	$w_{44}=1$ $c_{44}=0$ $r_{44}=0$
1	$w_{01}=8$ $c_{01}=8$ $r_{01}=1$	$w_{12}=7$ $c_{12}=7$ $r_{12}=2$	$w_{23}=3$ $c_{23}=3$ $r_{23}=3$	$w_{34}=3$ $c_{34}=3$ $r_{34}=4$	w c X r
2	$w_{02}=12$ $c_{02}=19$ $r_{02}=1$	$w_{13}=1$ $c_{13}=12$ $r_{13}=2$	$w_{24}=5$ $c_{24}=8$ $r_{24}=3$	w c X r	X
3	$w_{03}=14$ $c_{03}=25$ $r_{03}=2$	$w_{14}=11$ $c_{14}=19$ $r_{14}=2$	w c X r	w c X r	X
4	$w_{04}=16$ $c_{04}=32$ $r_{04}=2$	w X c r	w c X r	w X c r	X

$w[i,j] = p(j) + q(j) + w(i, j-1)$

$w(0,1) = p(1) + q(1) + w(0,0)$

$= 3 + 3 + 2$

$= 8$

$c[i,j] = \min_{i<k\leq j} \{c[i, k-1], c[k,j] + w(i,j)\}$

$c[0,1] = \min_{0<k\leq j} c[0,0], c[1,1] + w(0,1)$

$= 0 + 0 + 8.$

$r(i,j) = k = 1$

~~$w[1,2] = \min_{1<k\leq 2} c[1,1] + c[2,2] + w[1,2]$~~

~~$= 0 + 0 +$~~

$w(1,2) = p(2) + q(2) + w(i, j-1)$ ($w(1,1)$)

$= 3 + 1 + q(1)$

$= 3 + 1 + 3$

$= 7.$

$c[1,2] = \min_{1<k\leq 2} c[1,1] + c[2,2] + w(1,2)$

$= 7$

$r(i,j) = k = 2.$

$w(2,3) = p(3) + Q(3) + w(2,2)$

$= 1 + 1 + 1$

$= 3$

$c(2,3) = \min_{2<k\leq 3} c[2,2] + c[3,3] + w(2,3)$

$0 + 0 + 3$

$r(2,3) = 3.$

$w(3,4) = p(4) + Q(4) + w(3,3)$

$= 1 + 1 + 1$

$= 3$

$c(3,4) = \min_{3<k\leq 4} c(3,3) + c(4,4) + w(3,4)$

$= 0 + 0 + 3$

$= 3$

$r(3,4) = 4.$

$w(0,2) = p(2) + Q(2) + w(0,1)$

$= 3 + 1 + 8$

$= 12$

$C(0,2) = \min_{0<k\leq 2} C(i,k-1) + C(k,j) + w(i,j)$

$k = 1, 2.$

when k = 1

$c(0,2) = \min c(0,0) + c(1,2) + w(0,2)$

$= 7 + 12 = 19$

when k = 2.

$c(0,2) = \min c(0,1) + c(2,2) + w(0,2)$

$= 8 + 0 + 12$

$= 20.$

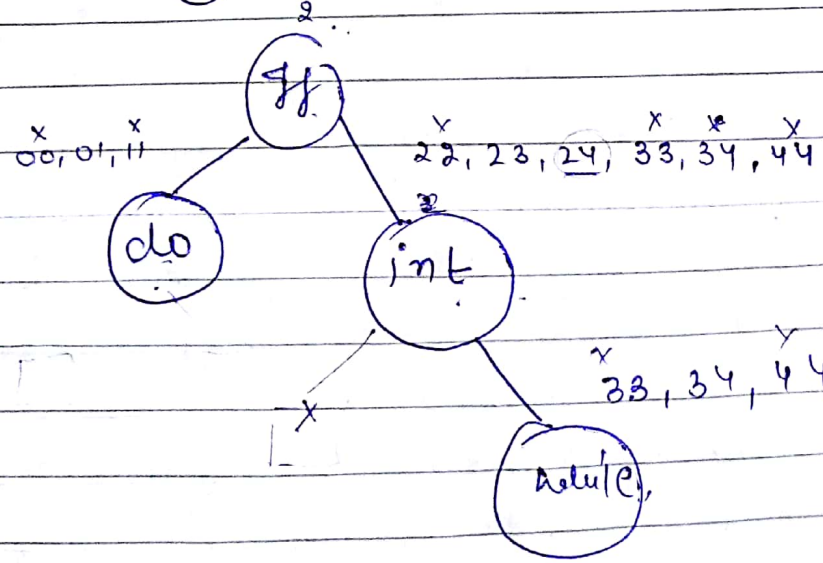
$r(0,2) = 1$

Now we find the maximum cost.

$c(0,4) = 32$

$r(0,4) = 2.$

if is our root node.



Cost = 3 x 1 + 3 x 1 + 3 x 1 + 3 x 1 + (2 x 3 + 2 + 3 x 1)

= 3 + 3 + 3 + 1 + 12 + 3

= 10 + 15.

25

0-1 Knapsack Problem

We are given n no. of objects with weight w_i and profit p_i where i varies from 1 to n and also a Knapsack with capacity M.

The problem is, we have to fill the bag with the help of n objects and resulting the profit has to be maximum.

This problem is similar to ordinary 0-1 Knapsack problem but we may not take a fraction of object.

$$\text{Maximum} \sum_{1}^{n} p_i x_i$$

$$\text{Subjected to} \sum_{1}^{n} w_i x_i \leq M$$

Ques

where $1 \leq i \leq n$

$$x_i = 0 \text{ or } 1$$

Step

- Let S^i be a pair of (p_i, w_i) where p_i is profit and w_i is weight of an object.
- Initially $S^0 = \{(0,0)\}$
- Compute S^{i+1} = Merge S^i and S_1^{i+1}

Step - 2

Generate the no. of decision sequence using the following formula.

Rule Purging Rule / Dominance Rule.

If S^{i+1} contains (P_j, w_j) and (P_k, w_k)

if $P_k \geq P_j$ and $w_j \geq w_k$

then (P_j, w_j) can be eliminated.

[Remove less profit and more weight].

K ⟹ Finding x_i values.

1. Set $x_n = 0$ if $(P, w) \in S^{n-1}$
2. Else Set $x_n = 1$ and 3. $(P, w) = (P - P_n, w - w_n)$
4. $n = n-1$, 5. repeat from step 1 until $n = 0$

Ques $(p^1, p^2, p^3) = [1, 2, 5]$

$[w_1, w_2, w_3] = [2, 3, 4]$

$M = 6 \qquad n = 3.$

$S^0 = \{(0,0)\}$

$S_1^0 = \{P_1, w_1\}$

$= \{1, 2\}$

$S^1 = S^0 + S_1^0 = \{(0,0)(1,2)\}$

$S_1^1 = \{P_2, w_2\} = \{2, 3\}$

$= \{(2,3)(3,5)\}$

$S^2 = S^1 + S_1^1$

$= \{(0,0)(1,2)(2,3)(3,5)\}$

$S_1^2 = \{P_3, w_3\} = \{5,4\}$

$= \{(5,4),(6,6),(7,7),(8,9)\}$

$S^3 = S^2 + S_1^2$

$= \{(0,0)(1,2)(2,3)(3,5)(5,4)(6,6)$
$(7,7)(8,9)\}$

3,5 will remove since it fails the purging rule.

$\therefore p_j \leq p_k \quad w_k \leq w_j$

$3 \leq 5 \quad 4 \geq 5$ Not satisfy.

So (3,5) discard.

$S^3 = \{(0,0)(1,2)(2,3)(3,5)(5,4)(6,6)(7,7)$
$(8,9)\}$

Now check the bag capacity $M = 6$.

So (6,6) is selected.

Now $(6,6) \in S^{n-1}$

$(6,6) \notin S^2$

Not belong

So $x_3 = 1$

Now $(P, w) = (P - P_n)\ (w - w_n)$

$= (6 - 5)\ (6 - 4)$

$= (1, 2)$

Now check $(1, 2) \in S^{1-1}$

$(1, 2) \in S^1$

Belong

so $x_2 = 0$.

Now $(P, w) = (P - P_n)\ (w - w_n)$

$= (6 - 2)\ (6$

Now $(P, w) = (P - P_n)\ (w - w_n)$

$= (1 - 1)\ (2 - 2)$

$= (0, 0)$

$(0, 0) \in S^{1-1}$

$(0, 0) \notin S^0$

Not Belong.

$x_1 = 1$

x_1	x_2	x_3
1	0	1

w_1 and w_3 are put in the gunny bag to get the maximum profit.

$1 + 5 = 6$

Binomial Coefficient computation through dynamic programming.

→ Computing Binomial coefficient is a non optimization problem but can be solved using dynamic programming.

Binomial coefficient are represented by ${}^{n}C_{K}$ and can be used to represent the coefficient of a Binomial:

$$(a+b)^n = {}^{n}C_0 a^n . b^0 + \cdots + {}^{n}C_K a^{n-K} b^K + \cdots + {}^{n}C_n b^n$$

The recursive relation is defined by

$$C(n,k) = C(n-1,k-1) + C(n-1,k)$$

where $n \geq K \geq 0$

$$C(n,0) = C(n,n) = 1$$

Dynamic ~~prog~~ algorithm constructs a table, with the ~~first~~ column and diagonal filled out using IC

Algorithm

```
BCof (int n, int K)
{
  If (K=0 || K = n)
  return 1;
  else
   return BCof (n-1, K-1) + Bof(n-1, K)
}
```

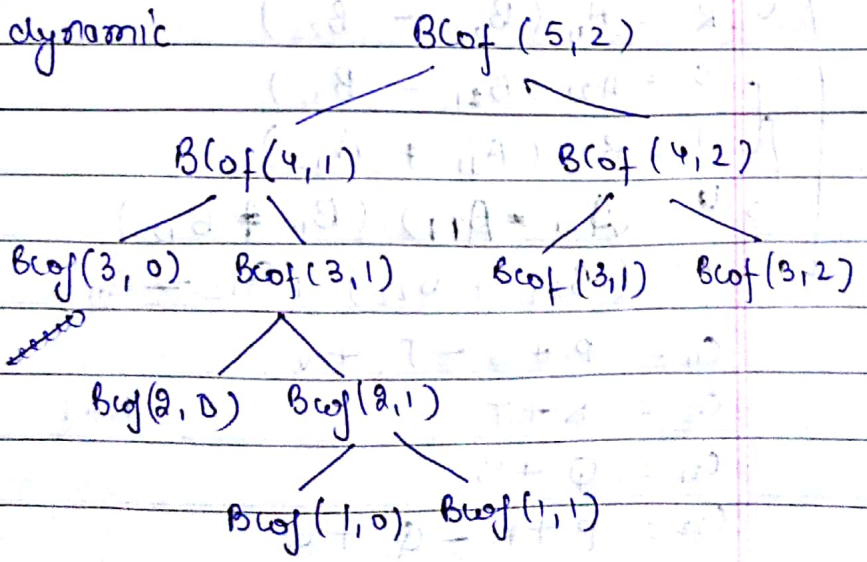
Ques Find ${}^5C_2$? By dynamic algo.

