CG-9
2022

1)

a) One problem may have different solution but we will select algorithm which is best w.r.t. time complexity & space complexity.

For given problem algorithm A has linear time complexity, for B it is exponential that means after certain value of n, growth rate of B is very high in comparison to A. So A is showing better time complexity than B, so we will select algorithm A.

b) The recursive algorithm to find the nth term of Fibonacci series:

```
function Fibo(n)
{
    if(n<=1)
        return 1

    return n*fibo(n-1)
}
```

c) NP problem

NP is a class of problems for which we have algorithm for which time complexity is polynomial only when the algorithm has at-least one decission where we use some non-deterministic technique. Here nondetermini-tic means at any certain state depending on present ~~output~~ input there are many possibili-ties. Using some hemeristic or probabilistic approach best decission is taken out of many possibilities. When this type of decission making is used only then we are getting solution of the problem using some polynomial function of number of inputs. If any problem of this class is solved using a deterministic algorithm with polynomial time complexity, then every problem in this set can be mapped into polynomial time solution technique. Hence all problems can be solved using polynomial time deterministic algorithm.

d) Space Complexity

Space complexity means additional space required by the algorithm to convert given input to the required output. If no. of such additional space not depending on no. of input the space complexity O(1).

Incase of bubble sort, selection sort etc. we need some index variables & data variables but these are constant independent no. of input, space complexity O(1).

In case of merge sort we need one more array of size same as no. of inputs say n. So space complexity O(n).

Note that during processing we have to maintain the input data. So space required by input is not considered in space complexity calculation.

e)

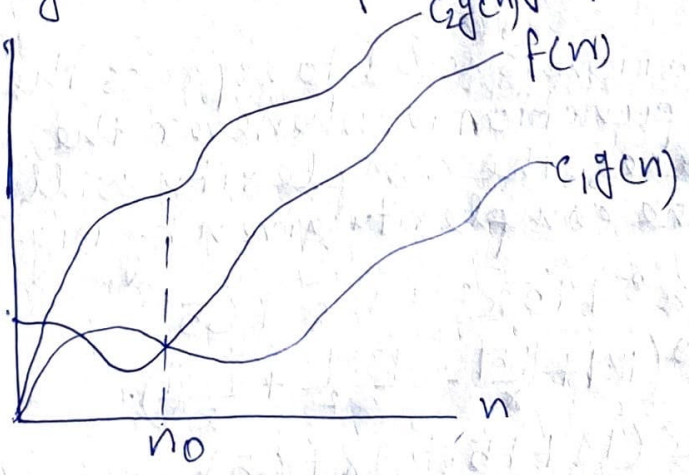
BFS	DFS
① BFS finds the shortest path using the Queue data structure.	DFS make use of the stack data structure.
② It is base on the FIFO principle (First In First Out) as Queue is used.	It is based on the LIFO (Last In First Out) as stack is used.
③ BFS constructs a tree level per level.	DFS constructs the tree subtree by subtree.
④ BFS is more suited for finding vertices that are near the provided source.	If there are alternatives away from the source, DFS is more appropriate.

f)

Dijkstra's Algorithm	Floyd's Algorithm
① This algorithm is used to find shortest path between a pair of vertex.	This algorithm is used to find all pair shortes path.
② Tree is generated if source vertex is given but destination vertex is not given.	Does not form a tree. as shortest paths exist among certain vertices.
③ It forms spanning tree.	It does not form spanning tree.
④ It is based on greedy algorithm.	It is base on dynamic algorithm.

g) Θ notation

Θ notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average case complexity of an algorithm.



$\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2 & n_0 such that $0 \le c_1g(n) \le f(n) \le c_2g(n)$ for all $n \ge n_0\}$

If a function $f(n)$ lies anywhere in between $c_1g(n)$ and $c_2g(n)$ for all $n \ge n_0$, then $f(n)$ is said to be asymptotically tight bound.

1) The growth of a function refers to how its output values change concerning the increase in the input values. In other words, it quantifies how quickly the function's output increases or decreases as its input changes. To measure the growth of function we use asymptotic analysis, generally "big O notation." Big O notation provides an upper bound on the growth rate of a function, which allows us to understand the function's worst-case behaviour for large input values.

Eg: function sum(n)
{
sum = 0
for i = 0 to n
sum = sum + i
return sum
}

In this function the no. of iterations of loop depends on n, the input number, that's why the growth of this function is in order of n or O(n).

