Algorithms (Pintu Sir)

☐ What is an algorithm?

Ans: An algorithm is a set of instructions that is designed to perform a specific task or solve a specific problem.

☐ Why is the order of an algorithm generally more important than the speed of the processor?

Ans:

The order of an algorithm, or its complexity, refers to how long it takes for the algorithm to run as a function of the input size. On the other hand, the speed of the processor refers to how quickly it can execute instructions.

Generally, the complexity of an algorithm is more important than the speed of the processor because the running time of an algorithm is

Algorithms

determined by the algorithm itself, not the speed of the processor. An algorithm with a higher complexity will always take longer to run than an algorithm with a lower complexity, regardless of the speed of the processor.

For example,

Sorting n numbers, where $n = 10^6$

One's computer speed $= 10^9$ ins./sec.

One's algorithm $= 2n^2$ ins.

And,

Another computer speed $= 10^7$ ins/sec

Another algorithm $= 50 n \log(n)$ ins.

Now, time for one's $= \frac{2 \times 10^{12}}{10^9} = 2000$ sec.

" " another $= \frac{50 \times 10^6 \times \log_2(10^6)}{10^7} = 100$ sec.

So, another algorithm is better. So, the complexity of the algorithm is a more significant factor in determining running time.

⊞ What are the differences between performance analysis and performance measurement of an algorithm?

Ans:

Performance analysis and performance measurement are two different aspects of evaluating the performance of an algorithm.

Performance analysis is the process of evaluating the performance of an algorithm in terms of its efficiency, correctness and other metrics. This can involve analyzing the algorithm's complexity, examining its inputs and outputs, and comparing it to other algorithms.

On the other hand, performance measurement involves actually running the algorithm and collecting data on its performance. This can involve measuring how long it takes to complete,

how much memory it uses, and other relevant metrics. Performance measurement is typically used to validate the results of performance analysis and to compare the performance of different algorithms in a more empirical way.

In summary, performance analysis is the process of understanding how an algorithm works and why it performs in a certain way, while performance measurement is the process of collecting data on the actual performance of an algorithm.

⊞ Basic characteristics of an algorithm / basic criteria that must satisfy :-

Ans:

1. Input: An algorithm must have some input on which it operates.

2. Output: An algorithm must have at least one output.

3. Definiteness: Each instruction is clear and unambiguous.

4. Finiteness: An algorithm must have a finite number of steps, so that it can be completed in a finite amount of time.

5. Effectiveness: Each of its steps must be able to be carried out in a finite amount of time

6. Generality: An algorithm should be able to solve a class of problems.

7. Optimality: It produces the best possible output for a given input in the least amount of time.

Define running time of an algorithm. Discuss some running time functions of an algorithm.

Ans:

The running time of an algorithm is the amount of time it takes for the algorithm to complete its task on a particular input. It is typically measured in terms of the number of basic operations the algorithm performs, such as comparisons, assignments and arithmetic operations.

The running time of an algorithm can depend on various factors, including the size of the input, the specific input values, and the hardware and software environment in which it is running.

There are several common functions that are used to describe the running time of an algorithm. These include:-

1. Constant time: Denoted by $O(1)$, has a running time that does not depend on the size of the input.

2. Logarithmic time: Denoted by $O(\log n)$, has a running time that grows logarithmically with the size of the input.

3. Linear time: Denoted by $O(n)$, proportional to the size of the input.

4. Quadratic time: Denoted by $O(n^2)$

5. Cubic time: Denoted by $O(n^3)$

6. Exponential time: Denoted by $O(2^n)$, increases exponentially with the size of the input.

7. Factorial time: Denoted by $O(n!)$.

How an algorithm can be devised. Explain with example.

Ans:

There are many ways to devise an algorithm for a given problem. The approaches are:-

(i) Understand the problem.

(ii) Identify the key steps.

(iii) Organize the steps.

(iv) Test the algorithm.

(v) Optimize the algorithm.

For example, Let, given a list of numbers, find the largest number in the list.

1. Understand the problem - The input is a list of numbers and the output is the largest number in the list.

2. Identify the key steps -

(i) Initialize a variable to store the largest number.

(ii) Iterate through the list of numbers.

(iii) Compare each number to the current largest number.

(iv) If a larger number is found, update the largest number.

3. Organize the steps -

(i) Initialized largest number to the first number in the list.

(ii) Compare.

(iii) return largest number.

4. Test the algorithm -

if the list : [1, 5, 2, 8, 3]

the algorithm should return 8.

5. Optimize the algorithm -

There may not be any further optimization needed for this algorithm.

Define and explain different asymptotic notations used to analyze the algorithms.

Ans:

Asymptotic notation is a mathematical notation used to describe the behavior of an algorithm as the input size grows indefinitely. It is a useful tool for comparing the efficiency of different algorithms and for understanding how well an algorithm scales with larger input sizes. Some common asymptotic notations are:-

1. Big "oh" (O) notation: Big O notation is used to describe the upper bound of an algorithm's running time. It represents the worst case scenario for the algorithm.