Normally, ${}^5C_2 = {}^4C_1 + {}^4C_2$

$$= \frac{4!}{1!(4-1)!} + \frac{4!}{2!\ 2!}$$

$$= 10$$

But in dynamic



DP Table

n↓ k→	0	1	2	3	4	5
0	1					
1	1	1				
2	1	2	1			
3	1	3	3	1		
4	1	4	6	4	1	
5	1	5	10	10	5	1

we have to find $^5C_2$.

$= 10$

Floyd Warshall Algorithm

F.W.A is an algorithm for finding the shortest path of in a weighted graph with positive and negative edge weights. The algorithm is used to find the ~~smallest~~ length of shortest path b/w all the pair of vertices.

At first the output matrix is same as the given cost matrix of the graph after that the output matrix will be updated with all vertices, and K as a intermediate vertices

$$d_{ij}^{k} = \begin{cases} w_{ij} & \text{when } k = 0 \\ \min\left(d_{ij}^{k-1},\ d_{ik}^{k-1} + d_{kj}^{k-1}\right) & k \geq 1 \end{cases}$$

$$D^{n} = d_{ij}^{n}$$

Distance Matrix

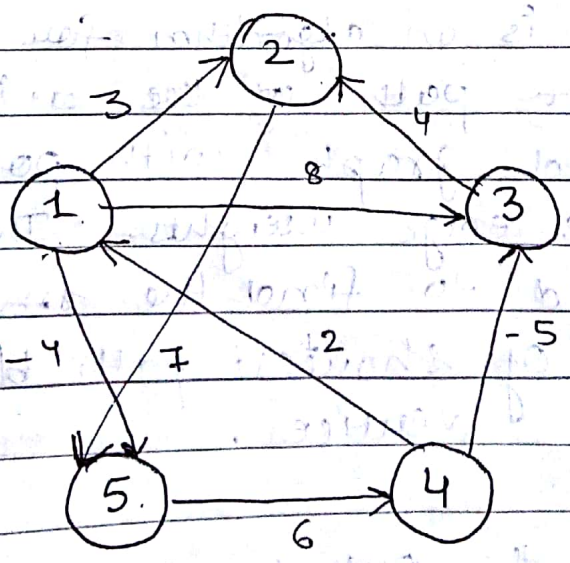
$$D^{0} = d_{ij}^{0} = \begin{cases} w_{ij} & \because i \neq j \\ 0 & \text{if } i = j \\ \infty & \text{otherwise} \end{cases}$$

Precedence matrix $\Rightarrow$ $P_{ij}^{k} = P_{kj}^{k-1}$

if $D_{ij}^{k} \neq P_{ij}^{k-1}$

$= P_{ij}^{k-1}$ otherwise

Ques: Apply Floyd Warshall algorithm for constructing shortest path.



$D^0 =$

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	∞	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0

$P^0 =$

.	1	2	3	4	5
1	0	1	1	0	1
2	0	0	0	0	2
3	0	3	0	0	0
4	4	0	4	0	0
5	0	0	0	5	0

$$D_{ij}^{k} = \min\left(D_{ij}^{k-1},\ D_{ik}^{k-1} + D_{kj}^{k-1}\right)$$

D^1

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	5	-5	0	-2
5	∞	∞	∞	6	0

	1	2	3	4	5
1	0	1	1	0	1
2		0		0	
3			0		
4				0	
5					0

$D_{12} = \min(D_{12}^0, D_{11}^0 + D_{12}^0)$

$= \min(3, 0 + 3)$

$= 3, 3.$

$D_{13} = \min(D_{13}^0, D_{11}^0 + D_{13})$

$= \min(8, 0 + 8)$

$= 8, 8.$

$= 8$

$D_{14} = \min(D_{14}^0, D_{11}^0 + D_{14})$

$= \min(\infty, 0 + \infty)$

$= \infty$

$D'_{15} = D^{\circ}_{15} + D_{11} + D_{15}$

$= -4 + 0 - 4$

$= -4$

$D'_{21} = D^{\circ}_{21} + D_{21} + D_{12}$

$= \infty + \infty + 3$

$= \infty$

$D_{23} = D^{\circ}_{23} + D_{21} + D_{13}$

$\infty + \infty + 8$

∞

$D'_{24} = D^{\circ}_{24} + D_{21} + D_{14}$

$1 + \infty + \infty$

1

$D_{31} = D^{\circ}_{31} + D_{31} + D_{13}$

$= \infty + \infty + 0$

$= \infty$

$D_{32} = D^{\circ}_{32} + D_{31} + D_{12}$

$= 4 + \infty + 3$

$= 4$

35

All D are calculated in the same manner.

$$D_{11}^{0} = D_{11}^{1}$$

$$\text{or} \quad P_{ij}^{0} = P_{ij}^{1}$$

$$D_{24}^{0} \neq D_{24}^{1}$$

$$P_{24}^{1} = P_{1j}^{k} = P_{kj}^{k-1} = P_{14}^{0}$$

Algorithm

```
#  FLOYD WARSHAL (W)
1.  n ← row[W]
2.  D^0 ← W
3.  for k ← 1 to n
4.      do for i ← 1 to n
5.      do for j ← 1 to n
6.      do d_ij^k ← [min d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1)]
7.  return D^n.
```

Complexity → $O(V^3)$

FIRST TERM EXAMINATION [SEP-2017]
FIFTH SEMESTER [B.TECH]
ALGORITHM DESIGN AND ANALYSIS
[ETCS–301]

Time : 1.30 hrs. **M.M. : 30**

Note: Attempt any three questions including Q.1 is compulsory.

Q. 1. (a) Define problem statement, problem instance and problem space with reference to algorithm with an example. **(2)**

Ans. A problem statement is a concise description of an issue to be addressed or a condition to be improved upon. A simple and well-defined problem statement will be used by the project team to understand the problem and work toward developing a solution.

In computational complexity theory, a problem refers to the abstract question to be solved. In contrast, an instance of this problem is a rather concrete utterance, which can serve as the input for a decision problem.

Problem Space refers to the entire range of components that exist in the process of finding a solution to a problem.

Q .1. (b) Define Algorithm and Asymptotic notations. **(2)**

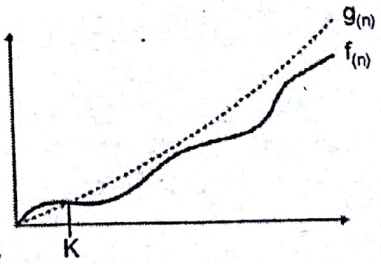
Ans. Algorithm: An algorithm (pronounced AL-go-rith-um) is a procedure or formula for solving a problem, based on conductiong a sequence of specified actions. A computer program can be viewed as an elaborate algorithm. In mathematics and computer science, an algorithm usually means a small procedure that solves a recurrent problem.

The commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- O Notation
- Ω Notation
- θ Notation

Big Oh Notation,O

The notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

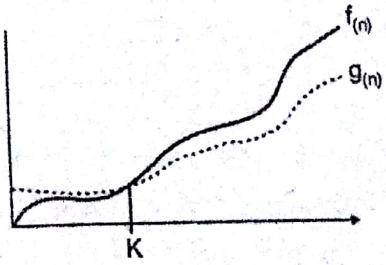


For example, for a function ***f*(n)**

$O(f(n)) = \{ g(n)$: there exists $c > 0$ and n_0 such that $f(n) \leq c.g(n)$ for all $n > n_0$. $\}$

Omega Notation, Ω

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.



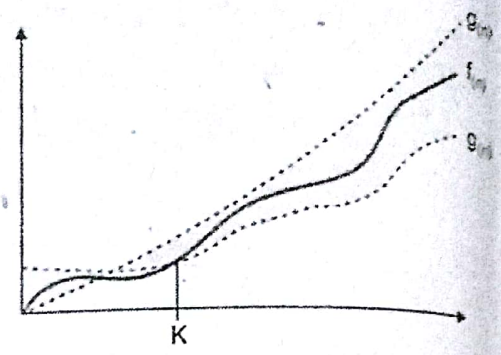
For example, for a function ***f*(n)**

$\Omega(f(n)) \geq \{g(n)$: there exists $c > 0$ and n_0 such that $g(n) \leq c.f(n)$ for all $n > n_0$. $\}$

Theta Notation, θ

The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows -

$\theta(f(n)) = \{g(n)$ if and only if $g(n) = O(f(n))$ and $g(n) = \Omega(f(n))$ for all



Q. 1. (c) List out Approaches to design an algorithms known to you. (2)

Ans.

- Divide and Conquer Method. In the divide and conquer approach, the problem is divided into several small sub-problems.
- Greedy Method. In greedy algorithm of optimizing solution, the best solution is chosen at any moment.
- Dynamic Programming.
- Backtracking Algorithm.
- Branch and Bound.
- Linear Programming.

Q. 1. (d) How correctness of algorithm is checked? (2)

Ans. The main steps in the formal analysis of the correctness of an algorithm are:

- Identification of the properties of input data (the so-called problem's preconditions).

Identification of the properties which must be satisfied by the output data (the so-called problem's postconditions).

- Proving that starting from the preconditions and executing the actions specified in the algorithms one obtains the postconditions.

When we analyze the correctness of an algorithm a useful concept is that of state. The algorithm's state is the set of the values corresponding to all variables used in the algorithm

Q. 1. (e) State master method to solve a recurrence relation with all the cases. (2)

Ans. Master Method is a direct way to get the solution. The master method works only for following type of recurrences or for recurrences that can be transformed to following type.

