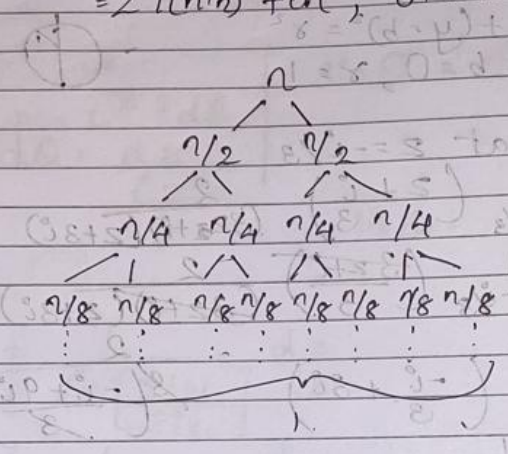
Page No.
Date.

AOA
Question Bank
Answers.

Q.1 Given the following recurrence relation, find its complexity using recursive tree method.

$T(n) = c$; if $n = 1$
$= 2T(n/2) + cn$; otherwise

→



The total cost adding upon the problem is the height of the tree.
Suppose we take n as 16
∴ 16 → 8 → 4 → 2 → 1

Height = 4 which would be given by the log function.
∴ $\log(16) = 4$

Thus the height of tree is $\log n$
∴ Complexity is $O(n \log n)$.

Page No.
Date.

Q.2 Sort the following array using quicksort algorithm.
[40, 11, 4, 72, 17, 2, 49]

→ Step 1: Fix pivot, low, high pointers.
Keep moving low and high pointer until A[low] > A[pivot] and A[high] ≤ A[pivot].
and if low < high then swap A[low] and A[high]
and if low ≥ high then swap A[pivot] and A[high]
Initially,

40 11 4 72 17 2 49
pivot low high

Step 2:

40 11 4 72 17 2 49
pivot low high

Step 3:

40 11 4 72 17 2 49
pivot low high

~~Step 4:~~

~~40 11 4 49 17 2 72~~
~~pivot low high~~

Step 4:

40 11 4 2 17 72 49
pivot low high

Step 5:

sorted

17 11 4 2 [40] 72 49

left sub list right sub list

Let sort the left sub list

Step 6:

17 11 14 2
pivot low high

Step 7:

17 11 14 2
pivot low high

Step 8:

17 11 14 2
pivot high low

Step 9:

17 11 14 2
pivot high low

Step 10:

2 11 14 17

Let sort the right sub list

Step 11:

72 49
pivot low high

Step 12:

49 72

Now the final sorted list is

2 11 14 17 40 49 72.

Page No.
Date.

Q.3) Explain subset sum problem using backtracking approach with the help of state space tree.

→ Subset sum problem is the problem of finding a subset such that the sum of elements equal a given number. The backtracking approach generates all permutations in the worst case but in general, performs better than the recursive algorithm approach towards subset sum problem.

In the backtracking approach as we go down along depth of tree we add elements so far, and if the added sum is satisfying explicit constraints, we will continue to generate child nodes further. whenever the constraints are not met, we stop further generation of sub-trees of that node, and backtrack to previous node to explore the nodes not yet explored. We need to explore the nodes along the breadth and depth of the tree. Generating nodes along breadth is controlled by loop and nodes along the depth are generated using recursion.

Steps.

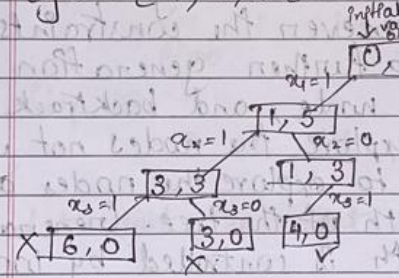
1. Start with an empty set.
2. Add the next element from the list to the set.
3. If the ~~set~~ subset is having solution M, then stop with that subset.

4. If the subset is not feasible or if we have reached the end of set, then backtrack through the subset until we find the most suitable value.

5. If the subset is feasible (sum of subset < M) then go to set step 2.

6. If we have visited all the elements without finding a suitable and if no backtracking is possible then stop without solution.

Eg: {1, 2, 3} M = 4 n = 3



Thus; x [1 | 0 | 1] gives solution M.
1 2 3

1 represents the element included.
0 represents the element not included.

Page No.
Date.

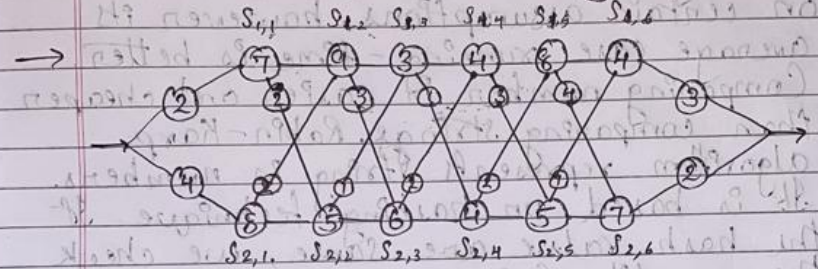
Q.4 Consider assembly line scheduling algorithm with the following specifications:

$e_1 = 2,\ e_2 = 4,\ x_1 = 3,\ x_2 = 2$

$a_1 = \{7, 9, 3, 4, 8, 4\},\ a_2 = \{8, 5, 6, 4, 5, 7\}$

$t_1 = \{2, 3, 1, 3, 4\},\ t_2 = \{2, 1, 2, 2, 1\}$

What will be the minimum time from start to station 3 on assembly line 1.



The minimum time from start to station 3 on assembly line 1 would be given by path.

$e_1 \rightarrow S_{1,1} \rightarrow S_{2,2} \rightarrow S_{1,3}$

$= 2 + 7 + 2 + 5 + 1 + 3$

$= 20$

	j=1	j=2	j=3
$f_1[j]$	9	18	20

j	2	3
$l_1[j]$	1	2

Page No.
Date.

Q.5 Write short note on Rabin Karp algorithm.

→ Rabin & Karp proposed a string matching algorithm that performs well in practice and that ~~to~~ also ~~generates~~ generalizes to other algorithms for related problems, such as two dimensional pattern matching. The Rabin - Karp algorithm uses $O(m)$ processing time, and its worst - case running time is $O((n-m+1)m)$. Based on certain assumptions, however its average case running - time is better. Comparing number is easier and cheaper than comparing strings. Rabin-Karp algorithm represents string in numbers. It is based on hashing technique. If the hash value are same, we check the equality of inverse hash similar to naïve method. If not no need to compare actual string. On hash match actual, characters of both strings are compared using brute force approach. If pattern is found then it is called hit. Otherwise it is spurious hit.

6. Explain the characteristics of dynamic programming approach with the help of Floyd-Warshall algorithm

Ans:

Dynamic programming works on following principles :
Characterize structure of optimal solution, i.e. build a mathematical model of the solution.
Recursively define the value of the optimal solution.
Using bottom-up approach, compute the value of the optimal solution for each possible subproblems.
Construct optimal solution for the original problem using information computed in the previous step.

Let G = <V, E> be a directed graph, where V is set of vertices and E is set of edges with nonnegative length. Find the shortest path between each pair of nodes.

L = Matrix, which gives length of each edge

L[i, j] = 0, if i = j // Distance of node from itself is zero

L[i, j] = ∞, if i ≠ j and (i, j) ∉ E

L[i, j] = w (i, j), if i ≠ j and (i, j) ∈ E

// w(i, j) is the weight of the edge (i, j)

Principle of optimality

If k is the node on shortest path from i to j, then path from i to k and k to j, must also be shortest. In Fig. 4.7.1, optimal path from i to j is either p or summation of p_1 and p_2. In first step, solution matrix is initialized with input graph matrix.

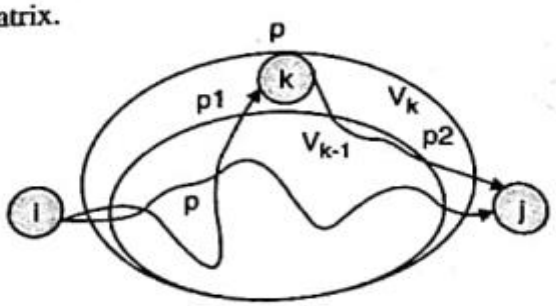


Fig. 4.7.1

We update the distance matrix by considering one by one all vertices as intermediate vertices. In iteration i, the i^{th} vertex is used as an intermediate vertex in the shortest path.