If $f(n) = a_m n^m + \cdots + a_1 n + a_0$, then $f(n) = O(n^m)$

Running time $O(n)$ means, the algorithm's running time grows linearly with the size of the input(n).

2. Big [Omega] Ω notation: It is used to describe the lower bound of an algorithm's running time. It represents the best case scenario for the algorithm.

If $f(n) = a_m n^m + \cdots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Omega(n^m)$.

3. Big [Theta] Θ notation: It is used to describe the average case scenario for an algorithm's running time.

If $f(n) = a_m n^m + \cdots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Theta(n^m)$.

4. Little "oh" o notation: It represents the upper bound of an algorithm's running time, but it ignores constant factors.

5. Little "omega" ω notation: It represents the lower bound of an algorithm's running time, but it ignores constant factors.

What is a pre-condition? Write down the pre-conditions of Bellman-Ford algorithm.

Ans:

A pre-condition is a condition that must be true before an algorithm or function is executed. It is a requirement that must be met in order for the algorithm or function to work correctly.

Some common pre-conditions for the Bellman-Ford algorithm are:-

1. The input graph must be a weighted, directed graph.
2. The weights of the edges in the graph must be real numbers (positive, negative or zero).

3. The graph must not contain any negative weight cycles. If the graph contains a negative weight cycle, the algorithm will not work correctly and may produce incorrect results.

4. The source vertex must be specified.

Define BFS and DFS. What are the time complexity of BFS and DFS of a graph?

Ans:

BFS: BFS (Breadth-First Search) is a graph traversal algorithm that starts at the root node and explores all the neighboring nodes first, before moving on to the next level neighbors. It uses a queue data structure to store the nodes that are waiting to be visited.

DFS: DFS (Depth-First Search) is a graph traversal algorithm that starts at the root node and explores as far as possible along each branch before backtracking. It uses a stack data structure to store the nodes that are waiting to be visited.

The time complexity of BFS and DFS on a graph depends on the number of vertices and edges in the graph.

For adjacency list representation,

Time complexity of BFS is $O(V+E)$ and DFS is $O(V+E)$ where V is the number of vertices and E is the number of edges.

For adjacency matrix representation,

Time complexity for both is $O(V^2)$.

Give the best Big-oh (O) characterization for each of the following running time estimates of the algorithms :-

(i) $f(n) = \log(n) + 1000$

Now,

$$\log(n) + 1000 \leq \log(n) + 1000 \log(n)$$

$$\Rightarrow \log(n) + 1000 \leq 1001 \log(n)$$

$$\therefore f(n) \leq c \cdot g(n)$$

$\therefore$ Best big-oh : $O(\log n)$

(ii) $f(n) = 2^{10} + 3^5$

Now,

$$2^{10} + 3^5 \leq 2(2^{10} + 3^5) \cdot n^0$$

$$\Rightarrow 2^{10} + 3^5 \leq 2(2^{10} + 3^5) \cdot 1$$

$$\Rightarrow f(n) \leq c \cdot g(1)$$

$\therefore$ Best big-oh : $O(1)$

(iii) $f(n) = n\log(n) + 15n + 0.002n^2$

Now,

$$n\log(n) + 15n + 0.002n^2 \leq n^2 + 15n^2 + n^2$$

$$\Rightarrow n\log(n) + 15n + 0.002n^2 \leq 17n^2$$

$$\Rightarrow f(n) \leq c \cdot g(n)$$

$\therefore$ best big-oh: $O(n^2)$

(iv) $f(n) = (n+1)^3$

Now,

$$(n+1)^3 \leq 3n^3$$

$$\Rightarrow f(n) \leq c \cdot g(n)$$

$\therefore$ best big-oh: $O(n^3)$

(v) $f(n) = n! = n \times (n-1) \times (n-2) \times \cdots \times 3 \times 2 \times 1$

Now,

$$n \times (n-1) \times \cdots \times 2 \times 1 \leq n \times n \times n \times n \cdots \times n$$

$$\Rightarrow n \times (n-1) \times \cdots \times 2 \times 1 \leq n^n$$

$$\Rightarrow f(n) \leq c \cdot g(n)$$

$\therefore$ best big-oh: $O(n^n)$

(VI) $6 \cdot 2^n + n^2$

Now,

$$6 \cdot 2^n + n^2 \leq 8 \cdot 2^n + 2^n$$

$$\Rightarrow 6 \cdot 2^n + n^2 \leq 9 \cdot 2^n$$

$$\Rightarrow f(n) \leq c \cdot g(n)$$

$\therefore$ best big-oh: $O(2^n)$

(VII) $f(n) = \log(n!)$

Now,

$$\log(1 \times 2 \times 3 \times \cdots \times n) \leq \log(n \times n \times \cdots \times n)$$