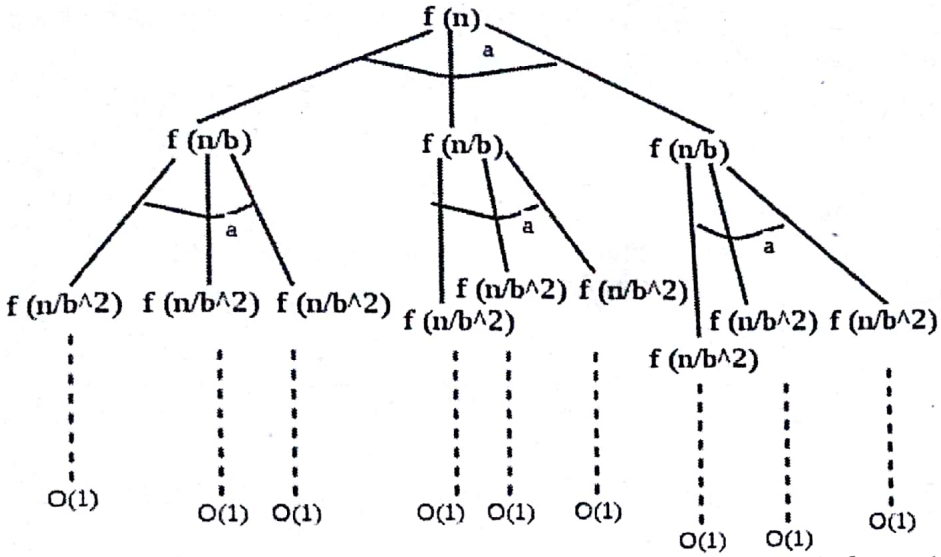
$T(n) = aT(n/b) + f(n)$ where $a >= 1$ and $b > 1$

There are following three cases:

1. If $f(n) = \Theta(n^c)$ where $c < \text{Log}_b a$ then $T(n) = \Theta(n^{\text{Log}_b a})$
2. If $f(n) = \Theta(n^c)$ where $c = \text{Log}_b a$ then $T(n) = \Theta(n^c \text{Log } n)$
3. If $f(n) = \Theta(n^c)$ where $c > \text{Log}_b a$ then $T(n) = \Theta(f(n))$

$af(n/b) \leq c f(n)$ for $c < 1$ regularity condition

How does this work?: Master method is mainly derived from recurrence tree method. If we draw recurrence tree of $T(n) = aT(n/b) + f(n)$, we can see that the work done at root is $f(n)$ and work done at all leaves is $\Theta(n^c)$ where c is $\text{Log}_b a$. And the height of recurrence tree is $\text{Log}_b n$



In recurrence tree method, we calculate total work done. If the work done at leaves is polynomially more, then leaves are the dominant part, and our result becomes the work done at leaves (Case 1). If work done at leaves and root is asymptotically same, then our result becomes height multiplied by work done at any level (Case 2). If work done at root is asymptotically more, then our result becomes work done at root (Case3).

Examples of some standard algorithms whose time complexity can be evaluated using Master Method

Merge Sort: $T(n) = 2T(n/2) + \Theta(n)$. It falls in case 2 as c is 1 and $Log_b a]$ is also 1. So the solution is $\Theta(n Logn)$

Binary Search: $T(n) = T(n/2) + \Theta(1)$. It also falls in case 2 as c is 0 and $Log_b a$ is also 0. So the solution is $\Theta(Logn)$

Notes:

1. It is not necessary that a recurrence of the form $T(n) = aT(n/b) + f(n)$ can be solved using Master Theorem. The given three cases have some gaps between them. For example, the recurrence $T(n) = 2T(n/2) + n/Logn$ cannot be solved using master method.

2. Case 2 can be extended for $f(n) = \Theta(n^c Log^k n)$

If $f(n) = \Theta(n^c Log^k n)$ for some constant $k >= 0$ and $c = Log_b a$, then $T(n) = \Theta(n^c Log^{k+1} n)$

Q. 2. (a) Can master method solve the recurrence relation t(n) = 3T(n/4) + nl gn? If "no" explain,if "yes" solve it. **(5)**

Ans.

Here $a = 3, \quad b = 4, \quad f(n) = n \lg n.$

$$n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$$

$$f(n) = \Omega(n^{\log_4 3 + \varepsilon}) \text{ where } \varepsilon = 0.2$$

Case 3 applies, now for regularity condition i.e.,

$$af\left(\frac{n}{b}\right) \le c f(n)$$

$$3\left(\frac{n}{4}\right)\log\left(\frac{n}{4}\right) \le \left(\frac{3}{4}\right) n \lg n = cf(n) \text{ for } c = \frac{3}{4}$$

Therefore, the solution is $T(n) = \theta(n \lg n)$.

A globally optimal soln is a feasible soln with an objective value that is as good or better than all other feasible solns to the model

Q .2. (b) Can master method solve the recurrence relation T(n) = 2T(n/2) + nlgn? If "no" explain,if "yes" solve it. (5)

Ans. Yes it can be solved by master method a = 2 b = 2 f(n) = nlogn

$$n^{\log_b a} = n^{\log_2 2} = n \text{ but } f(n) = n\log n$$

therefore T(n)= nlogn

Q. 3. (a) Discuss the essence of Dynamic Programming. (5)

Ans. Greedy algorithms makes the best local decision at each step, but may not guarantee the global optimal. Exhaustive search algorithms explore all possibilities and always select the optimal, but the cost is too high.

Thus leads to dynamic programming: search all possibilities (correctness) while restoring results to avoid recomputing (efficiency).

Dynamic programming can efficiently implement recursive algorithm by storing partial results.

If recursive algorithm computes the same subproblems over and over and over again, storing the answer for each subproblem in a table to look up instead of recompute.

Generally for optimization problem for left-to right-order objects such as characters in string, elements of a permutation, points around a polygon, leaves in a search tree, integer sequences. Because once the order is fixed, there are relatively few possible stopping places or states.

Use when the problem follow the *principle of optimality*: Future decisions are made based on the overall consequences of previous decisions, not the actual decisions themselves.

Independent subproblems: solution to one subproblem doesn't affect solution to another subproblem of the same problem. Using resources to solve one subproblem renders them unavailable to solve the other subproblem. (longest path)

Q. 3. (b) Give the optimal parenthesis for matrix multiplication problem with input of 6 matrix of size: << 4, 10, 3, 12, 20, 7 >>.

Ans. Refer Q. 4. (b) of End Term Examination 2017.

Q. 4. (a) Give the problem statement of 0/1 knapsack. Consider the following input instance of 0/1 knapsack problem;

3 items weight 20,30,40 units and profit associated with them 10,20,50 units respectively with knapsack of capacity 60 units.Solve it using dynamic programming approach. (5)

Ans. Problem can be solved in this way

0–1 Knapsack Problem

value [] = {60, 100, 120};
weight [] = {10, 20, 30};
W = 50;

Solution: 220

Weight = 10; Value = 60;
Weight = 20; Value = 100;
Weight = 30; Value = 120;
Weight = (20 + 10); Value = (100 + 60)
Weight = (30 + 10); Value = (120 + 60);
Weight = (30 + 20); Value = (120 + 100);
Weight = (30 + 20 + 10) > 50.

Q. 4. (b) Write down the psuedocode of insertion sort and analyze its complexity in all cases of input instance. (5)

Ans. Insertion-Sort (A)

```
1. for j ← 2 to length [A]
2.     do key ← A[j]
3.         ▷ Insert A[j] into the sorted sequence A[1..j – 1]
4.         i ← j – 1
5.         while i > 0 and A[i] > key
6.             do A [i + 1] ← A[i]
7.                 i ← i – 1.
8.         A[i + 1] ← key
```

Analysis of Insertion Sort

The time taken by INSERTION - SORT procedure depends on the input. We start presenting the INSERTION - SORT procedure with the time "cost" of each statement and the number of time each statement is executed.

Insection-Sort (A)	Cost	times
1. for $j \leftarrow 2$ to length [A]	c_1	n
2. do key $\leftarrow A[j]$	c_2	$n-1$
3. ▷insert $A[j]$ into the sorted sequence $A[i..j-1]$	0	$n-1$
4. $i \leftarrow j-1$	$c4$	$n-1$
5. While $i > 0$ and $A[i] >$ key	c_5	$\sum_{j=2}^{n} t_j$
6. do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7. $i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8. $A[i+1] \rightarrow$ key	c_8	$n-1$

The running time of the algorithm is the sum of running times for each statement executed.

To compute T(n) the running time, we sum the product of the cost and time columns.

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Best case occurs if the array is already sorted. For each $j = 2, 3, \ldots n$, we find that $A[i] \leq$ key in line 5 when $i = j - 1$.

$$\text{Thus } t_j = 1 \text{ for } j = 2, 3, \ldots\ldots n$$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$

$$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

$= an + b$, where a and b are constants.

$=$ linear function of $n = O(n)$.

In worst case, the array is in reverse order. We must compare each elements A[j] with each element in the entire sorted, subarray $A[1...j-1]$ and $t_j = j$ for $j = 2, 3, ...n$.

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} j + c_6 \sum_{j=2}^{n} (j-1) + c_7 \sum_{j=2}^{n} (j-1) + c_8(n-1)$$

$$= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$

$$\left(\because \sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \text{ and } \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}\right)$$

$$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8)$$

$= an^2 + bn + c$ for a, b and c are constant.

$=$ quadratic function of n.

$= O(n^2)$

END TERM EXAMINATION DEC-2017
FIFTH SEMESTER [B.TECH]
ALGORITHM DESIGN AND ANALYSIS
[ETCS–301]

Time : 3 hrs. M.M. : 75

Note: *Attempt any five questions including Q.1 is compulsory.*

Q. 1. (a) Define Aymptotic notation?

Ans. Refer Q. 1. (b) of First Term Examination 2017.

Q. 1. (b) What is Substitution method ?

Ans. The substitution method for solving recurrences is famously described using wo steps:

- **1:** Guess the form of the solution
- **2:** Use induction to show that the guess is valid

Determine a tight asymptotic lower bound for the following recurrence

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

Let us guess that $T(n) = n^2 lg(n)$. Therefore, our induction hypothesis is there exists c and an n_0 such that

$$T(n) \geq cn^2 lg(n)$$

$\delta\, n_0 > n$ and $c > 0$.

For the **base case**($n = 1$). we have $T(1) = 1 > c1^2 lg1$. This is true for all $c > 0$.

Now for the inductive step, assume the hypothesis is true for m < n, thus:

$$T(m) \leq cm^2 lg(m)$$

So: $T(n) = 4T(n/2) + n^2 \geq 4c \frac{n^2}{4} \lg \frac{n}{2} + n^2 = cn^2 lg(n) - cn^2 lg(2) + n^2 = cn^2 lg(n)$ $+(1-c)n^2$ If we now pick as $c < 1$, then.