When a k^{th} vertex is under consideration, we already have explored 1 to k – 1 vertex as intermediate vertices. While considering k^{th} vertex as an intermediate vertex, there are two possibilities :

If k is not part of the shortest path from i to j, we keep the distance D[i, j] as it is.

If k is part of shortest path from i to j, update distance D[i, j] as D[i, k] + D[k, j].

Optimal sub structure of the problem is given as :

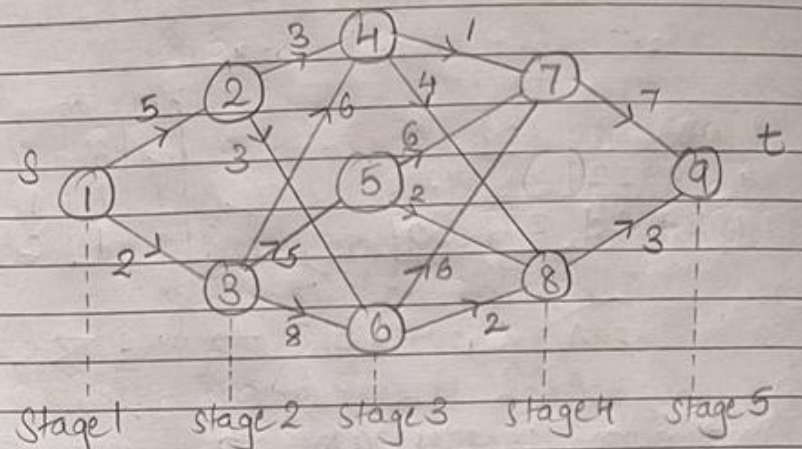
$D^k[i, j] = \min\{ D^{k-1}[i, j], D^{k-1}[i, k] + D^{k-1}[k, j] \}$

D^k = Distance matrix after k^{th} iteration

This approach is also known as **Floyd-warshall** shortest path algorithm. Algorithm for all pair shortest path (APSP) problem is described below:

Page No.
Date.

10. Consider following multistage graph. Write a backward approach algorithm for computing the cost from source node s to target node t. Also compute the cost from s and t using backward approach.



→ $Cost[j] = \min\{c[j, r] + cost[r]\}$

Here number of stages are $k = 5$, number of vertices $n = 9$ source $s = 1$ and target $t = 9$.

Initialization:

$cost[n] = 0 \Rightarrow cost[9] = 0.$

$p[1] = s \Rightarrow p[1] = 1$

$p[k] = t \Rightarrow p[5] = 9$

$r = t = 9.$

Page No.
Date.

Stage 4

$cost[7] = c[7,9] + cost[9]$
$= 7 + 0 = 7$
$\pi[7] = 9$

$cost[8] = c[8,9] + cost[9]$
$= 3 + 0 = 3$
$\pi[8] = 9$

Stage 3

$cost[4] = \min \{ c[4,7] + cost[7],$
$c[4,8] + cost[8] \}$
$= \min \{ 8, 7 \}$
$= 7$
$\pi[4] = 8$

$cost[5] = \min \{ c[5,7] + cost[7],$
$c[5,8] + cost[8] \}$
$= \min \{ 13, 5 \}$
$= 5$
$\pi[5] = 8$

$cost[6] = \min \{ c[6,7] + cost[7],$
$c[6,8] + cost[8] \}$
$= \min \{ 13, 5 \}$
$= 5$
$\pi[6] = 8$

Stage 2

$cost[2] = \min\{ c[2,4] + cost[4],$
$c[2,6] + cost[6] \}$
$= \min\{ 10, 8 \}$
$= 8$

$\pi[2] = 5$

$cost[3] = \min\{ c[3,4] + cost[4],$
$c[3,5] + cost[5],$
$c[3,6] + cost[6] \}$
$= \min\{ 13, 10, 13 \}$
$= 10$

$\pi[3] = 5$

Stage 1

$cost[1] = \min\{ c[1,2] + cost[2]$
$+ c[1,3] + cost[3] \}$
$= \min\{ 13, 12 \}$
$= 12$

$\pi[1] = 3$

Trace the solution

$\pi[1] = 3$ $\pi[3] = 5$
$\pi[5] = 8$ $\pi[8] = 9$

Minimum cost path is
$1 - 3 - 5 - 8 - 9$
and minimum cost is $2 + 5 + 2 + 3 = 12$.

8. Explain Dijkstra's Single source shortest path algorithm. Explain how it is different from Bellman Ford algorithm. Explain 15-puzzle problem using LC search technique.

Ans:

Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph $G = (V, E)$ for the case in which all edge weights are nonnegative. In this section, therefore, we assume that $w(u, v) \geq 0$ for each edge $(u, v) \in E$. As we shall see, with a good implementation, the running time of Dijkstra's algorithm is lower than that of the Bellman-Ford algorithm.

Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u. In the following implementation, we use a min-priority queue Q of vertices, keyed by their d values.

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

Bellman-Ford algorithm is a single-source shortest path algorithm, so when you have negative edge weight then it can detect negative cycles in a graph. The only difference between the two is that **Bellman-Ford is also capable of handling negative weights whereas Dijkstra Algorithm can only handle positives**

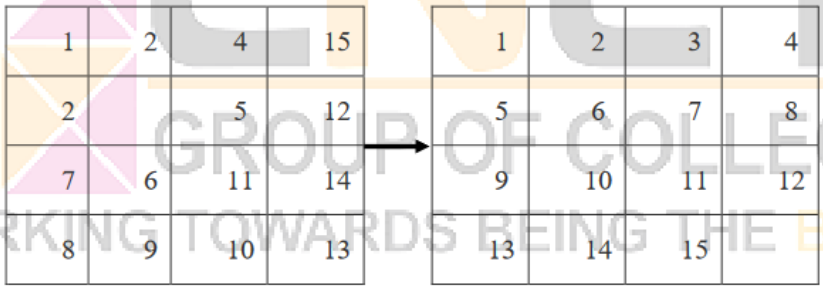
15 Puzzel Problem by Branch and Bound (Least Cost Search)

The 15 Puzzel problem is invented by Sam Loyd in 1878.

The problem cinsist of 15 numbered (0-15) tiles on a square box with 16 tiles(one tile is blank or empty).

The objective of this problem is to change the arrangement of initial node to goal node by using series of legal moves.

The Initial and Goal node arrangement is shown by following figure.



Initial Arrangement Final Arrangement

In initial node four moves are possible. User can move any one of the tile like 2,or 3, or 5, or 6 to the empty tile. From this we have four possibilities to move from initial node.

The legal moves are for adjacent tile number is left, right, up, down, ones at a time.

Each and every move creates a new arrangement, and this arrangement is called state of puzzle problem.

By using different states, a state space tree diagram is created, in which edges are labeled according to the direction in which the empty space moves.

The state space tree is very large because it can be 16! Different arrangements.

In state space tree, nodes are numbered as per the level. In each level we must calculate the value or cost of each node by using given formula:

$C(x)=f(x)+g(x)$,

$f(x)$ is length of path from root or initial node to node x,

$g(x)$ is estimated length of path from x downward to the goal node. Number of non blank tile not in their correct position.

$C(x)<$ Infinity.(initially set bound).

Each time node with smallest cost is selected for further expansion towards goal node. This node become the e-node.

State Space tree with node cost is shown in diagram.

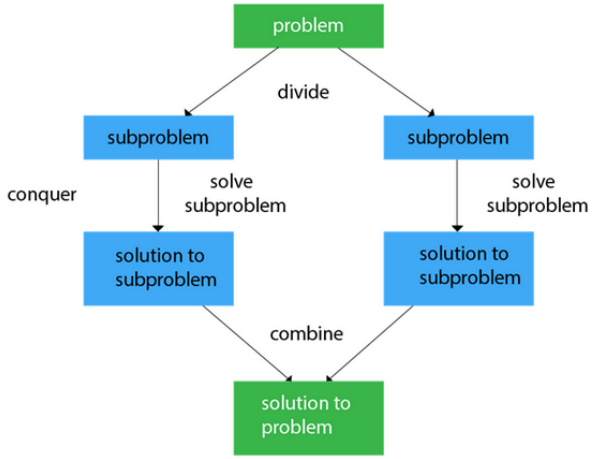
9. Write short note on divide and conquer strategy.