2) a) Big-O notation is used to express the upperbound of a function ie. whenever the input size increases, the complexity will not suppress the complexity given by Big-O.

$$f(x) = 19x^3 + 15x^2 + 98x + 65$$
$$\leq x^3\left(|19| + \frac{|15|}{x} + \frac{|98|}{x^2} + \frac{|65|}{x^3}\right)$$
$$< x^3(19 + 15 + 98 + 65)$$
$$= 197x^3$$

$\therefore f(x) \leq c.g(x)$, for $n \geq n_0$

Here $c = 197$ $g(x) = x^3$

So, the given function is in order of ~~O(g(x)), O(x³).~~ O(g(x)) or, O(x³)

b) n^2 is in order of $O(n^2)$

n is in order of $O(n)$, which is linear complexity

$\log(n)$ is in order of $O(\log n)$, which is less than linear.

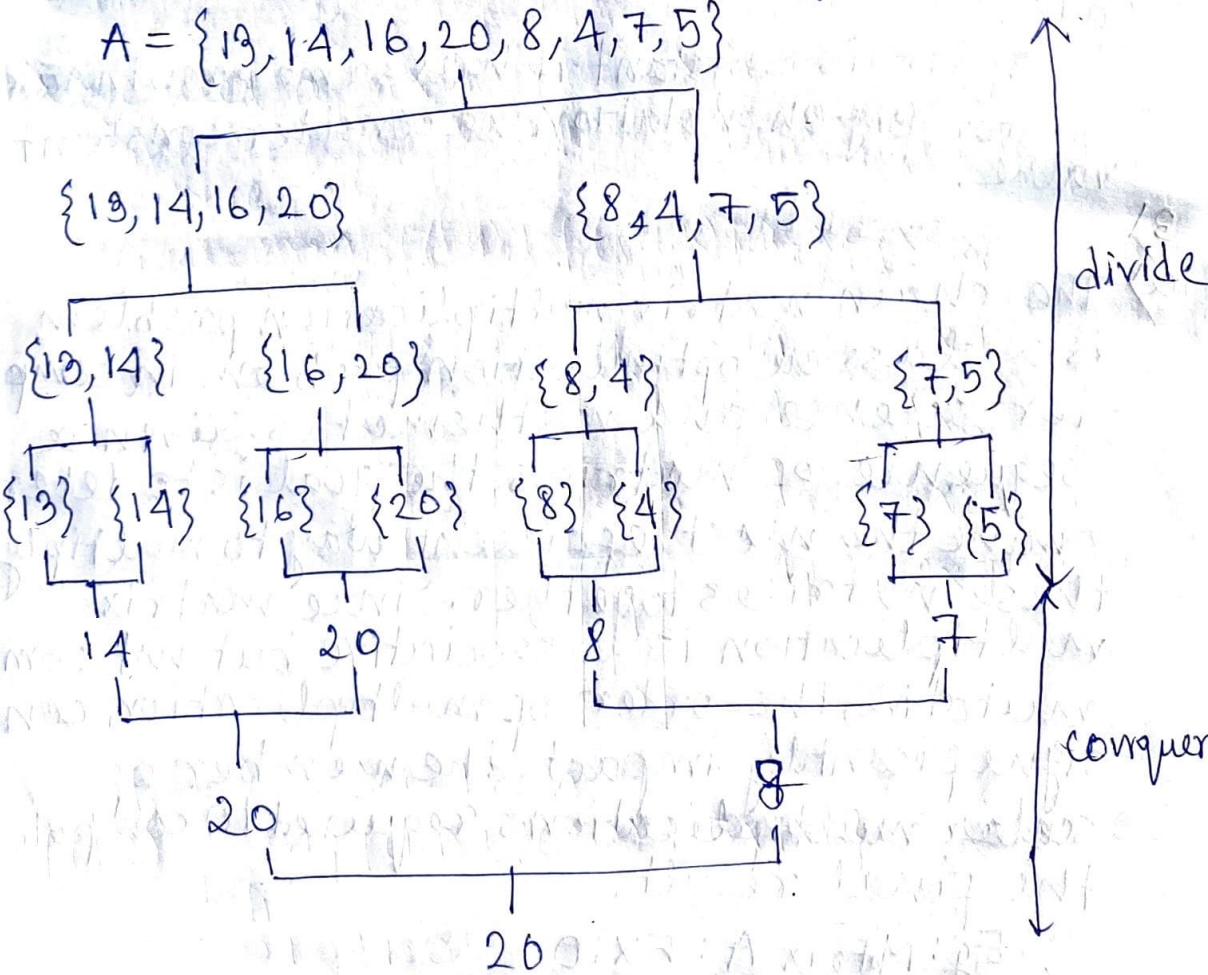
e^n ~~is in order~~ has exponential complexity.