Q. 1. (c) Explain Hashing and elaborate its disadvantages over linear and binary search

Ans. Hashing is generating a value or values from a string of text using a mathematical function.

Hashing is one way to enable security during the process of message transmission when the message is intended for a particular recipient only. A formula generates the hash, which helps to protect the security of the transmission against tampering.

Hashing is also a method of sorting key values in a database table in an efficient manner.

Advantages: The main advantage of hash tables over other table data structures is speed. This advantage is more apparent when the number of entries is large (thousands or more).

Hash tables are particularly efficient when the maximum number of entries can be predicted in advance, so that the bucket array can be allocated once with the optimum size and never resized.

Disadvantages: Hash tables can be more difficult to implement than self-balancing binary search trees.

Although operations on a hash table take constant time on average, the cost of a good hash function can be significantly higher than the inner loop of the lookup algorithm for a sequential list or search tree.

For certain string processing applications, such as spell-checking, hash tables may be less efficient than tries, finite automata, or Judy arrays.

There are no collisions in this case.

The entries stored in a hash table can be enumerated efficiently (at constant cost per entry), but only in some pseudo-random order. In comparison, ordered search trees have lookup and insertion cost proportional to log(n), but allow finding the nearest key at about the same cost, and ordered enumeration of all entries at constant cost per entry.

Hash tables in general exhibit poor locality of reference—that is, the data to be accessed is distributed seemingly at random in memory. Hash tables become quite inefficient when there are many collisions.

Q. 1. (d) Differentiate between dynamic programming and divide and conquer approach.

Ans.

S.No.	Divide-and-conquer algorithm	Dynamic Programming
1.	Divide-and-conquer algorithms splits a problem into separate subproblems, solve the subproblems, and combine the results for a solution to the original problem. Example : Quick sort, Merge sort, Binary search.	Dynamic Programming splits a problem into subproblems, some of which are common, solves the subproblems, and combines the results for a solution to the original problem. Example : Matrix Chain Multiplication, Longest Common Subsequence.
2.	Divide-and-conquer algorithms can be thought of as top-down algorithms.	Dynamic programming can be thought of as bottom-up
3.	In divide and conquer, sub-problems are independent.	In Dynamic Programming, sub-problems are not independent.
4.	Divide & Conquer solutions are simple as compared to Dynamic programming	Dynamic Programming solutions can often be quite complex and tricky.
5.	Divide & Conquer can be used for any kind of problems.	Dynamic Programming is generally used for Optimization problems.
6.	Only one decision sequence is ever generated.	Many decision sequences may be generated.

Q. 1. (e) Explain the concept of overlapping subproblems.

Ans. A problem is said to have overlapping subproblems if the problem can be broken down into subproblems which are reused several times or a recursive algorithm for the problem solves the same subproblem over and over rather than always generating new subproblems.

For example, the problem of computing the Fibonacci sequence exhibits overlapping subproblems. The problem of computing the nth Fibonacci number F(n), can be broken down into the subproblems of computing F(n – 1) and F(n – 2), and then adding the two. The subproblem of computing F(n – 1) can itself be broken down into a subproblem that involves computing F(n – 2). Therefore the computation of F(n – 2) is reused, and the Fibonacci sequence thus exhibits overlapping subproblems.

A naive recursive approach to such a problem generally fails due to an exponential complexity. If the problem also shares an optimal substructure property, dynamic programming is a good way to work it out.

```
/* simple recursive program for Fibonacci numbers */
intfib(intn)
{
  if( n <= 1 )
    returnn;
  returnfib(n-1) + fib(n-2);
}
```

Q. 1. (f) What are the advantages of optimal binary search tree over binary search tree?

Ans. • Binary search trees are used to organize a set of keys for fast access: the tree maintains the keys in-order so that comparison with the query at any node either results in a match, or directs us to continue the search in left or right subtree.

• A balanced search tree achieves a worst-case time O(log n) for each key search, but fails to take advantage of the structure in data.

• For instance, in a search tree for English words, a frequently appearing word such as "the" may be placed deep in the tree while a rare word such as "machiocolation" may appear at the root because it is a median word.

• In practice, key searches occur with different frequencies, and an Optimal Binary Search Tree tries to exploit this non-uniformity of access patterns, and has the following formalization.

• The input is a list of keys (words) w1, w2, . . . , wn, along with their access probabilities p1, p2, . . . , pn. The prob. are known at the start and do not change.

• The interpretation is that word wi will be accessed with relative frequency (fraction of all searches) pi . The problem is to arrange the keys in a binary search tree that minimizes the (expected) total access cost.

Q. 1. (g) Explain 0-1 knapsack problem.

Ans. In 0-1 Knapsack, items cannot be broken which means the thief should take the item as a whole or should leave it. This is reason behind calling it as 0-1 Knapsack.

Hence, in case of 0-1 Knapsack, the value of x_i can be either *0* or *1*, where other constraints remain the same.

0 – 1 Knapsack cannot be solved by Greedy approach. Greedy approach does not ensure an optimal solution. In many instances, Greedy approach may give an optimal solution.

The following examples will establish our statement.

Example-1

Let us consider that the capacity of the knapsack is W = 25 and the items are as shown in the following table.

Item	A	B	C	D
Profit	24	18	18	10
Weight	24	10	10	7

Without considering the profit per unit weight (p_i/w_i), if we apply Greedy approach to solve this problem, first item *A* will be selected as it will contribute maximum profit among all the elements.

After selecting item *A*, no more item will be selected. Hence, for this given set of items total profit is *24*. Whereas, the optimal solution can be achieved by selecting items, *B* and C, where the total profit is 18 + 18 = 36.

Q. 1. (h) What are the elements of greedy strategy?

Ans. Elements of greedy strategy are:

1. Optimal substructure
2. 0/1 knapsack (Fractional)
3. Activity selection problem
4. Huffman coding

Q. 1. (i) Define Matroid with an example.

Ans. A matroid M = (S, I) is a finite ground set S together with a collection of sets $I \subseteq 2^S$, known as the independent sets, satisfying the following axioms: 1. If $I \in I$ and $J \subseteq I$ then $J \in I$. 2. If $I, J \in I$ and $|J| > |I|$, then there exists an element $z \in J \setminus I$ such that $I \cup \{z\} \in I$.

A second original source for the theory of matroids is graph theory.

Every finite graph (or multigraph) *G* gives rise to a matroid *M*(*G*) as follows: take as *E* the set of all edges in *G* and consider a set of edges independent if and only if it is a forest; that is, if it does not contain a simple cycle. Then *M*(*G*) is called a **cycle matroid**. Matroids derived in this way are **graphic matroids**. Not every matroid is graphic, but all matroids on three elements are graphic. Every graphic matroid is regular.

Q. 1. (j) Explain P and NP briefly.

Ans. P- Polynomial time **solving** . Problems which can be solved in polynomial time, which take time like O(n), O(n2), O(n3). Eg: finding maximum element in an array or to check whether a string is palindrome or not. so there are many problems which can be solved in polynomial time.

NP- Non deterministic Polynomial time solving. Problem which can't be solved in polynomial time like TSP(travelling salesman problem) or An easy example of this is subset sum: given a set of numbers, does there exist a subset whose sum is zero?.

but NP problems are checkable in polynomial time means that given a solution of a problem , we can check that whether the solution is correct or not in polynomial time.

Q. 2. (a) Explain quick sort and compute the analysis of quick sort perform quick sort on following data 14,15,25,28,30,32,35,40

What is the problem with quick sort ,if the data is already sorted? Discuss.

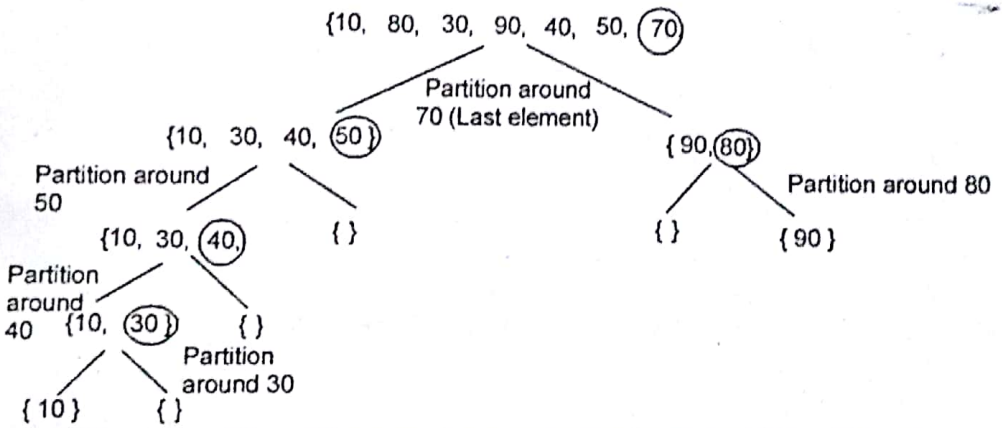
Ans. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Pseudo Code for recursive QuickSort function :

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
  if (low < high)
  {
    /* pi is partitioning index, arr[pi] is now
       at right place */
    pi = partition(arr, low, high);
    quickSort(arr, low, pi - 1);  // Before pi
    quickSort(arr, pi + 1, high); // After pi
  }
}
```



Partition Algorithm: There can be many ways to do partition, following pseudo code adopts the method given in CLRS book. The logic is simple, we start from the leftmost element and keep track of index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap current element with arr[i]. Otherwise we ignore current element.

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
  if (low < high)
  {
    /* pi is partitioning index, arr[p] is now
       at right place */
    pi = partition(arr, low, high);
    quickSort(arr, low, pi - 1);  // Before pi
    quickSort(arr, pi + 1, high); // After pi
  }
}
```

Array 14,15,25,28,30,32,35,40 can be sorted acc to the above method but we see that its already sorted than it shows the worst case of quicksort.