Ans:

Divide and Conquer is an algorithmic pattern. In algorithmic methods, the design is to take a dispute on a huge input, break the input into minor pieces, decide the problem on each of the small pieces, and then merge the piecewise solutions into a global solution. This mechanism of solving the problem is called the Divide & Conquer Strategy.

Divide and Conquer algorithm consists of a dispute using the following three steps.

1. **Divide** the original problem into a set of subproblems.
2. **Conquer:** Solve every subproblem individually, recursively.
3. **Combine:** Put together the solutions of the subproblems to get the solution to the whole problem.



Generally, we can follow the **divide-and-conquer** approach in a three-step process.

Examples: The specific computer algorithms are based on the Divide & Conquer approach:

1. Maximum and Minimum Problem
2. Binary Search
3. Sorting (merge sort, quick sort)
4. Tower of Hanoi.

10. Define: P, NP, NP-complete, NP-Hard

Ans:

- The Class P
  P: the class of problems that have polynomial-time deterministic algorithms.
  – That is, they are solvable in O(p(n)), where p(n) is a polynomial on n
  – A deterministic algorithm is (essentially) one that always computes the correct answer

Sample Problems in P
Fractional Knapsack, MST, Sorting

The class NP

NP: the class of decision problems that are solvable in polynomial time on a nondeterministic machine (or with a nondeterministic algorithm)

– (A determinstic computer is what we know)

– A nondeterministic computer is one that can "guess" the right answer or solution

• Think of a nondeterministic computer as a parallel machine that can freely spawn an infinite number of processes

• Thus NP can also be thought of as the class of problems "whose solutions can be verified in polynomial time"

• Note that NP stands for "Nondeterministic Polynomial-time"

Sample Problems in NP:- Fractional Knapsack, Travelling salesman, etc.

The class NP Hard

If every problem in NP is polynomial time reduceable to a problem 'A', the 'A' is called NP Hard. If 'A' could be solved in polynomial time, then every problem in NP is P.

Eg: Hamiltonian Cycle.

The class NP Complete

If a problem is in NP and NP Hard then that problem is called NP Complete.

11. Compare Bellman Ford algorithm with Dijkstra's algorithm.

Ans:

Bellman Ford's Algorithm	**Dijkstra's Algorithm**
Bellman Ford's Algorithm works when there is negative weight edge, it also detects the negative weight cycle.	Dijkstra's Algorithm may or maynot work when there is negative weight edge.But will definitely not work when there is a negative weight cycle.

Bellman Ford's Algorithm	Dijkstra's Algorithm
The result contains the vertices which contains the information about the other vertices they are connected to.	The result contains the vertices containing whole information about the network, not only the vertices they are connected to.
It can easily be implemented in a distributed way.	It can not be implemented easily in a distributed way.
It is more time consuming than Dijkstra's algorithm. Its time complexity is O(VE).	It is less time consuming. The time complexity is O(E logV).
Dynamic Programming approach is taken to implement the algorithm.	Greedy approach is taken to implement the algorithm.

12. Write a short note on job sequencing with deadline.

Ans: In job sequencing problem, the objective is to find a sequence of jobs, which is completed within their deadlines and gives maximum profit. Let us consider, a set of ***n*** given jobs which are associated with deadlines and profit is earned, if a job is completed by its deadline. These jobs need to be ordered in such a way that there is maximum profit.

It may happen that all of the given jobs may not be completed within their deadlines.

Assume, deadline of **ith** job J_i is d_i and the profit received from this job is p_i. Hence, the optimal solution of this algorithm is a feasible solution with maximum profit.

Thus, $D(i)>0$ for $1 \leqslant i \leqslant n$.

Initially, these jobs are ordered according to profit, i.e. $p_1 \geqslant p_2 \geqslant p_3 \geqslant \ldots \geqslant p_n$

```
Algorithm: Job-Sequencing-With-Deadline (D, J, n, k)
D(0) := J(0) := 0
k := 1
J(1) := 1   // means first job is selected
for i = 2 … n do
   r := k
   while D(J(r)) > D(i) and D(J(r)) ≠ r do
      r := r – 1
   if D(J(r)) ≤ D(i) and D(i) > r then
      for l = k … r + 1 by -1 do
         J(l + 1) := J(l)
         J(r + 1) := i
         k := k + 1
```

13. Apply dynamic programming approach to compute the maximum profit for the following instance of knapsack problem. N= 4, Profit= {1,2,5,6}, Weight = {2,3,4,5}

Ans: Question is Incomplete.

14. What is backtracking? Explain how it is applicable to Graph coloring problem?

Ans:

reject the given item.

- Backtracking is an intelligent way of gradually building the solution. Typically, it is applied to constraint satisfaction problems like Sudoku, crossword, 8-queen puzzles, chess and many other games.
- Backtracking builds the solution incrementally. Partial solutions which do not satisfy the constraints are abandoned.

Backtracking is recursive approach and it is a refinement of brute force method. Brute force approach evaluates all possible candidates, whereas backtracking limits the search by eliminating the candidates that do not satisfy certain constraints. Hence, backtracking algorithms are often faster than brute force approach.

Graph Coloring Problem

In this problem, for any given graph G we will have to color each of the vertices in G in such a way that no two adjacent vertices get the same color and the least number of colors are used.

Solution:

i. First take input number of vertices and edges in graph G.

ii. Then input all the indexes of adjacency matrix of G whose value is 1. Now we will try to color each of the vertex.

iii. A next_color(k) function takes in index of the kth vertex which is to be colored. First we assign color1 to the kth vertex.

iv. Then we check whether it is connected to any of previous (k-1) vertices using backtracking.

v. If connected then assign a color x[i]+1 where x[i] is the color of ith vertex that is connected with kth vertex.

15. Explain the different asymptotic notations with graphs.

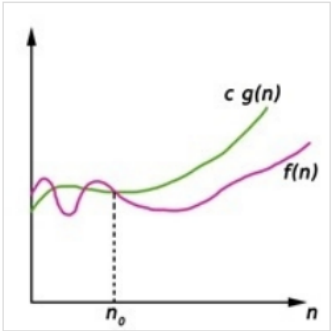
Ans:

Asymptotic Notations

Asymptotic notations are used to represent the complexities of algorithms for asymptotic analysis. These notations are mathematical tools to represent the complexities. There are three notations that are commonly used.

Big Oh Notation

Big-Oh (O) notation gives an upper bound for a function f(n) to within a constant factor.

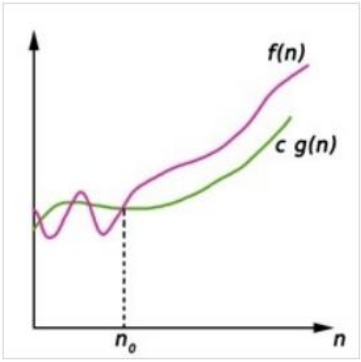


We write f(n) = O(g(n)), If there are positive constantsn0 and c such that, to the right of n0 the f(n) always lies on or below c*g(n).

O(g(n)) = { f(n) : There exist positive constant c and n0 such that $0 \leq f(n) \leq c\ g(n)$, for all $n \geq n0$}

Big Omega Notation

Big-Omega (Ω) notation gives a lower bound for a function f(n) to within a constant factor.

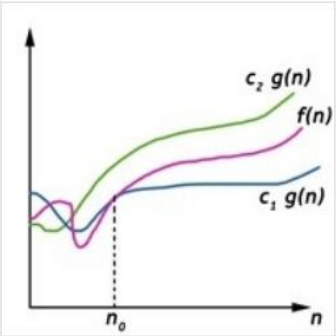


We write f(n) = Ω(g(n)), If there are positive constantsn0 and c such that, to the right of n0 the f(n) always lies on or above c*g(n).

Ω(g(n)) = { f(n) : There exist positive constant c and n0 such that $0 \leq c\ g(n) \leq f(n)$, for all $n \geq n0$}

Big Theta Notation

Big-Theta(Θ) notation gives bound for a function f(n) to within a constant factor.



We write f(n) = Θ(g(n)), If there are positive constantsn0 and c1 and c2 such that, to the right of n0 the f(n) always lies between c1*g(n) and c2*g(n) inclusive.

Θ(g(n)) = {f(n) : There exist positive constant c1, c2 and n0 such that 0 ≤ c1 g(n) ≤ f(n) ≤ c2 g(n), for all n ≥ n0}

16. Explain multistage graph problem with suitable example.

Ans:

A multistage graph **G = (V, E)** is a directed graph where vertices are partitioned into **k** (where ***k* > 1**) number of disjoint subsets ***S* = {s_1,s_2,…,s_k}** such that edge *(u, v)* is in E, then *u* $\in s_i$ and *v* $\in s_{1+1}$ for some subsets in the partition and $|s_1| = |s_k| = 1$.