∴ Arranging in ascending order becomes:

$\log(n) < n < n^2 < e^n$.

c) To solve the given question we have to divide the array until we get arrays with single element & after that comparing each element we conquer the maximum elements of the following array.

$A = \{13, 14, 16, 20, 8, 4, 7, 5\}$



∴ The maximum element in the array is 20.

1/h) Alternative

Growth rate definition = $\frac{f(x_2) - f(x_1)}{f(x_1)} \times 100\%$

On the basis of complexity it can be written as

$$\lim_{x \to \infty} \frac{f(x)}{g(x)}$$

Among all asymptotic notation, the most common one $O(n)$. We use two function $f(x)$ & $g(x)$ for comparison & here growth rate is defined

$$\lim_{x \to \infty} \frac{f(x)}{g(x)}$$

If the value is tending to 0 then growth rate of $f(x)$ lower than $g(x)$.

For larger value $f(x)$ has higher growth rate.

If it is constant it has same growth rate.

For Big-Oh notation we consider constant value.

3)

a) The chain matrix multiplication problem is a classical optimization problem in computer science and mathematics. Given a sequence of matrices, the goal is to determine the most efficient way to multiply these matrices together. Since matrix multiplication is associative but not commutative, the order of multiplication can significantly impact the number of scalar multiplications required to compute the final result.

Eg: Matrix A: 5×10
Matrix B: 10×3
Matrix C: 3×6

In case of ((AxB)xC) no. of multiplications

$= 5 \times 10 \times 3 + 5 \times 3 \times 6$

$= 150 + 90$

$= 240$

In case of (Ax(BxC)), it is $= 5 \times 10 \times 6 + 10 \times 3 \times 6$

$= 300 + 180$

$= 480$

so, here the first sequence is the better option.

If we solve this problem recursively then time complexity becomes $O(2^n)$, which is exponential.

If we use dynamic programming to reduce redundant calculation, then time complexity is $O(n^3)$, & space complexity is $O(n^2)$

b) We need a nxn space & here we are using a basic formula to find no. of steps ie,

$$f(i,k) = \min_{i \le j < k} \{f(i,j) + f(j+1,k) + r_i \times c_j \times c_k\}$$

1	2	3	4
(1,1) 0	(1,2) 120	(1,3) 88	(1,4)
	(2,2) 0	(2,3) 48	(2,4)
		(3,3) 0	(3,4) 84
			(4,4) 0

A_1	5x4
A_2	4x6
A_3	6x2
A_4	2x7

$f(i,i) = 0 \quad \forall i = 1 \text{ to } n$

$f(1,2) = \min_{1 \le j < 2} \{f(1,1) + f(2,2) + 5 \times 4 \times 6\}$

$= 0 + 0 + 120$

$f(2,3) = \{f(2,2) + f(3,3) + 4 \times 6 \times 2\} \quad [\because j = 2]$

$= 48$

$f(3,4) = \{f(3,3) + f(4,4) + 6 \times 2 \times 7\} \quad [j = 3]$

$= 84$

$f(1,3) = \{f(1,1) + f(2,3) + 5\times4\times2\}$ [$\because j=1$]

$= 0 + 48 + 40$

$= 88$

$f(1,3) = \{f(1,2) + f(3,3) + 5\times6\times2\}$ [$\because j=2$]

$= 120 + 0 + 60$

$= 180$

$\therefore \min f(1,3) = 88$

$f(2,4) = \{f(2,2) + f(3,4) + 4\times6\times7\}$ [$\because j=2$]

$= 0 + 84 + 168$

$= 252$

$f(2,4) = \{f(2,3) + f(4,4) + 4\times2\times7\}$ [$\because j=3$]

$= 48 + 56$

$= 104$

$\therefore \min f(2,4) = 104$

$f(1,4) = \{f(1,1) + f(2,4) + 5\times4\times7\}$ [$\because j=1$]