Worst-case running time: When quicksort always has the most unbalanced partitions possible, then the original call takes cncnc, n time for some constant ccc, the recursive call on n−1 n−1n, minus, 1elements takes c(n−1) c(n−1)c, left parenthesis, n, minus, 1, right parenthesis time, the recursive call on n−2n−2n, minus, 2 elements takes c(n−2)c(n−2)c, left parenthesis, n, minus, 2, right parenthesis time, and so on. Here's a tree of the subproblem sizes with their partitioning times:

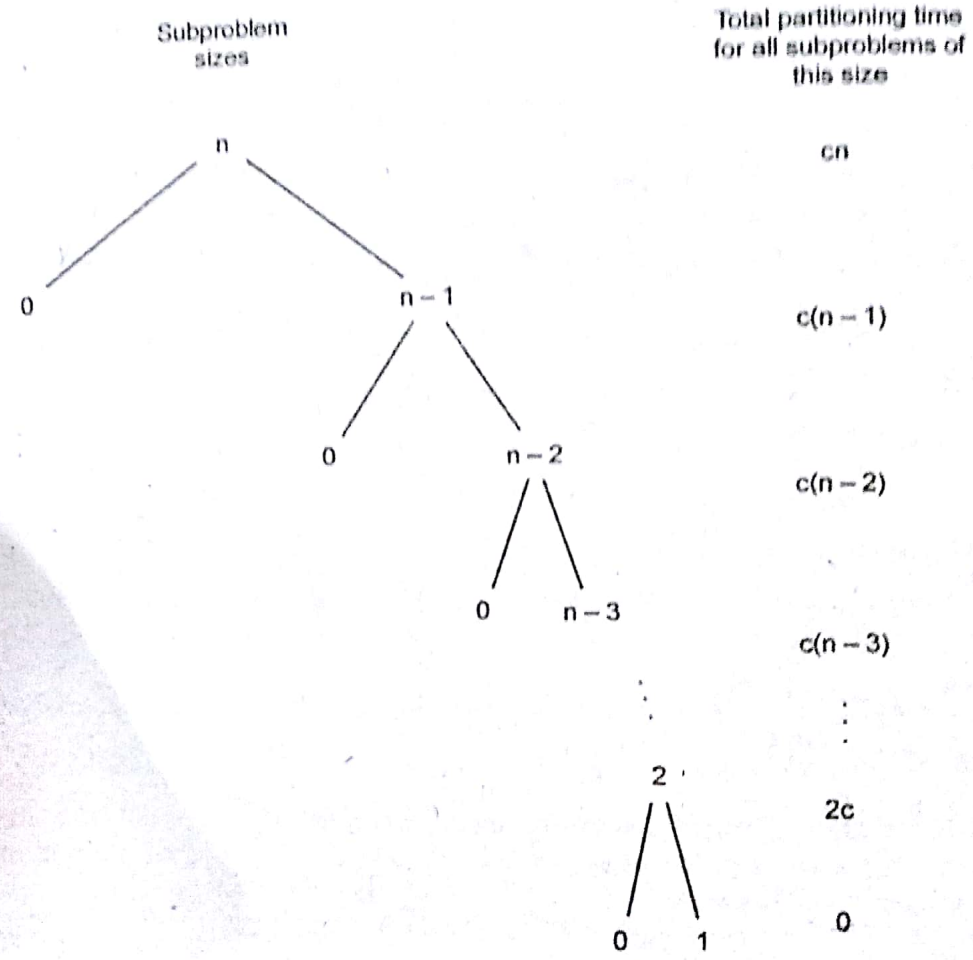


Diagram of worst case performance for Quick Sort

When we total up the partitioning times for each level, we get

\begin{aligned} cn + c(n-1) + c(n-2) + \cdots + 2c &= c(n + (n-1) + (n-2) + \cdots + 2) \\ &= c((n+1)(n/2) - 1) \ .

\end{aligned}$cn+c(n−1)+c(n−2)+...+2c$

$=c(n+(n−1)+(n−2)+...+2)=c((n+1)(n/2)−1)$.

The last line is because 1 + 2 + 3 + \cdots + n1+2+3+"+n is the arithmetic series, as we saw when we analyzed selection sort. (We subtract 1 because for quicksort, the summation starts at 2, not 1.) We have some low-order terms and constant coefficients, but when we use big-Θ notation, we ignore them. In big-Θ notation, quicksort's worst-case running time is \Theta(n^2) Θ (n2).

Best-case running time: Quicksort's best case occurs when the partitions are as evenly balanced as possible: their sizes either are equal or are within 1 of each other. The former case occurs if the subarray has an odd number of elements and the pivot is right in the middle after partitioning, and each partition has (n–1)/2(n–1)/2left parenthesis, n, minus, 1, right parenthesis, slash, 2elements. The latter case occurs if the subarray has an even number nnn of elements and one partition has n/2n/2n, slash, 2 elements with the other having n/2–1n/2–1n, slash, 2, minus, 1. In either of these cases, each partition has at most n/2n/2n, slash, 2 elements, and the tree of subproblem sizes looks a lot like the tree of subproblem sizes for merge sort, with the partitioning times looking like the merging times:

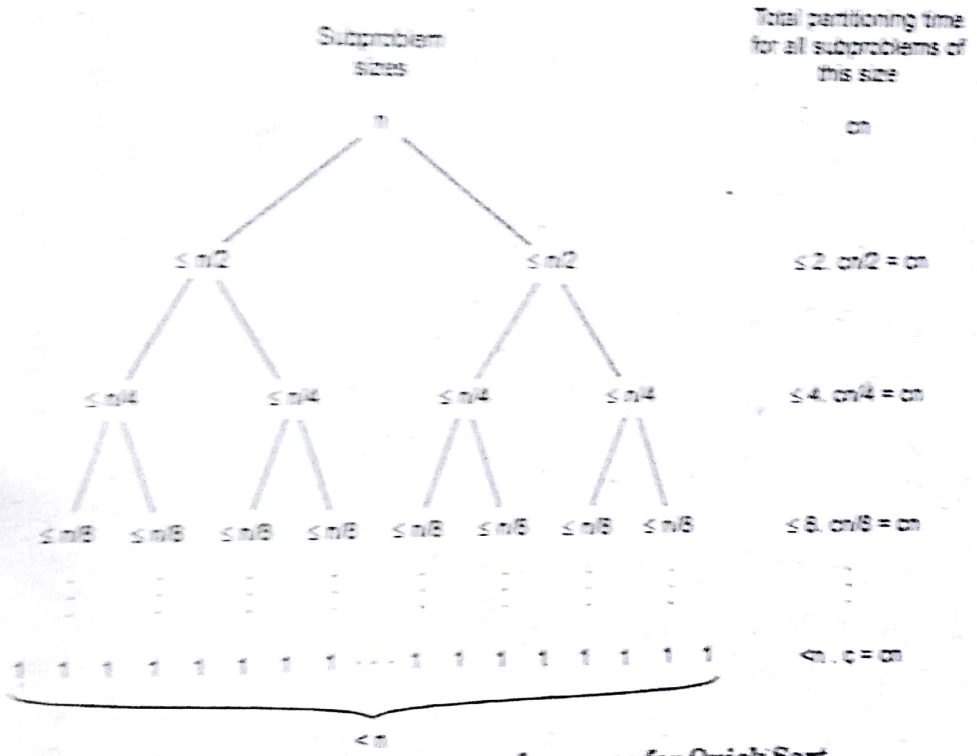


Diagram of best case performance for Quick Sort

Using big-Θ notation, we get the same result as for merge sort: \Theta(n \log_2 n)Θ (nlog2n).

Average-case running time: Showing that the average-case running time is also \Theta(n \log_2 n) Θ (nlog2 n) takes some pretty involved mathematics, and so we won't go there. But we can gain some intuition by looking at a couple of other cases to understand why it might be $O(n \log_2 n)O(nlog2n)O$, left parenthesis, n, log, start subscript, 2, end subscript, n, right parenthesis. (Once we have $O(n \log_2 n)O(nlog2n)O$, left parenthesis, n, log, start subscript, 2, end subscript, n, right parenthesis, the \Theta(n \log_2 n) Θ (nlog2n) bound follows because the average-case running time cannot be better than the best-case running time.) First, let's imagine that we don't always get evenly balanced partitions, but that we always get at worst a 3-to-1 split. That is, imagine that each time we partition, one side gets 3n/43n/43, n, slash, 4 elements and the other side gets n/4n/4n, slash, 4. (To keep the math clean, let's not worry about the pivot.) Then the tree of subproblem sizes and partitioning times would look like this:

Best-case running time: Quicksort's best case occurs when the partitions are as evenly balanced as possible: their sizes either are equal or are within 1 of each other. The former case occurs if the subarray has an odd number of elements and the pivot is right in the middle after partitioning, and each partition has (n–1)/2(n–1)/2left parenthesis, n, minus, 1, right parenthesis, slash, 2elements. The latter case occurs if the subarray has an even number nnn of elements and one partition has n/2n/2n, slash, 2 elements with the other having n/2-1n/2–1n, slash, 2, minus, 1. In either of these cases, each partition has at most n/2n/2n, slash, 2 elements, and the tree of subproblem sizes looks a lot like the tree of subproblem sizes for merge sort, with the partitioning times looking like the merging times:

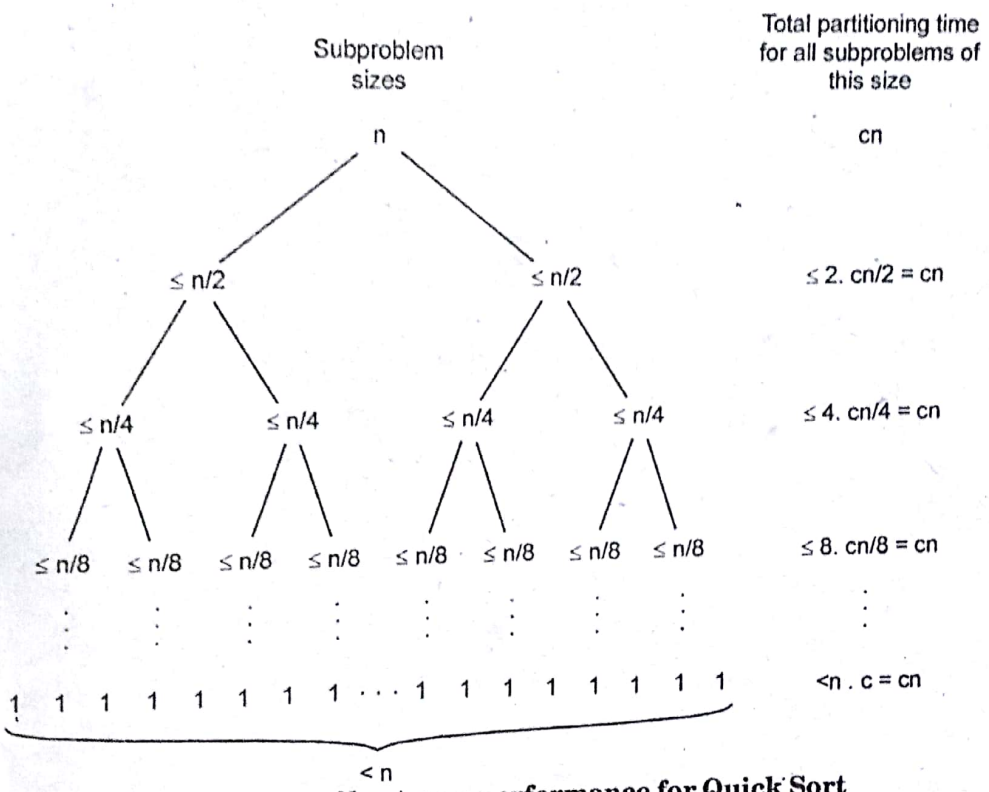
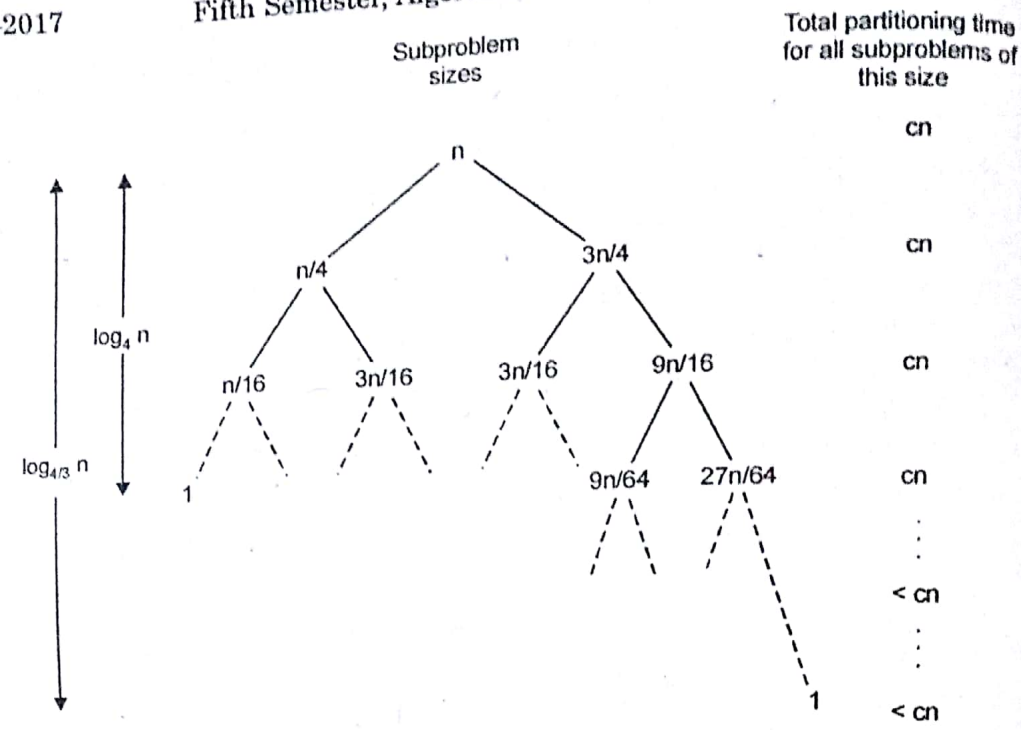


Diagram of best case performance for Quick Sort

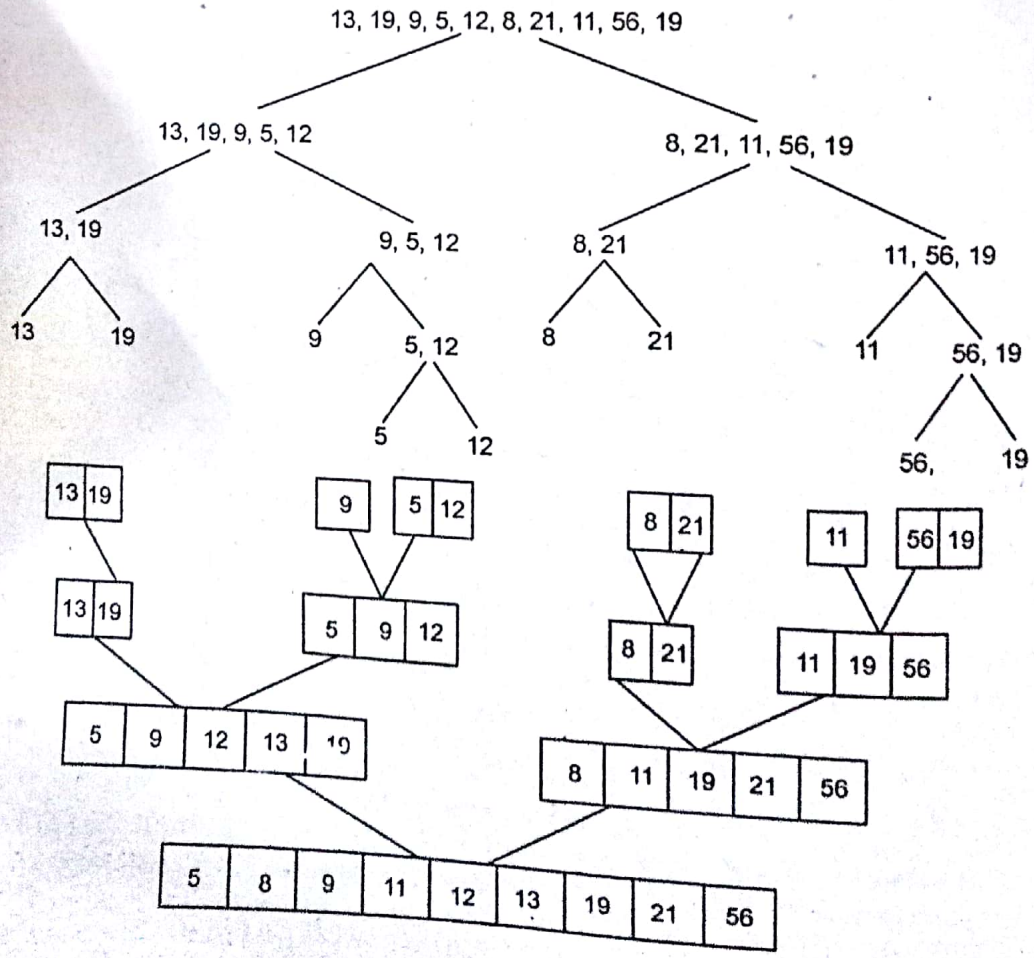
Using big-Θ notation, we get the same result as for merge sort: \Theta(n \log_2 n)Θ ($n$log2n).

Average-case running time: Showing that the average-case running time is also \Theta(n\log_2 n) Θ ($n$log2 n) takes some pretty involved mathematics, and so we won't go there. But we can gain some intuition by looking at a couple of other cases to understand why it might be O(n\log_2 n)O($n$log2n)O, left parenthesis, n, log, start subscript, 2, end subscript, n, right parenthesis. (Once we have O(n\log_2 n) O($n$log2n)O, left parenthesis, n, log, start subscript, 2, end subscript, n, right parenthesis, the \Theta(n\log_2 n) Θ ($n$log2n) bound follows because the average-case running time cannot be better than the best-case running time.) First, let's imagine that we don't always get evenly balanced partitions, but that we always get at worst a 3-to-1 split. That is, imagine that each time we partition, one side gets 3n/43n/43, n, slash, 4 elements and the other side gets n/4n/4n, slash, 4. (To keep the math clean, let's not worry about the pivot.) Then the tree of subproblem sizes and partitioning times would look like this:



Q. 2. (b) Sort the following numbers using merge sort
13,19,9,5,12,8,21,11,56,19

Ans.



Q.3. (a) Explain the data structure for disjoint set,its operations and its applications.

Ans. A disjoint set is a data structure which keeps track of all elements that are separated by a number of disjoint (not connected) subsets. With the help of disjoints sets, you can keep a track of the existence of elements in a particular group.

Let's say there are 6 elements A, B, C, D, E, and F. B, C, and D are connected and E and F are paired together. This gives us 3 subsets that have elements (A), (B, C, D), and (E, F).

Disjoint sets help us quickly determine which elements are connected and close and to unite two components into a single entity.

MAKE-SET(x): make a new set Si = {x} and add Si to S

• UNION(x, y): if x ∈ Si and y ∈ Sj , then S ← S – {Si , Sj} ∪ {Si ∪ Sj} – Representative of Si ∪ Sj is typically the representative of Si or Sj

• FIND(x): returns the representative of the set containing x

• Analyse complexity of sequence of m MAKE-SET, FIND and UNION operations, n of which are MAKE-SET operations

• Complexity is analysed in terms of n and m

Application:

MST-KRUSKAL(G, ẃ)

1 A ← ∅

2 for each vertex v ∈ V [G]

3 do MAKE-SET(v)

4 sort the edges of E into nondecreasing order by weight w

5 for each edge (u, v) ∈ E, taken in nondecreasing order by weight 6 do if FIND(u)

6 = FIND(v)

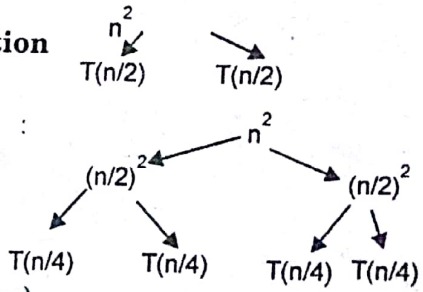
7 then A ← A ∪ {(u, v)}

8 UNION(u, v)

9 return A

Q. 3. (b) Solve the following recurrence relation

(i) T(n) = 4T(n/2) + n^2(using recurrence tree)



Ans.

$T(n) = n^2+n^2/4+n^2/4^2+$.....lg n times

$<= n^2(1/1-1/4)$

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

(ii) T(n) = 5T(n/4) + (n^3) (using master theorem)

Ans. By Master theorem

a = 5 b = 4f(n) = n^3

$n^{\log_b a} = n^{\log 45} = n^2$

$f(n) = n^3 = n \log^{a+\epsilon} = n^{2+1}$

therefore ∈ = 1

hence by case 3 sol is T(n) = theta(n^3)

Q. 4. What are the basic steps of dynamic programming?

Ans. Dynamic programming is used where we have problems, which can be divided into similar sub-problems, so that their results can be re-used. Mostly, these algorithms are used for optimization. Before solving the in-hand sub-problem, dynamic algorithm will try to examine the results of the previously solved sub-problems. The solutions of

sub-problems are combined in order to achieve the best solution.