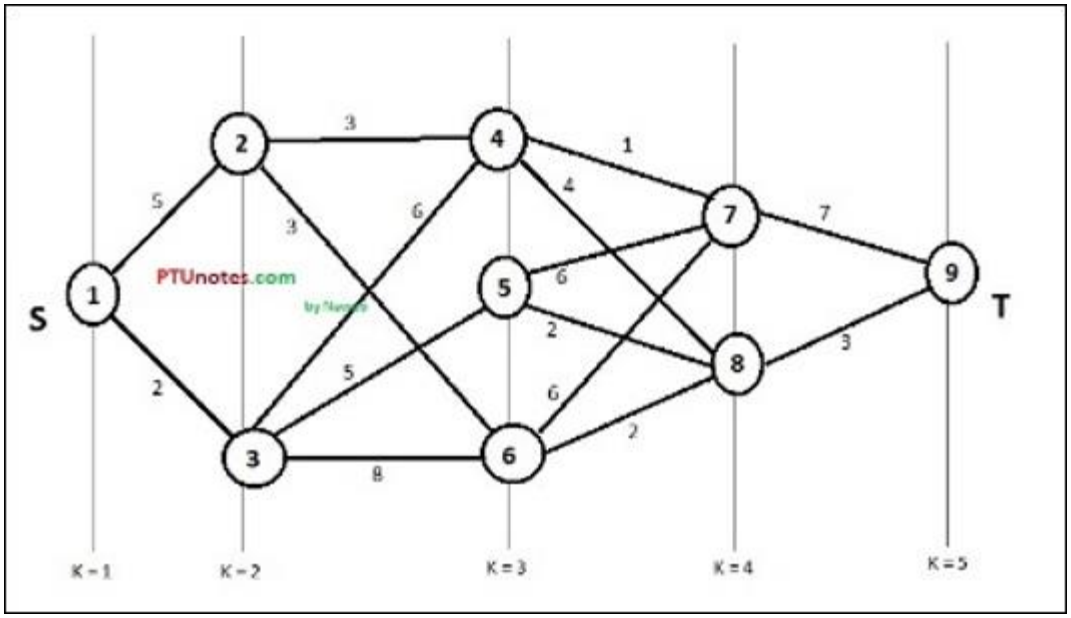
The vertex ***s* $\in s_1$** is called the **source** and the vertex ***t* $\in s_k$** is called **sink**.

G is usually assumed to be a weighted graph. In this graph, cost of an edge *(i, j)* is represented by *c(i, j)*. Hence, the cost of path from source ***s*** to sink ***t*** is the sum of costs of each edges in this path.

The multistage graph problem is finding the path with minimum cost from source ***s*** to sink ***t***.

Example

Consider the following example to understand the concept of multistage graph.



According to the formula, we have to calculate the cost **(i, j)** using the following steps

Step-1: Cost (K-2, j)

In this step, three nodes (node 4, 5. 6) are selected as **j**. Hence, we have three options to choose the minimum cost at this step.

Cost(3, 4) = min {c(4, 7) + Cost(7, 9),c(4, 8) + Cost(8, 9)} = 7

Cost(3, 5) = min {c(5, 7) + Cost(7, 9),c(5, 8) + Cost(8, 9)} = 5

Cost(3, 6) = min {c(6, 7) + Cost(7, 9),c(6, 8) + Cost(8, 9)} = 5

Step-2: Cost (K-3, j)

Two nodes are selected as j because at stage k - 3 = 2 there are two nodes, 2 and 3. So, the value i = 2 and j = 2 and 3.

Cost(2, 2) = min {c(2, 4) + Cost(4, 8) + Cost(8, 9),c(2, 6) +

Cost(6, 8) + Cost(8, 9)} = 8

Cost(2, 3) = {c(3, 4) + Cost(4, 8) + Cost(8, 9), c(3, 5) + Cost(5, 8)+ Cost(8, 9), c(3, 6) + Cost(6, 8) + Cost(8, 9)} = 10

Step-3: Cost (K-4, j)

Cost (1, 1) = {c(1, 2) + Cost(2, 6) + Cost(6, 8) + Cost(8, 9), c(1, 3) + Cost(3, 5) + Cost(5, 8) + Cost(8, 9))} = 12

c(1, 3) + Cost(3, 6) + Cost(6, 8 + Cost(8, 9))} = 13

Hence, the path having the minimum cost is **1→ 3→ 5→ 8→ 9**.

17. What is minimum spanning tree. Explain Prim's algorithm for computing minimum spanning tree.

Ans: A **Minimum Spanning Tree (MST)** is a subset of edges of a connected weighted undirected graph that connects all the vertices together with the minimum possible total edge weight. To derive an MST, Prim's algorithm or Kruskal's algorithm can be used.

Prims Algorithm:

It is a greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices:

- Contain vertices already included in MST.
- Contain vertices not yet included.

At every step, it considers all the edges and picks the minimum weight edge. After picking the edge, it moves the other endpoint of edge to set containing MST.

Steps for finding MST using Prim's Algorithm:

1. Create MST set that keeps track of vertices already included in MST.
2. Assign key values to all vertices in the input graph. Initialize all key values as INFINITE (∞). Assign key values like 0 for the first vertex so that it is picked first.
3. While MST set doesn't include all vertices.
    1. Pick vertex u which is not is MST set and has minimum key value. Include 'u' to MST set.
    2. Update the key value of all adjacent vertices of u. To update, iterate through all adjacent vertices. For every adjacent vertex v, if the weight of edge u.v less than the previous key value of v, update key value as a weight of u.v.

Q. 18 Solve the following using quick sort algorithm.

74, 25, 14, 66, 84, 53, 30, 48

→ Fix pivot point, low, and high pointers.
Keep moving low & high pointer until A[low] > A[pivot] and A[high] < A[pivot]
- and if low < high then } normal swap
  swap A[low] with A[high]
- and if low ≥ high then } if high and low crosses.
  swap A[pivot] with A[high]

Initially

74	25	14	66	84	53	30	48
pivot	low						high

Step 1

74	25	14	66	84	53	30	48
pivot		low					high

Step 2

74	25	14	66	84	53	30	48
pivot			low				high

Step 3

74	25	14	66	84	53	30	48
pivot				low			high

Step 4

74 25 14 66 48 53 30 84
pivot low high

Step 5

74 25 14 66 48 53 30 84
pivot low high

Step 6

74 25 14 66 48 53 30 84
pivot low high low

Step 7

~~74 25 14 66 48 53 3~~
30 25 14 66 48 53 [74] 84
left sub string right sub string

Solving left sub string.

Step 8

30 25 14 66 48 53
pivot low high

Step 9

30 25 14 66 48 53
pivot low high

Step 10

30 25 14 66 48 53
pivot high low high

Step 11

30 25 14 66 48 53
pivot high low

Step 12

14 25 [30] 66 48 53

left sub string — right sub string

Solving right sub string

Step 13

66 48 53
pivot low high

Step 14

66 48 53 ∞
pivot low high → low

Step 15

53 48 [66]
pivot low high

Step 16

48 [53]

Thus the final sorted list is

14, 25, 30, 48, 53, 66, 74, ~~88~~ 84

19. Write the Kruskal’s algorithm for minimum spanning tree. What is the complexity of Kruskal’s algorithm?

Ans: **Kruskal's Algorithm** is used to find the minimum spanning tree for a connected weighted graph. The main target of the algorithm is to find the subset of edges by using which we can traverse every vertex of the graph. It follows the greedy approach that finds an optimum solution at every stage instead of focusing on a global optimum. In Kruskal's algorithm, we start from edges with the lowest weight and keep adding the edges until the goal is reached. The steps to implement Kruskal's algorithm are listed as follows

- First, sort all the edges from low weight to high.
- Now, take the edge with the lowest weight and add it to the spanning tree. If the edge to be added creates a cycle, then reject the edge.
- Continue to add the edges until we reach all vertices, and a minimum spanning tree is created.