$= 104 + 140$

$= 244$

$f(1,4) = \{f(1,2) + f(3,4) + 5\times6\times7\}$ [$\because j=2$]

$= 120 + 84 + 210$

$= 414$

$f(1,4) = \{f(1,3) + f(4,4) + 5\times2\times7\}$ [$\because j=3$]

$= 88 + 70$

$= 158$

$\therefore \min f(1,4) = 158$

So, the answer is

$((A_1)_{5\times4} \times ((A_2)_{4\times6} \times (A_3)_{6\times2})) \times (A_4)_{2\times7}$

7)

a) The salient features of recursive algorithm:

① Definition: In which algorithm the function calls itself in its body and after hitting base condition, it starts to return the value and the function is returned, it is called recursive algorithm.

② Base condition: Recursive algorithm must have a base condition to stop, if it is not available stack overflow may occur.

③ Call itself: The function must call itself in the function body, without this recursion is not possible.

④ Uses external stack: External stack is used to store previously called function until these function get any return value.

⑤ Divide & Conquer: We can use divide and ~~conq~~ conquer approach using recursion to ~~get~~ solve any problem, eg. merge sort.

b) Greedy Algorithm fails to give global optimum solution because it is not exploring all possibilities instead it is selecting local solution in hand.

For example to get the change of rupees 7 usings coins 1, 3, 4, 5. In greedy approach we are using minimum and maximum value coins so 2 "1"s & 1 "5" that is 3 coins may be used for the change, however in optimal solution only 2 "3" & "4" is sufficient.

c) 0-1 Knapsack Problem

In the 0-1 knapsack problem, you must either take an item or not take an item. You may not take part of an item, since the goal is to optimize the profit of the items in the knapsack, we might consider using the Greedy method to solve the problem.

We have a knapsack with a capacity of 20 weight units

item	profit	weight	profit/weight
1	50	5	10
2	60	10	6
3	140	20	7

if we pick according to profit/weight ratio, we'll end up with items 1 and 2 for total profit of 110. The better solution is item 3 for total profit of 140. So, greedy algorithm does not give optimum solution for 0/1 knapsack problem.

Fractional knapsack problem

In this case we can take fraction of any item & here here also we will choose product based on profit/weight using greedy algorithm.

item	profit	weight	profit/weight
1	50	5	10
2	60	10	6
3	140	20	7

if we take knapsack with capacity 30 weight unit, then we will first choose item 1 & 3 and after that remaining 5 units are filled with 50% of item 2. So, the profit becomes 220. So, greedy algorithm gives optimum solution for fractional knapsack.

5)

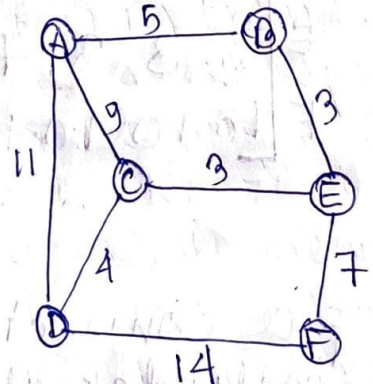
a) Comparison between the greedy approach and Dynamic programic approach:

① Both greedy & dynamic programming approach used for communitorial problems or optimizing problems to find best combination to get optimal solution.

② In greedy approach without exploring all possibilities if we get any solution that is considered as final optimum solution. This is always not true. In dynamic programming all possibilities explored and always best solution path is selected therefore solution is globally optimum.

③ In greedy algorithm only local solution used. It is not requiring any extra storage to maintain solution of all subproblems which are already solved. For dynamic programming this is main issue to maintain lower time complexity. Here instead recomputing it utilises previously computed value at cost of higher space complexity.

b)



The distance matrix will be

$$D^0 = \begin{array}{c} \\ A \\ B \\ C \\ D \\ E \\ F \end{array} \begin{array}{c} \begin{array}{cccccc} A & B & C & D & E & F \end{array} \\ \begin{bmatrix} 0 & 5 & 9 & 11 & \infty & \infty \\ 5 & 0 & \infty & \infty & 3 & \infty \\ 9 & \infty & 0 & 4 & 3 & \infty \\ 11 & \infty & 4 & 0 & \infty & 14 \\ \infty & 3 & 3 & \infty & 0 & 7 \\ \infty & \infty & \infty & 14 & 7 & 0 \end{bmatrix} \end{array}$$