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Q. 4. (b) Find optimal parenthesization of a matrix chain product whose sequence of dimension is< <4,10,3,12,20,7>

Ans. Sequence of dimensions are,<4,10,3,12,20,7>. The matrices have sizes 4*10,10*3,3*12,12*20,20*7 .we need to compute M[i, j],0 < =i, j<=5.We know M[i, i] = 0 for all i

1	2	3	4	5	
0					1
	0				2
		0			3
			0		4
				0	5

We proceed, working away from the diagonal. We compute the optimal solutions for products of 2 matrices.

1	2	3	4	5	
0	120				1
	0	360			2
		0	720		3
			0	1680	4
				0	5

Now products of 3 matrices

{4, 10, 3, 12, 20, 7}

$$M[1,3] = \min \begin{cases} M[1,2] + M[3,3] + p_0p_2p_3 = 120 + 0 + 4 \cdot 3 \cdot 12 = 264 \\ M[1,1] + M[2,3] + p_0p_1p_3 = 0 + 360 + 4 \cdot 10 \cdot 12 = 840 \end{cases} = 264$$

$$M[2,4] = \min \begin{cases} M[2,3] + M[4,4] + p_1p_3p_4 = 360 + 0 + 10 \cdot 12 \cdot 20 = 2760 \\ M[2,2] + M[3,4] + p_1p_2p_4 = 0 + 720 + 10 \cdot 3 \cdot 20 = 1320 \end{cases} = 1320$$

$$M[3,5] = \min \begin{cases} M[3,4] + M[5,5] + p_2p_4p_5 = 720 + 0 + 3 \cdot 20 \cdot 7 = 1140 \\ M[3,3] + M[4,5] + p_2p_3p_5 = 0 + 1680 + 3 \cdot 12 \cdot 7 = 1932 \end{cases} = 1140$$

1	2	3	4	5	
0	120				1
	0	360			2
		0	720		3
			0	1680	4
				0	5

⇒

1	2	3	4	5	
0	120	264			1
	0	360	1320		2
		0	720	1140	3
			0	1680	4
				0	5

Now, products of 4 matrices

$$M[1,4] = \min \begin{cases} M[1,3] + M[4,4] + p_0p_3p_4 = 264 + 0 + 4.12.20 = 1224 \\ M[1,2] + M[3,4] + p_0p_2p_4 = 120 + 720 + 4.3.20 = 1080 \\ M[1,1] + M[2,4] + p_0p_1p_4 = 0 + 1320 + 4.10.20 = 2120 \end{cases} = 1080$$

$$M[2,5] = \min \begin{cases} M[2,4] + M[5,5] + p_1 p_4 p_5 = 1320 + 0 + 10 \cdot 20 \cdot 7 = 2720 \\ M[2,3] + M[4,5] + p_1 p_3 p_5 = 360 + 1680 + 10 \cdot 12 \cdot 7 = 2880 \\ M[2,2] + M[3,5] + p_1 p_2 p_5 = 0 + 1140 + 10 \cdot 3 \cdot 7 = 1350 \end{cases} = 1350$$

1	2	3	4	5	
0	120	264			1
	0	360	1320		2
		0	720	1140	3
			0	1680	4
				0	5

$\Rightarrow$

1	2	3	4	5	
0	120	264	1080		1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

Now products of 5 matrices

$$M[1,5] = \min \begin{cases} M[1,4] + M[5,5] + p_0 p_4 p_5 = 1080 + 0 + 4 \cdot 20 \cdot 7 = 1544 \\ M[1,3] + M[4,5] + p_0 p_3 p_5 = 264 + 1680 + 4 \cdot 12 \cdot 7 = 2016 \\ M[1,2] + M[3,5] + p_0 p_2 p_5 = 120 + 1140 + 4 \cdot 3 \cdot 7 = 1344 \\ M[1,1] + M[2,5] + p_0 p_1 p_5 = 0 + 1350 + 4 \cdot 10 \cdot 7 = 1630 \end{cases} = 1344$$

1	2	3	4	5	
0	120	264	1080		1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

$\Rightarrow$

1	2	3	4	5	
0	120	264	1080	1344	1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

To print the optimal parenthesization, we use the PRINT-OPTIMAL-PARENS procedure.

PRINT-OPTIMAL-PARENS (s, i, f)

1. if $i = j$
2. then print "A"
3. else print "("
4. PRINT-OPTIMAL-PARENS $(s, i, s, [i, j])$
5. PRINT-OPTIMAL-PARENS $(s, s[i, j] + 1, j)$
6. print ")"

Now for optimal paranthesization, Each time we find the optimal value for $M[i, j]$ we also store the value of k that we used. If we did this for the example, we would get.

1	2	3	4	5	
0	120/1	264/2	1080/2	1344/2	1
	0	360/2	1320/2	1350/2	2
		0	720/3	1140/4	3
			0	1680/4	4
				0	5

The k value for the solution is 2, so we have $((A_1 A_2)(A_3 A_4 A_5))$. The first half is done. The optimal solution for the second half comes from entry $M[3, 5]$. The value of k here is 4, so now we have $((A_1 A_2)((A_3 A_4) A_5))$. Thus the optimal solution is to parenthesize $((A_1 A_2)((A_3 A_4) A_5))$.

Q. 5. (a) Compute binomial coefiicient using dynamic programming.

Ans: 1. A binomial coefficient C(n, k) can be defined as the coefficient of X^k in the expansion of (1 + X)^n.

2. A binomial coefficient C(n, k) also gives the number of ways, disregarding order, that k objects can be chosen from among n objects; more formally, the number of k-element subsets (or k-combinations) of an n-element set.

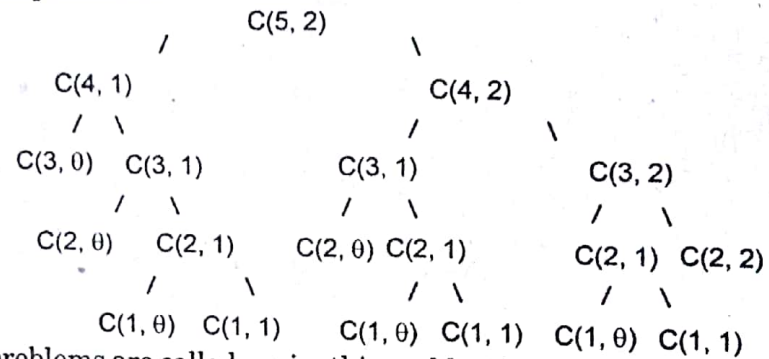
The Problem: *Write a function that takes two parameters n and k and returns the value of Binomial Coefficient C(n, k).* For example, your function should return 6 for n = 4 and k = 2, and it should return 10 for n = 5 and k = 2.

(1) Optimal Substructure: The value of C(n, k) can be recursively calculated using following standard formula for Binomial Coefficients.

$C(n, k) = C(n-1, k-1) + C(n-1, k)$

$C(n, 0) = C(n, n) = 1$

(2) Overlapping Subproblems: It should be noted that the above function computes the same subproblems again and again. See the following recursion tree for $n = 5$ an $k = 2$. The function $C(3, 1)$ is called two times. For large values of n, there will be many common subproblems.



Since same suproblems are called again, this problem has Overlapping Subproblems property. So the Binomial Coefficient problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, re-computations of same subproblems can be avoided by constructing a temporary array C[][] in bottom up manner. Following is Dynamic Programming based implementation.

Q. 5. (b) Explain the Floyd Warshall algorithm and discuss its complexity.

Ans. Floyd-Warshall algorithm (sometimes known as the Roy-Floyd algorithm, since Bernard Roy described this algorithm in 1959) is a graph analysis algorithm for finding shortest paths in a weighted, directed graph. A single execution of the algorithm will find the shortest path between all pairs of vertices. It does so in $\theta(V^3)$ time, where V is the number of vertices in the graph. Negative-weight edges may be present, but we shall assume that there are no negative-weight cycles.

FLOYD-WARSHALL (W)

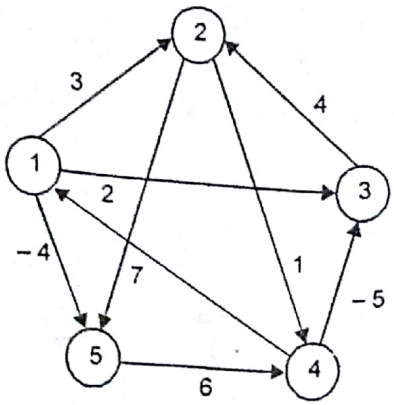
1. n ← rows[W]
2. $D^{(0)} \leftarrow W$
3. for k ← 1 to n
4. do for I ← 1 to n
5. do for j ← 1 to n
6. do $d_{ij}^{(k)} \leftarrow \min (d_{ij}^{(k-1)}, d_{jk}^{(k-1)}, d_{kj}^{(k-1)})$
7. return $D^{(n)}$

$O(V^3)$

The strategy adopted by the Floyd-Warshall algorithm is dynamic programming.

The running time of the Floyd-Warshall algorithm is determined by the triply nested for loops of lines 3-6. Each execution of line 6 takes O(1) time. The algorithm thus runs in time $\theta(n^3)$.

Apply Floyd Warshall algorithm for constructing shortest path



$$d_{ij}^{(k)} = \min\left[d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right]$$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} \text{ if } d_{ij}^{(k-1)} \le d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} \text{ if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \pi^{(0)} = \begin{bmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \pi^{(1)} = \begin{bmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \pi^{(2)} = \begin{bmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \pi^{(4)} = \begin{bmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{bmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \pi^{(5)} = \begin{bmatrix} NIL & 3 & 4 & 5 & 1 \\ 4 & NIL & 4 & 2 & 1 \\ 4 & 3 & NIL & 2 & 1 \\ 4 & 3 & 4 & NIL & 1 \\ 4 & 3 & 4 & 5 & NIL \end{bmatrix}$$

Q. 6. (a) Explain the difference between Dijkstra,s and Bellmanford algorithm with the help of example.

Ans. DIJKSTRA (G, w, s)

1. INITIALIZE-SINGLE-SOURCE (G,s)
2. S ← 00
3. Q ← V[G]
4. while Q ≠ 0
5. do u ← EXTRACT-MIN (Q)
6. S ← S ∪ {u}
7. for each vertex v ∈ Adj [u]
8. do RELAX (u,v,w)

BELLMAN-FORD (G, w, s)

1. INITIALIZE-SINGLE-SOURCE (G, s)
2. for i ← 1 to | V[G] | – 1
3. do for each edge (u,v)∈ E[G]
4. do RELAX (u,v,w)
5. for each edge (u,v) ∈ E[G]
6. do if d [v] > d [u] + w (u, v)
7. then return FALSE
8. return TRUE

Q. 6. (b) Find the optimal Schedule for the following jobs with profit (p1,p2,p3,p4,p5,p6) = (3,5,17,20,6,10) and deadlines(d1,d2,d3,d4,d5,d6) = (1,3,3,4,1,2)

Ans. above question can be solved in this way:

Step: 1 We will arrange the profits Pi in descending order, along with corresponding deadlines.