Algorithm

1. Step 1: Create a forest F in such a way that every vertex of the graph is a separate tree.
2. Step 2: Create a set E that contains all the edges of the graph.
3. Step 3: Repeat Steps 4 and 5 while E is NOT EMPTY and F is not spanning
4. Step 4: Remove an edge from E with minimum weight
5. Step 5: IF the edge obtained in Step 4 connects two different trees, then add it to the f orest F
6. (for combining two trees into one tree).
7. ELSE
8. Discard the edge
9. Step 6: END

The time complexity of Kruskal's algorithm is O(E logE) or O(V logV), where E is the no. of edges, and V is the no. of vertices.

20. Explain Branch and Bound with Travelling salesperson problem.

Ans: Branch and Bound builds the state space tree and finds the optimal solution quickly by pruning few of the tree branches which does not satisfy the bound. In Branch and Bound, all the children of E nodes are generated before any other live node becomes E node.

Branch And Bound technique in which E-node puts its children in the queue is called FIFO Branch & Bound Approach, and if E-node puts its children in stack it is called LIFO Branch & Bound Approach

Travelling Salesman Problem (TSP) is interesting problem. Problem is defined as "given n cities and distance between each pair of cities, find out the path which visits each city exactly once and come back to starting city, with constraint of minimizing the travelling distance."

Consider directed weighted graph G = (V, E), where node represents cities and weighted directed edges represents direction and distance between two cities.

1. Initially, graph is represented by cost matrix C, where
   C_{ij} = cost of edge, if there is a direct path from city i to city j
   C_{ij} = ∞, if there is no direct path from city i to city j.
2. Convert cost matrix to reduced matrix by subtracting minimum values from appropriate rows and columns, such that each row and column contains at least one zero entry.
3. Find cost of reduced matrix. Cost is given by summation of subtracted amount from the cost matrix to convert it in to reduce matrix.
4. Prepare state space tree for the reduce matrix
5. Find least cost valued node A by computing reduced cost node matrix with every remaining node. This is required to find next best move from current city.
6. If <i, j> edge is to be included, then do following :
   (a) Set all values in row i and all values in column j of A to ∞
   (b) Set A[j, i] = ∞
   (c) Reduce A again, except rows and columns having all ∞ entries.
7. Compute the cost of newly created reduced matrix as,
   Cost = L + Cost(i, j) + r
   Where, L is cost of reduced cost matrix and r is A[i, j].
8. If all nodes are not visited then go to step 4.

21. Explain multistage graph problem with suitable example.

Ans:

A multistage graph **G = (V, E)** is a directed graph where vertices are partitioned into **k** (where ***k* > 1**) number of disjoint subsets ***S* = {s_1,s_2,…,s_k}** such that edge *(u, v)* is in E, then *u* $\in s_i$ and *v* $\in s_{1+1}$ for some subsets in the partition and $|s_1| = |s_k| = 1$.

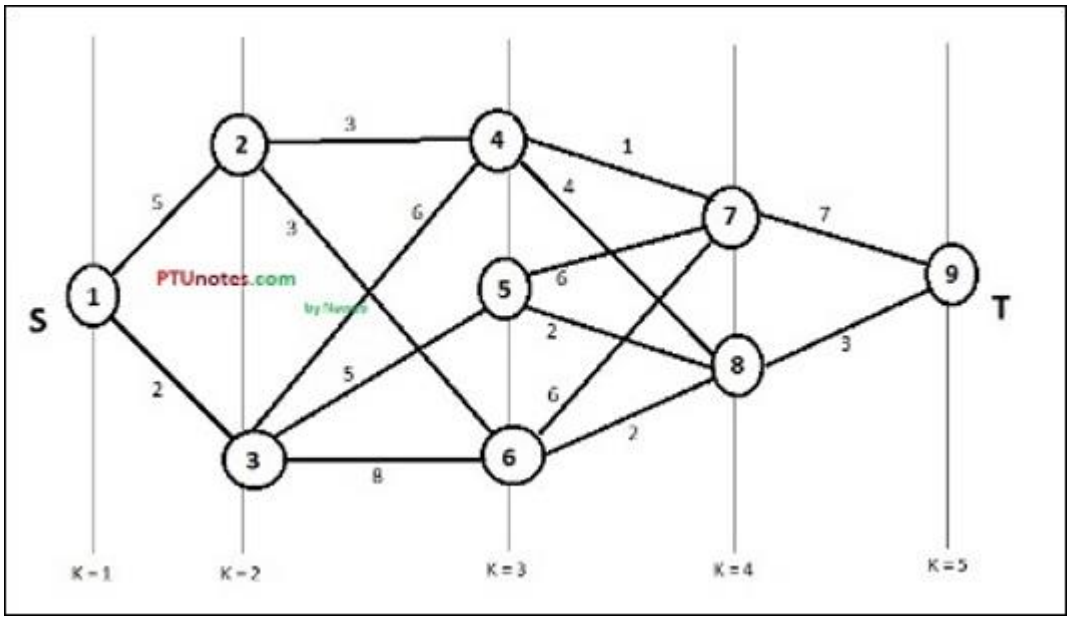
The vertex ***s* $\in$ s_1** is called the **source** and the vertex ***t* $\in$ s_k** is called **sink**.

G is usually assumed to be a weighted graph. In this graph, cost of an edge *(i, j)* is represented by *c(i, j)*. Hence, the cost of path from source ***s*** to sink ***t*** is the sum of costs of each edges in this path.

The multistage graph problem is finding the path with minimum cost from source ***s*** to sink ***t***.

Example

Consider the following example to understand the concept of multistage graph.



According to the formula, we have to calculate the cost **(i, j)** using the following steps

Step-1: Cost (K-2, j)

In this step, three nodes (node 4, 5. 6) are selected as **j**. Hence, we have three options to choose the minimum cost at this step.

Cost(3, 4) = min {c(4, 7) + Cost(7, 9),c(4, 8) + Cost(8, 9)} = 7

Cost(3, 5) = min {c(5, 7) + Cost(7, 9),c(5, 8) + Cost(8, 9)} = 5

Cost(3, 6) = min {c(6, 7) + Cost(7, 9),c(6, 8) + Cost(8, 9)} = 5

Step-2: Cost (K-3, j)

Two nodes are selected as j because at stage k - 3 = 2 there are two nodes, 2 and 3. So, the value i = 2 and j = 2 and 3.

Cost(2, 2) = min {c(2, 4) + Cost(4, 8) + Cost(8, 9),c(2, 6) +

Cost(6, 8) + Cost(8, 9)} = 8

Cost(2, 3) = {c(3, 4) + Cost(4, 8) + Cost(8, 9), c(3, 5) + Cost(5, 8)+ Cost(8, 9), c(3, 6) + Cost(6, 8) + Cost(8, 9)} = 10

Step-3: Cost (K-4, j)

Cost (1, 1) = {c(1, 2) + Cost(2, 6) + Cost(6, 8) + Cost(8, 9), c(1, 3) + Cost(3, 5) + Cost(5, 8) + Cost(8, 9))} = 12

c(1, 3) + Cost(3, 6) + Cost(6, 8 + Cost(8, 9))} = 13

Hence, the path having the minimum cost is **1→ 3→ 5→ 8→ 9**.

22. Write algorithm for binary search. Explain the algorithm with example

Ans:

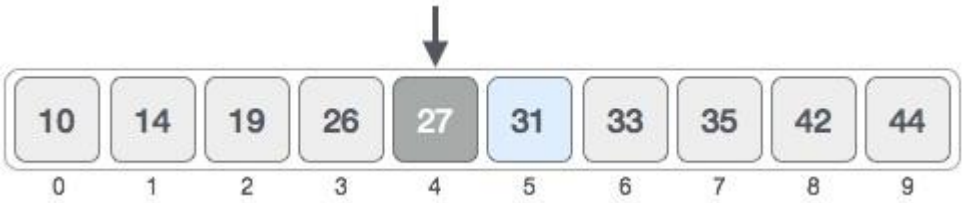
```
Procedure binary_search
   A ← sorted array
   n ← size of array
   x ← value to be searched

   Set lowerBound = 1
   Set upperBound = n

   while x not found
      if upperBound < lowerBound
         EXIT: x does not exists.

      set midPoint = lowerBound + ( upperBound - lowerBound ) / 2

      if A[midPoint] < x
         set lowerBound = midPoint + 1

      if A[midPoint] > x
         set upperBound = midPoint - 1

      if A[midPoint] = x
         EXIT: x found at location midPoint
   end while

end procedure
```

The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.