$D^A =$

	A	B	C	D	E	F
A	0	5	9	11	∞	∞
B	5	0	14	16	3	∞
C	9	14	0	4	3	∞
D	11	16	4	0	∞	14
E	∞	3	3	∞	0	7
F	∞	∞	∞	14	7	0

$D^B =$

	A	B	C	D	E	F
A	0	5	9	11	8	∞
B	5	0	14	16	3	∞
C	9	14	0	4	3	∞
D	11	16	4	0	19	14
E	8	3	3	19	0	7
F	∞	∞	∞	14	7	0

$D^C =$

	A	B	C	D	E	F
A	0	5	9	11	8	∞
B	5	0	14	16	3	∞
C	9	14	0	4	3	∞
D	11	16	4	0	7	14
E	8	3	3	7	0	7
F	∞	∞	∞	14	7	0

$D^D =$

	A	B	C	D	E	F
A	0	5	9	11	8	25
B	5	0	14	16	3	30
C	9	14	0	4	3	18
D	11	16	4	0	7	14
E	8	3	3	7	0	7
F	25	30	18	14	7	0

$D^E =$

	A	B	C	D	E	F
A	0	5	9	11	8	15
B	5	0	6	10	3	10
C	9	6	0	4	3	10
D	11	10	4	0	7	14
E	8	3	3	7	0	7
F	15	10	10	14	7	0

$D^F =$

	A	B	C	D	E	F
A	0	5	9	11	8	15
B	5	0	6	10	3	10
C	9	6	0	4	3	10
D	11	10	4	0	7	14
E	8	3	3	7	0	7
F	15	10	10	14	7	0

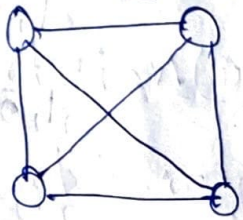
6)