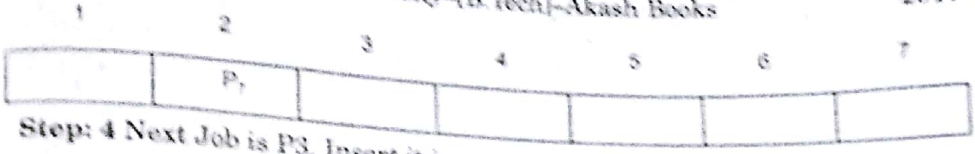
Profit	30	20	18	6	5	3	1
Job	P7	P3	P4	P6	P2	P1	P5
Deadlines	2	4	3	1	3	1	2

Step: 2 Create an array J [] which stores the jobs. Initially j [] will be

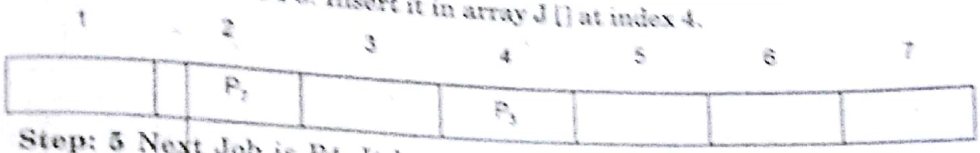
1	2	3	4	5	6	7
0	0	0	0	0	0	0

Step: 3 Add ith Job in array J [ ] at index denoted by its deadlines Di
First Job is P7, its deadline is 2.

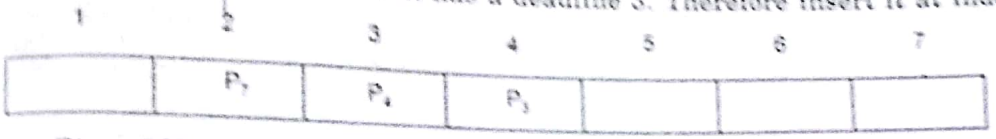
Hence insert P7 in the array J [] at 2nd index.



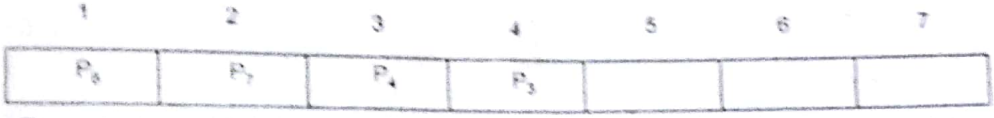
Step: 4 Next Job is P3. Insert it in array J [] at index 4.



Step: 5 Next Job is P4. It has a deadline 3. Therefore insert it at index 3.



Step: 6 Next Job is P6, it has deadline 1. Hence Place P6 at index 1.



Step: 7 Next Job is P2, it had deadline 3. But as 3 is already occupied and there is no empty slot at index < J [3]. Just discard job P2. Similarly Job P1 and P5 will get discarded.

Step: 8 Thus the optimal sequence which we will obtain will be 6-7-4-3. The maximum profit will be 74.

Q. 7. (a) Explain Prim's algorithm for finding the minimum spanning tree and analyze its complexity.

Ans. MST-PRIM (G,w,r)

1. for each u ∈ V[G]
2. do key [u] ← ∞
3. π[u] ← NIL
4. key [r] ← 0
5. Q ← V[G]
6. while Q ≠ ∅
7. do u ← EXTRACT-MIN(Q)
8. for each v ∈ Adj [u]
9. do if v ∈ Q and w (u,v) < key [v]
10. then π[v] ← u
11. key [v] ← w (u,v)

Q. 7. (b) Consider 5 items along their respective weights and values

$$I = (I_1, I_2, I_3, I_4, I_5)$$

$$w = (5, 10, 20, 30, 40)$$

$$v = (30, 20, 100, 90, 160)$$

The capacity of knapsack W = 60. Find the solution to the fractional knapsack problem.

22–2017

Ans. Initially,

Item	w_i	v_i
I_1	5	30
I_2	10	20
I_3	20	100
I_4	30	90
I_5	40	160

Taking value per weight ratio *i.e.*, $P_i = v_i / w_i$

Item	w_i	v_i	$P_i = v_i/w_i$
I_1	5	30	6.0
I_2	10	20	2.0
I_3	20	100	5.0
I_4	30	90	3.0
I_5	40	160	4.0

Now, arrange the value of P_i in decreasing order.

Item	w_i	v_i	$P_i = v/w_i$
I_1	5	30	6.0
I_3	20	100	5.0
I_5	40	160	4.0
I_4	30	90	3.0
I_2	10	20	2.0

Now, fill the knapsack according to the decreasing value of P_i.

First we choose item I_1 whose weight is 5, then choose item I_3 whose weight is 20. Now the total weight in knapsack is 5 + 20 = 25.

Now, the next item is I_5 and its weight is 40, bit we want only 35. So we choose fractional part of if i.e.,

35
20
5
– 60 The value of fractional part of I_5 is $\frac{160}{40} \times 35 = 140$

Thus the maximum value = 30 + 100 + 140 = 270

Q. 8. (a) Differntiate between P and NP Problems.Explain polynomial time verification with an example.How it is different from polynolial time solution.

Ans. P problems are the problems which can be solved in a Polynomial time complexity.For example: finding the greatest number in a series of multiple numbers, will take atmost 'N' number of steps , where N is the count of numbers in the sequence.

Where as NP problems may include those which cannot be solved(or are yet to be solved) in Polynomial time

Although some NP problems are yet to be solved in Polynomial time, once we have a solution, it can be checked in Polynomial time.

For example: '*n* queens problem' where you have to arrange '*n*' number of queens in a chess board in such a way that no 2 queens are in the same row, column or diagonal.

Here, finding a suitable arrangement in Polynomial time may be impossible , but verifying that a certain arrangement works can be done in Polynomial time.

Polynomial time verification: For some problems, the answer can be verified to be correct in Polynomial Time, even if there is no known way of solving the original problem in Polynomial Time.

For example, consider solving NxN Sudoku. There is no known way to solve this where the time is bound by a polynomial function of N for sufficiently large values.

However, it is easy to check if an NxN Sudoku is solved correctly. Simply 1) verify that every row, column, and box contains each of the numbers 1..N exactly once; 2) verify that the original clues are respected in the solution. Both of these operations are polynomial-time.

Another example is integer factorization. While there is no known way to factor N-digit numbers in Polynomial time, it is easy to check whether a factorization is correct by simply multiplying the purported factors and comparing to the original number.

Q. 8. (b) Illustrate string matching with finite automata.

Ans. We define the string-matching automation corresponding to a given pattern P[1..m] as follows.

The state set Q is {0,1,..,m}. The start state q_0 is state 0, and state m is the only accepting state.

The transition function δ is defined by the following equation for any state q and character a:

$$\delta(q, a) = \sigma(P_q a)$$

As for any string-matching automation for a pattern of length m, the state set Q is {0,1,...,m}, the start state is 0, and the only accepting state is state m.

FINITE-AUTOMATION-MATCHER (T, δ, m)

1. n ← length [T]
2. q ← 0
3. for i ← 1 to n
4. do q ←δ (q,T[i])
5. if q = m
6. then s ← i – m
7. print "Pattern occurs with shift" s

COMPUTE-TRANSITION-FUNCTION (P,Σ)

1. m ← length [P]
2. for q ← 0 to m
3. do for each character a ∈ Σ*
4. do k ← min (m+1, q+2)
5. repeat k ← k – 1
6. until
7. δ (q,a)← k
8. return δ

Q. 9. (a) Explain NP hard and NP Complete problems with the help of suitable example.

Ans. NP-complete problems are special kinds of NP problems. You can take any kind of NP problem and twist and contort it until it looks like an NP-complete problem.

For example, the knapsack problem is NP. It can ask what's the best way to stuff a knapsack if you had lots of different sized pieces of different precious metals lying on

the ground, and that you can't carry all of them in the bag.

Surprisingly, there are some tricks you can do to convert this problem into a travelling salesman problem. In fact, any NP problem can be made into a travelling salesman problem, which makes travelling salesman NP-complete.

(Knapsack is also NP-complete, so you can do the reverse as well!)

NP-Hard problems are worst than NP problems. Even if someone suggested you a solution to a NP-Hard problem, it'd still take forever to verify if they were right.

For example, in travelling salesman, trying to figure out the absolute shortest path through 500 cities in your state would take forever to solve. Even if someone walked up to you and gave you an itinerary and claimed it was the absolute shortest path, it'd still take you forever to figure out whether he was a liar or not.

Q. 9. (b) Explain KNUTH_MORRIS_PRATT string matching algorithm

Ans. Knuth–Morris–Pratt string searching algorithm (or KMP algorithm) searches for occurrences of a "word" W within a main "text string" S by employing the observation that when a mismatch occurs, the word itself embodies sufficient information to determine where the next match could begin, thus bypassing re-examination of previously matched character consider a (relatively artificial) run of the algorithm, where W = "ABCDABD" and S = "ABC ABCDAB ABCDABCDABDE". At any given time, the algorithm is in a state determined by two integers:

- m, denoting the position within S where the prospective match for W begins,
- i, denoting the index of the currently considered character in W.

```
COMPUTE-PREFIX-FUNCTION(p)
1  m <– length[p]
2  π[1] <–0
3  k <– 0
4  for q <– 2 to m
5  do while k>0 and p[k+1] ≠ p[q]
6  do k <– π[k]
7  if p[k+1] = p[q]
8  then k<–k+1
9  π[q]<–k
10 return π
```

The KMP matching algorithm is given in KMP-MATCHER.

```
KMP-MATCHER(t,p)
1  n<-length[t]
2  m<-length[p]
3  π<-COMPUTE-PREFIX-FUNCTION(p)
4  q<-0
5  for i<-1 to n
6  do while q>0 and p[q+1] ≠t[i]
7   do q<-π[q]
8  if p[q+1] = t[i]
9  q <-q+1
10 if q= m
11 then print "Pattern occurs with shift"i-m
12 q<-π[q]
```