First, we shall determine half of the array by using this formula −

```
mid = low + (high - low) / 2
```

Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

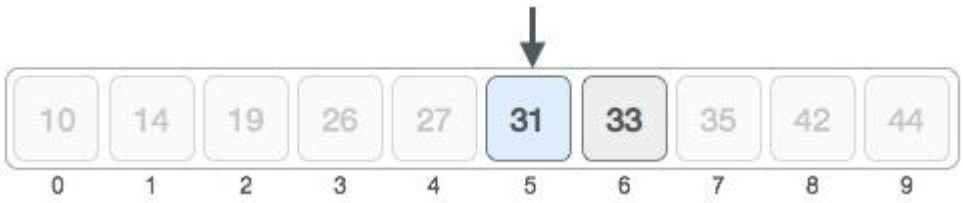
We change our low to mid + 1 and find the new mid value again.

```
low = mid + 1
mid = low + (high - low) / 2
```

Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.

Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.

We conclude that the target value 31 is stored at location 5.

23. Explain the difference between greedy approach and dynamic programming approach.

Ans:

Feature	**Greedy method**	**Dynamic programming**
Feasibility	In a greedy Algorithm, we make whatever choice seems best at the moment in the hope that it will lead to global optimal solution.	In Dynamic Programming we make decision at each step considering current problem and solution to previously solved sub problem to calculate optimal solution .
Optimality	In Greedy Method, sometimes there is no such guarantee of getting Optimal Solution.	It is guaranteed that Dynamic Programming will generate an optimal solution as it generally considers all possible cases and then choose the best.

Feature	Greedy method	Dynamic programming
Recursion	A greedy method follows the problem solving heuristic of making the locally optimal choice at each stage.	A Dynamic programming is an algorithmic technique which is usually based on a recurrent formula that uses some previously calculated states.
Memoization	It is more efficient in terms of memory as it never look back or revise previous choices	It requires dp table for memoization and it increases it's memory complexity.
Time complexity	Greedy methods are generally faster. For example, Dijkstra's shortest path algorithm takes O(ELogV + VLogV) time.	Dynamic Programming is generally slower. For example, Bellman Ford algorithm takes O(VE) time.
Fashion	The greedy method computes its solution by making its choices in a serial forward fashion, never looking back or revising previous choices.	Dynamic programming computes its solution bottom up or top down by synthesizing them from smaller optimal sub solutions.
Example	Fractional knapsack .	0/1 knapsack problem

24. Write a short note on Bellman Ford Algorithm.

Ans:

Bellman-Ford algorithm is used to find the shortest path from the source vertex to remaining all other vertices in the weighted graph.

It is slower compared to Dijkstra's algorithm but it can handle ***negative weights*** also.

If the graph contains negative-weight cycle (edge sum of the cycle is negative), it is not possible to find the minimum path, because on every iteration of cycle gives a better result.

Bellman-Ford algorithm can detect a negative cycle in the graph but it cannot find the solution for such graphs.

Like Dijkstra, this algorithm is also based on the principle of relaxation. The path is updated with better values until it reaches the optimal value.

Dijkstra uses a priority queue to greedily select the closest vertex and perform relaxation on all its adjacent outgoing edges. By contrast, Beliman-Ford relaxes all the edges |V| - 1 time. Bellman-Ford runs in O(|V|.|E|) time.

Page No.
Date.

Q.25 Solve the following using master method.
i. $T(n) = 8T(n/2) + n^2$
ii. $T(n) = 4T(n/2) + n\log n$

→ Master method
Let $a \geq 1$, $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the non-negative integers by the recurrence

$$T(n) = aT(n/b) + f(n)$$

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(f(n))$.

i) $T(n) = 8T(n/2) + n^2$
Here $a = 8$, $b = 2$ and $f(n) = n^2$
L.H.S : $f(n) = n^2$
R.H.S : $n^{\log_b a}$
$= n^{\log_2 8}$
$= n^3$
Let ϵ be 1 here
$\therefore = n^2$

$\therefore$ L.H.S = R.H.S

Since $f(n) = O(n^{\log_2 8 - \epsilon})$
$\therefore$ By case 1 $T(n) = \Theta(n^3)$.

ii) $T(n) = 4T(n/2) + n\log n$

Here $a=4$, $b=2$ and $f(n) = n\log n$

Thus we apply $n^{\log_b a} = n^{\log_2 4}$

$= n^2$

The extended master's Theorem.

$T(n) = a\,T(n/b) + \Theta(n^k \log^p n)$,

where $a \geq 1, b > 1, k \geq 0$ and p is real number

1) If $a > b^k$, then $T(n) = \Theta(n^{\log_b a})$
2) If $a = b^k$
   a) If $p > -1$, then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$
   b) If $p = -1$, then $T(n) = \Theta(n^{\log_b a} \log\log n)$
   c) If $p < -1$, then $T(n) = \Theta(n^{\log_b a})$
3) If $a < b^k$
   a) If $p \geq 0$, then $T(n) = \Theta(n^k \log^p n)$
   b) If $p < 0$, then $T(n) = \Theta(n^k)$

ii) $T(n) = 4\,T(n/2) + n\log n$

Here $a=4, b=2, k=1, p=1$

Here $4 > 2^1$ i.e $a > b^k$

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

$= \Theta(n^{\log_2 4})$

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

26. Explain the different methods used to solve recurrence equations.

Ans:

Substitution Method: We make a guess for the solution and then we use mathematical induction to prove the guess is correct or incorrect.

```
For example consider the recurrence T(n) = 2T(n/2) + n

We guess the solution as T(n) = O(nLogn). Now we use induction
to prove our guess.

We need to prove that T(n) <= cnLogn. We can assume that it is true
for values smaller than n.

T(n) = 2T(n/2) + n
    <= 2cn/2Log(n/2) + n
    =  cnLogn - cnLog2 + n
    =  cnLogn - cn + n
    <= cnLogn
```

Recurrence Tree Method: In this method, we draw a recurrence tree and calculate the time taken by every level of tree. Finally, we sum the work done at all levels. To draw the recurrence tree, we start from the given recurrence and keep drawing till we find a pattern among levels. The pattern is typically a arithmetic or geometric series.

```
For example consider the recurrence relation
T(n) = T(n/4) + T(n/2) + cn^2

           cn^2
         /      \
     T(n/4)     T(n/2)

If we further break down the expression T(n/4) and T(n/2),
we get following recursion tree.

                cn^2
           /            \
       c(n^2)/16      c(n^2)/4
      /      \          /     \
  T(n/16)     T(n/8)  T(n/8)    T(n/4)
Breaking down further gives us following
                 cn^2
             /            \
       c(n^2)/16          c(n^2)/4
       /      \            /      \
c(n^2)/256   c(n^2)/64  c(n^2)/64    c(n^2)/16
 /    \      /    \    /    \        /    \

To know the value of T(n), we need to calculate sum of tree
nodes level by level. If we sum the above tree level by level,
we get the following series
T(n)  = c(n^2 + 5(n^2)/16 + 25(n^2)/256) + ....
The above series is geometrical progression with ratio 5/16.
```

To get an upper bound, we can sum the infinite series.
We get the sum as $(n^2)/(1 - 5/16)$ which is $O(n^2)$

Master Method:
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 $f(n) = O(n^c)$ 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 $f(n) = \Omega(n^c)$ where $c > \text{Log}_b a$ then $T(n) = \Theta(f(n))$

Q.27 Determine the LCS of the following sequences.

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

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

→

if (A[i] = B[j])

LCS[i,j] = 1 + LCS[i-1, j-1]

else

LCS[i,j] = max(LCS[i-1, j], LCS[i, j-1]).

		B	D	C	A	B	A
	0	1	2	3	4	5	6
∅	0	0	0	0	0	0	0
A	1	0	↑0	↑0	↑0	1	1	1
B	2	0	1	1	1	↑1	2	2
C	3	0	↑1	↑1	2	2	2	↑2
B	4	0	1	↑1	↑2	↑2	3	3
D	5	0	↑1	2	↑2	↑2	↑3	↑3
A	6	0	↑1	↑2	↑2	3	↑3	4
B	7	0	1	↑2	↑2	↑3	4	↑4

B C B A

Each (↖) on the shaded entry correspond to an entry highlighted for which (xi = yj) is a member of an LCS.

Thus the LCS sequence is BCBA.

28. Explain Single source shortest path algorithm using dynamic programming approach. Explain how it is different from Dijkstra's greedy approach.