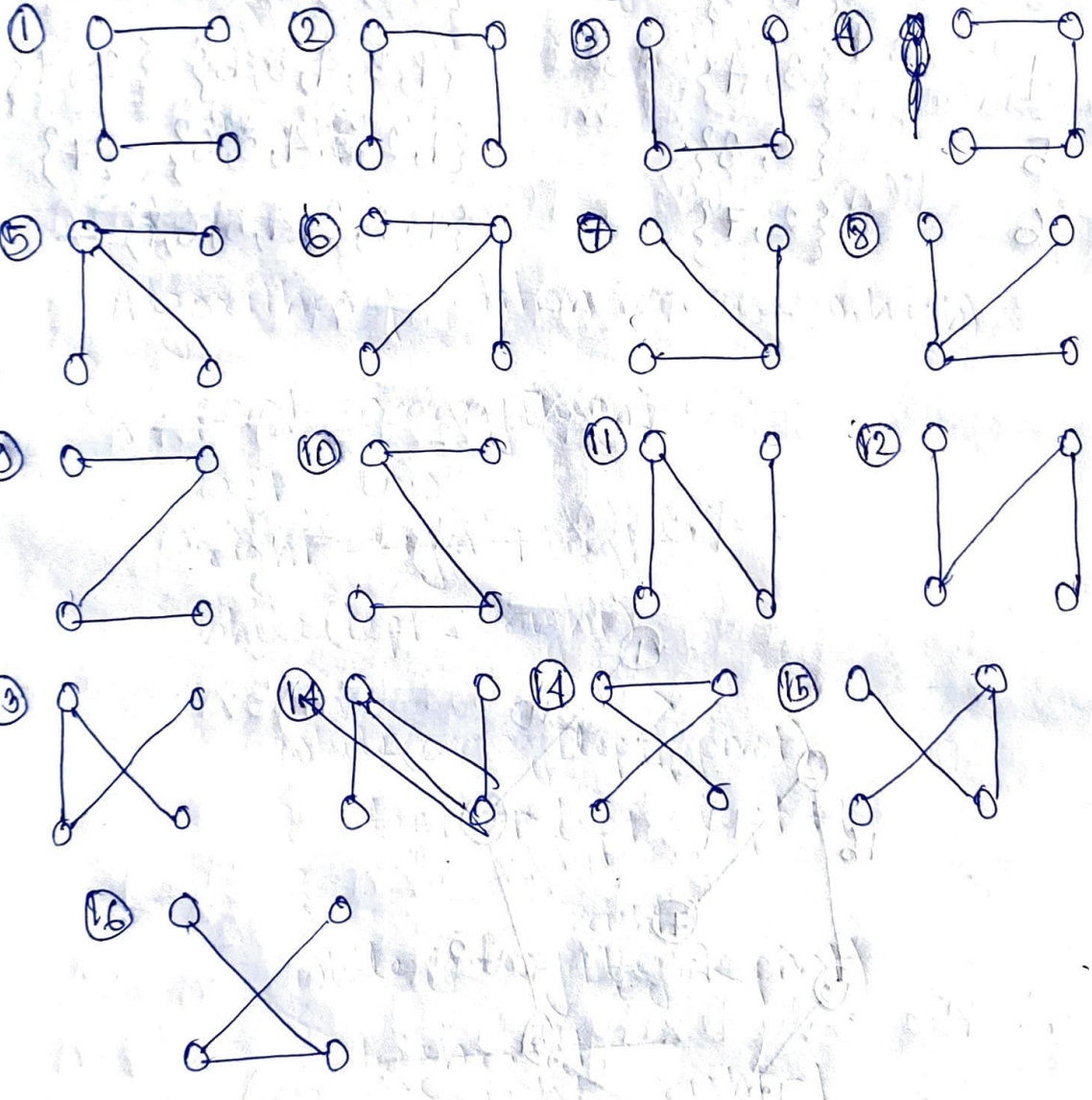
a) **Spanning tree**

A spanning tree of a connected graph is a subgraph that includes all the vertices of the original graph while forming a tree (a connected acyclic graph). In other words, a spanning tree is a subset of the edges of the original graph that connects all the vertices without forming any cycle.

b) The given graph has $n = 4$ vertices so, it will form $n^{n-2} = 4^{4-2} = 16$ spanning trees:

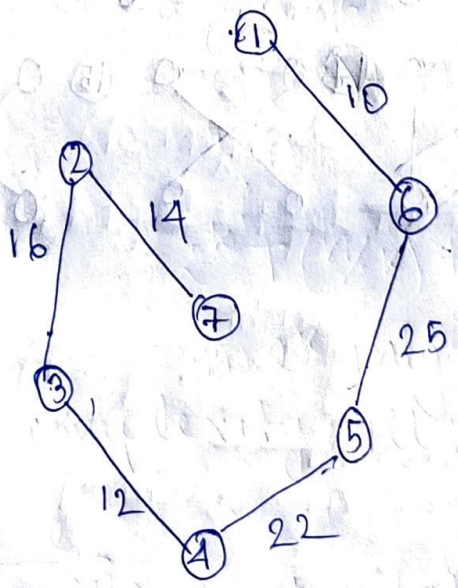
Given graph





c)

Step	Edge Considered	B	V/B
—	Initialization	{1}	{2,3,4,5,6,7}
1	{1,6}	{1,6}	{2,3,4,5,7}
2	{6,5}	{1,5,6}	{2,3,4,7}
3	{4,5}	{1,4,5,6}	{2,3,7}
4	{3,4}	{1,3,4,5,6}	{2,7}
5	{2,3}	{1,2,3,4,5,6}	{7}
6	{2,7}	{1,2,3,4,5,6,7}	—



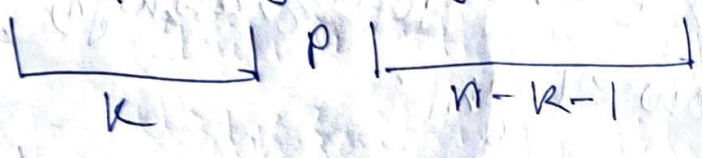
7) a) Algorithm QuickSort(array, low, high)

```
{
    if low < high
    pi = partition(array, low, high)
    Quicksort(array, low, pi-1)
    QuickSort (array, pi+1, high)
    endif
}
```

Algorithm partition (array, low, high)

```
{
    pivot = array[high]
    left = low
    right = right + high-1
    while(left <= right)
    {
        while(array[left] < pivot)
        { left++ left = left+1
        }
        while (array[right] > pivot)
        { right++ right = right-1
        }
        if (left < right)
        { swap (array[left], array[right]
        }
    }
    swap (array[left], array[high])
    return (left)
}
```

b) Complexity analysis



$T(n) = T(k) + T(n-k-1) + \Theta(n)\ n$

Best case

When the pivot element is in the middle, the problem is divided into 2 parts.