Ans:

```
Algorithm BELLMAN-FORD (G)
// Description : Find the shortest path from source vertex to all
      other vertices using Bellman-Ford algorithm
// Input : Weighted graph G = (V, E, W)
          w(u, v) : Weight of edge (u, v)
          d[u] : distance of vertex u from source
          π[u] : Successor of vertex u
// Output : Shortest distance of each vertex from given source
     , vertex.

// Initialization
for each v ∈ V do
     d[v] ←∞
     π[v] ← NULL
end
d[s] ← 0

// Relaxation
for i ← 1 to |V| - 1 do
     for each edge (u, v) ∈ E do
          if d[u] + w (u, v) < d[v] then
               d[v] ← d[u] + w(u, v)
               π[v] ← u
          end
     end
end

// Check for negative cycle
for each edge (u, v) ∈ E do
     if d[u] + w(u, v) < d[v] then
      error "Graph contains negative cycle"
     end
end
return d, π
```

Ex. 4.6.1

Find the shortest path from source vertex 5 for given graph.

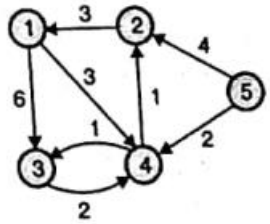
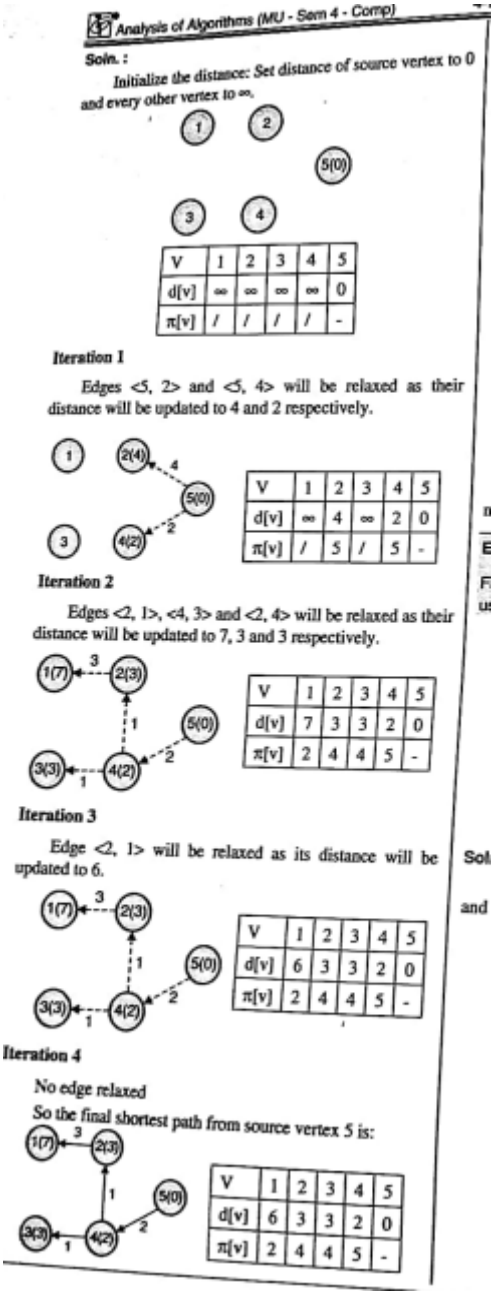


Fig. P. 4.6.1



Analysis of Algorithms (MU - Sem 4 - Comp)

Soln. :

Initialize the distance: Set distance of source vertex to 0 and every other vertex to ∞.

V	1	2	3	4	5
d[v]	∞	∞	∞	∞	0
π[v]	/	/	/	/	-

Iteration 1

Edges <5, 2> and <5, 4> will be relaxed as their distance will be updated to 4 and 2 respectively.

V	1	2	3	4	5
d[v]	∞	4	∞	2	0
π[v]	/	5	/	5	-

Iteration 2

Edges <2, 1>, <4, 3> and <2, 4> will be relaxed as their distance will be updated to 7, 3 and 3 respectively.

V	1	2	3	4	5
d[v]	7	3	3	2	0
π[v]	2	4	4	5	-

Iteration 3

Edge <2, 1> will be relaxed as its distance will be updated to 6.

V	1	2	3	4	5
d[v]	6	3	3	2	0
π[v]	2	4	4	5	-

Iteration 4

No edge relaxed

So the final shortest path from source vertex 5 is:

V	1	2	3	4	5
d[v]	6	3	3	2	0
π[v]	2	4	4	5	-

Shortest Path form any vertex can be found out by starting from destination vertex and following its predecessor π back to the source. For Example starting at vertex 1 u1. π = 2, u2. π = 4, u4. π = 5 Thus the shortest path from vertex is [5,4,2,1].

Bellman-Ford algorithm is used to find the shortest path from the source vertex to remaining all other vertices in the weighted graph.

It is slower compared to Dijkstra's algorithm but it can handle ***negative weights*** also.

If the graph contains negative-weight cycle (edge sum of the cycle is negative), it is not possible to find the minimum path, because on every iteration of cycle gives a better result.

Bellman-Ford algorithm can detect a negative cycle in the graph but it cannot find the solution for such graphs.

Like Dijkstra, this algorithm is also based on the principle of relaxation. The path is updated with better values until it reaches the optimal value.

Dijkstra uses a priority queue to greedily select the closest vertex and perform relaxation on all its adjacent outgoing edges. By contrast, Beliman-Ford relaxes all the edges |V| - 1 time. Bellman-Ford runs in O(|V|.|E|) time.

Q.29 Explain and apply Naivee String matching on following strings

String 1: COMPANION

String 2: PANI

→

Here we will maintain ti and pj are indices of text and pattern & respectively.

Step 1: T[i] ≠ P[j], so advance text pointer, t++

```
1 2 3 4 5 6 7 8 9
C O M P A N I O N
↕
P A N I
1 2 3 4
```

Step 2: T[2] ≠ P[j], t++

```
1 2 3 4 5 6 7 8 9
C O M P A N I O N
  ↕
  P A N I
  1 2 3 4
```

Step 3: T[3] ≠ P[j], t++

```
1 2 3 4 5 6 7 8 9
C O M P A N I O N
    ↕
    P A N I
    1 2 3 4
```

Step 4: T[4] = P[j], advance both pointer, t++, p++

```
1 2 3 4 5 6 7 8 9
C O M P A N I O N
      ↕
      P A N I
      1 2 3 4
```

Page No.
Date.

Step 5: T[5] = P[2], t++, p++

```
1 2 3 4 5 6 7 8 9
C O M P A N I O N
      ↕
      P A N I
      1 2 3 4
```

Step 6: T[6] = P[3], t++, p++

```
1 2 3 4 5 6 7 8 9
C O M P A N I O N
        ↕
        P A N I
        1 2 3 4
```

Step 7: T[7] = P[4], t++, p++

```
1 2 3 4 5 6 7 8 9
C O M P A N I O N
          ↕
          P A N I   Match found
          1 2 3 4
```

- This is simple and inefficient brute force approach. It compares first character of pattern with searchable text, If match is found, pointer in both string are advanced. If not found, pointer of text is incremented & pattern is reset.
- Given text T and Pattern P, it directly starts comparing both strings character by character.
- After each comparison it shifts pattern string one position to right.

30. Explain assembly line scheduling problem with example.

Ans: Assembly Line Scheduling is a manufacturing problem. In automobile industries, assembly lines are used to transfer parts from one station to another. For Eg: Manufacturing a car may be done in several stages, the particular task is carried out at the station dedicated to that task only. Based on requirements, there may be more than one assembly line. In case of 2 assembly lines, Determining which station should be selected from assembly line 1 and which to choose from assembly line 2 in order to minimize the total time to build the entire product.

General architecture of two assembly lines with n stations is shown in Fig. 4.8.1. Terminologies are explained here :

- S_{ij} = Station j on assembly line i.
- a_{ij} = Time needed to assemble the partial component at station j on assembly line i.

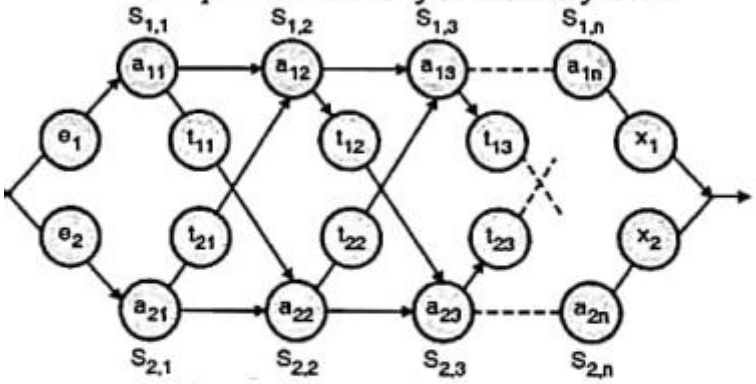
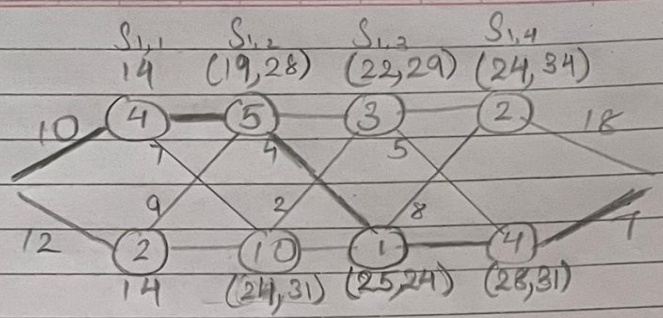


Fig. 4.8.1 : Architecture of assembly line

- t_{ij} = Time required to transfer component from one assembly to other from station j to (j + 1).
- e_i = Entry time on assembly line i.
- x_i = Exit time from assembly line i.



	1	2	3	4	
F_{1j}	14	19	22	24	
F_{2j}	14	24	29	27	35

	2	3	4	
L_{1j}	1	1	1	
L_{2j}	2	1	2	2

The path was $e_1 \rightarrow S_{1,1} \rightarrow S_{1,2} \rightarrow S_{2,3} \rightarrow S_{2,4} \rightarrow x_2$

The minimum time was 35.

31. Write an algorithm to find min and max number using divide and conquer strategy

Ans:

a. Let P = (n, a [i],……,a [j]) denote an arbitrary instance of the problem.

b. Here 'n' is the no. of elements in the list (a [i],….,a[j]) and we are interested in finding the maximum and minimum of the list.

c. If the list has more than 2 elements, P has to be divided into smaller instances.

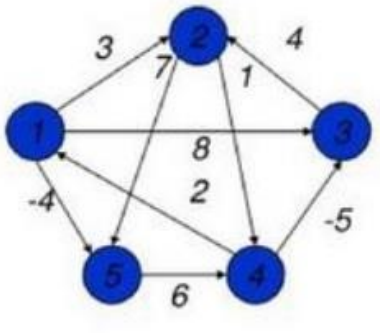
d. For example, we might divide 'P' into the 2 instances, P1=([n/2],a[1],……..a[n/2]) & P2= (n-[n/2], a[[n/2]+1],….., a[n]) After having divided 'P' into 2 smaller sub problems, we can solve them by recursively invoking the same divide-and-conquer algorithm.

```
MaxMin (i, j, max, min)
 // a [1: n] is a global array, parameters i& j are integers, 1<= i<=j <=n. The effect is
to4.
// Set max & min to the largest & smallest value 5 in a [i: j], respectively.
{
If (i=j) then Max = Min = a[i];
Else if (i=j-1) then
{
if (a[i] < a[j]) then
        {
                Max = a[j];
                Min = a[i];
        }
        Else
        {
                Max = a[i];
                Min = a[j];
        }
}Else
{
Mid = (i + j) / 2;
MaxMin (I, Mid, Max, Min);
MaxMin (Mid +1, j, Max1, Min1);
If (Max < Max1) then Max = Max1;
If (Min > Min1) then Min = Min1;
}}
The procedure is initially invoked by the statement, MaxMin (1, n, x, y)
```

32. Write a short note on All pairs shortest path algorithm.

Ans:

The all pair shortest path algorithm is also known as Floyd-Warshall algorithm is used to find all pair shortest path problem from a given weighted graph. As a result of this algorithm, it will generate a matrix, which will represent the minimum distance from any node to all other nodes in the graph.



$$\begin{bmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix}$$

At first the output matrix is same as given cost matrix of the graph. After that the output matrix will be updated with all vertices k as the intermediate vertex.

The time complexity of this algorithm is O(V3), here V is the number of vertices in the graph.

Input − The cost matrix of the graph.

Output − Matrix of all pair shortest path.

33. Rewrite and Compare Rabin Karp and Knuth Morris Pratt Algorithms

Ans:

Rabin Karp	Knuth Morris Pratt
Comparing numbers is easier and cheaper than comparing strings. Rabin Karp algorithm represents strings in numbers. Suppose p represents values corresponding to pattern P[1...m] of length m. And t_s represents values of m-length substrings T[(s + 1) ... (s + m)] for s = 0, 1, 2, ..., n – m. We can compute p in O(m) time and all t_s can be computed in O(n – m + 1) time. Rabin Karp algorithm is based on hashing technique. It first computes the hash value of p and t_s. If hash values are same, i.e if hash(p) = hash(t_s), we check the equality of inverse hash similar to naïve method. If hash values are not same, no need to compare actual string. On hash match, actual characters of both strings are compared using brute force approach. If pattern is found then it is called *hit*. Otherwise it is called *spurious hit*. For example, let us consider the hash value of string T = ABCDE is 38 and hash of string P = ABCDX is 71. Clearly, hash values are not same, so strings cannot be same. Brute force approach does five comparisons where as Rabin Karp dose only one comparison. However, same hash value does not ensure the string match. Two different strings can have same hash values. That is why we need to compare them character by character on hash hit.	This algorithm is also known as KMP (Knuth-Morris-Pratt) algorithm. This is the first linear time algorithm for string matching. It utilizes the concept of naïve approach in some different way. This approach keeps track of matched part of pattern. Main idea of the algorithm is to avoid computation of transition function δ and reducing useless shifts performed in naïve approach. By maintaining the information of already processed string, this approach reduce the time to O(m + n). That is, in worst case, KMP algorithm examines all characters of input pattern and text exactly once. This improvement is achieved by using auxiliary function π. Naïve approach was not storing anything it has already matched, and that's why it was shifting the pattern right side by just one. Computation of auxiliary function π take O(m) time. Function π is very useful, it computes the number of shifts of pattern by finding pattern matches within itself.

KMP Algorithm:

Algorithm

findPrefix(pattern, m, prefArray)

Input − The pattern, the length of pattern and an array to store prefix location

Output − The array to store where prefixes are located

```
Begin
   length := 0
   prefArray[0] := 0

   for all character index 'i' of pattern, do
      if pattern[i] = pattern[length], then
         increase length by 1
         prefArray[i] := length
      else
         if length ≠ 0 then
            length := prefArray[length - 1]
            decrease i by 1
         else
            prefArray[i] := 0
   done
End
```

kmpAlgorithm(text, pattern)

Input: The main text, and the pattern, which will be searched

Output − The location where patterns are found

```
Begin
   n := size of text
   m := size of pattern
   call findPrefix(pattern, m, prefArray)

   while i < n, do
      if text[i] = pattern[j], then
         increase i and j by 1
      if j = m, then
         print the location (i-j) as there is the pattern
         j := prefArray[j-1]
      else if i < n AND pattern[j] ≠ text[i] then
         if j ≠ 0 then
            j := prefArray[j - 1]
         else
            increase i by 1
   done
End
```

Rabin Karp Algorithm

Algorithm

```
rabinKarpSearch(text, pattern, prime)
```

Input − The main text and the pattern. Another prime number of find hash location

Output − location where patterns are found

```
Begin
   patLen := pattern Length
   strLen := string Length
   patHash := 0 and strHash := 0, h := 1
   maxChar := total number of characters in character set

   for index i of all character in pattern, do
      h := (h*maxChar) mod prime
   done

   for all character index i of pattern, do
      patHash := (maxChar*patHash + pattern[i]) mod prime
      strHash := (maxChar*strHash + text[i]) mod prime
   done

   for i := 0 to (strLen - patLen), do
      if patHash = strHash, then
         for charIndex := 0 to patLen -1, do
            if text[i+charIndex] ≠ pattern[charIndex], then
               break the loop
         done

         if charIndex = patLen, then
            print the location i as pattern found at i position.
      if i < (strLen - patLen), then
         strHash := (maxChar*(strHash – text[i]*h)+text[i+patLen]) mod
prime, then
      if strHash < 0, then
         strHash := strHash + prime
   done
End
```