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

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

$= 2\{2T(n/4) + n/2\} + \Theta(n)\ n$

$= 4\{2T(n/8) + n/4\} + \Theta(n)\ 2n$

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

$\vdots$

$= 2^k T(n/2^k) + kn$ $\quad \frac{n}{2^k} = 1$

$= n\,2^k + n\lg n$ $\quad n = 2^k$

$\therefore$ The complexity will be $O(n \lg n)$ $\quad k = \lg n$

Worst case

When the pivot is at highest or lowest point then time complexity is worst.

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

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

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

$\vdots$

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

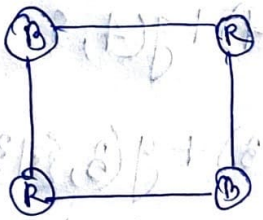
$= T(0) + 1 + 2 + \ldots + (n-1) + n$

$= \frac{n(n+1)}{2} = \frac{n^2}{2} + n/2 \therefore O(n^2)$

8)

a) The graph coloring problem is a well-known combinatorial optimization problem in computer science and graph theory. It involves assigning colours to the vertices of an undirected graph such that no two adjacent vertices (connected by an edge) share the same color. The objective is to minimize the total number of colors used to color the graph while satisfying the constraint of no adjacent vertices having the same color.

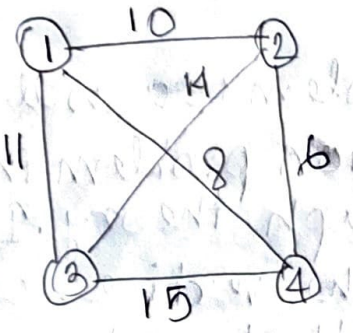
Eg:



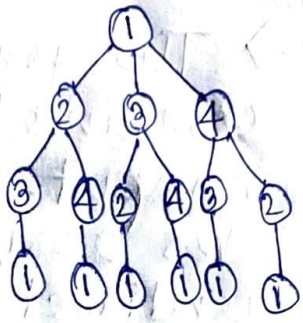
Here 4 vertices can be colored with 2 colours blue and red because it maintains the constraint.

But for complete graph we need n (number of vertices) no. of colors because all the vertices are connected with each other by an edge.

b)



	1	2	3	4
1	0	10	11	8
2	10	0	14	6
3	11	14	0	15
4	8	6	15	0



$$g(i,s) = \min_{j \in s} [w(i,j) + g(j, \{s-j\})]$$

$g(1,\{2,3,4\}) = \min$ [$w(1,2)+g(2,\{3,4\}) = 10+32 = 42$

$w(1,3)+g(3,\{2,4\}) = 11+28 = 39$

$w(1,4)+g(4,\{2,3\}) = 8+31 = 39$]

$g(2,\{3,4\}) = w(2,3)+g(3,\{4\}) = 14+23 = 37$

$= w(2,4)+g(4,\{3\}) = 6+26 = 32$

$g(3,\{4\}) = w(3,4)+g(4,\{\phi\}) = 15+8 = 23$

$g(4,\{3\}) = w(4,3)+g(3,\{\phi\}) = 15+11 = 26$

$g(3,\{2,4\}) = w(3,2)+g(2,\{4\}) = 14+14 = 28$

$= w(3,4)+g(4,\{2\}) = 15+16 = 31$

$g(2,\{4\}) = w(2,4)+g(4,\{\phi\}) = 6+8 = 14$

$g(4,\{2\}) = w(4,2)+g(2,\{\phi\}) = 6+10 = 16$

$g(4,\{2,3\}) = w(4,2)+g(2,\{3\}) = 6+25 = 31$

$= w(4,3)+g(3,\{2\}) = 15+24 = 39$

$g(2,\{3\}) = w(2,3)+g(3,\{\phi\}) = 14+11 = 25$

$g(3,\{2\}) = w(3,2)+g(2,\{\phi\}) = 14+10 = 24$

∴ We can choose two paths ~~ie~~ which has minimum cost ie.

$1 \to 3 \to 2 \to 4 \to 1$, cost = 39

& $1 \to 4 \to 2 \to 3 \to 1$, cost = 39