$$\Rightarrow \log(n!) \leq \log n^n$$

$$\Rightarrow \log(n!) \leq n \log n$$

$$\Rightarrow f(n) \leq c \cdot g(n)$$

$\therefore$ best big-oh: $O(n \log n)$

(VIII) $f(n) = 1000n^2 + 100n - 6$

Now,

$1000n^2 + 100n - 6 \leq 1000n^2 + 100n^2 - 6n^2$

$\Rightarrow 1000n^2 + 100n - 6 \leq 1094n^2$

$\Rightarrow f(n) \leq c \cdot g(n)$

$\therefore$ best big-oh: $O(n^2)$

(IX) $f(n) = 1000n^2 + 16n + 2^n$

Now,

$1000n^2 + 16n + 2^n \leq 1000\,2^n + 16\,2^n + 2^n$

$\Rightarrow 1000n^2 + 16n + 2^n \leq 1017\,2^n$

$\Rightarrow f(n) \leq c \cdot g(n)$

$\therefore$ best big-oh: $O(2^n)$

(X) $f(n) = 50n + n\log(n^2) + 1000\log(n)$

Now,

$50n + n\log(n^2) + 1000\log(n) \leq 50n\log(n) + n\log(n^2) + 1000n\log(n)$

$\Rightarrow 50n + n\log(n^2) + 1000\log(n) \leq 1052\, n\log(n)$

$\Rightarrow f(n) \leq c \cdot g(n)$

$\therefore$ best big-oh: $O(n \log n)$

(XI) $f(n) = n+(n-1)+(n-2)+\cdots+3+2+1$

Now,

$$n+(n-1)+\cdots+2+1 \leq n+n+n+\cdots+n$$

$$\Rightarrow \frac{n(n+1)}{2} \leq n^2$$

$$\Rightarrow \frac{n^2}{2} + \frac{n}{2} \leq n^2+n^2$$

$$\Rightarrow \frac{n^2}{2} + \frac{n}{2} \leq 2n^2$$

$$\Rightarrow f(n) \leq c \cdot g(n)$$

$\therefore$ best big-oh : $O(n^2)$

(XII) $f(n) = \dfrac{6n^3}{(\log n+1)}$

Now,

Let, $\log n \geq 1$

$$\frac{6n^3}{(\log n+1)} \leq \frac{6n^3}{1+1}$$

$$\frac{1}{\log n} \leq 1$$

$$\Rightarrow \frac{6n^3}{(\log n+1)} \leq 3n^3$$

$$\Rightarrow f(n) \leq c \cdot g(n)$$

$\therefore$ best big-oh : $O(n^3)$

(xiii) $f(n) = \sum_{i=1}^{n} i$

$= 1+2+3+ \cdots + n$

Now,

$1+2+3+ \cdots + n \leq n+n+n+ \cdots + n$

$\Rightarrow \frac{n(n+1)}{2} \leq n^2$

$\Rightarrow \frac{n^2+n}{2} \leq n^2$

$\Rightarrow \frac{n^2}{2} + \frac{n}{2} \leq n^2+n^2$

$\therefore f(n) \leq c \cdot g(n)$

$\therefore$ Best big-oh: $O(n^2)$

Describe the steps in applying dynamic programming strategy when developing an algorithm.

Ans:

Two main properties of a problem suggest that the given problem can be solved using dynamic programming. These are overlapping sub-problems and optimal substructure.

The steps for developing an algorithm using dynamic programming are:-

1. Identify the problem and determine if it is suitable for dynamic programing.
2. Determine the order in which the sub-problems should be solved. In DP, subproblems are typically solved in a bottom-up fashion.

3. Develop a recursive solution to the problem, solving the subproblems as needed.

4. Use memoization to store the solutions to the subproblems as they are solved, so that they can be reused later.

5. Convert the recursive solution to an iterative solution.

6. Test the algorithm thoroughly to ensure it is correct and performs efficiently.

What are the differences between branch-and-bound and backtracking paradigm?

Ans:

Branch-and-Bound	Backtracking
1. It is used to solve optimization problem.	1. It is used to find all possible solutions available to a problem.
2. It may traverse the tree in any manner, DFS or BFS.	2. It traverse the state space tree by DFS manner.
3. It realizes that it already has a better optimal solution that the pre-solution leads to so it abandons that pre-solution.	3. It realizes that it has made a bad choice and undoes the last choice by backing up.
4. It involves a bounding function.	4. It involves feasibility function.
5. Generally more efficient.	5. Less efficient
6. Difficult to implement	6. Simpler to implement.

Difference between Brute-force and backtracking paradigm.

Ans:-

Brute-Force	Backtracking
1. Brute-force algorithms do not require any special data structures or techniques to solve a problem.	1. Backtracking algorithms often make use of recursion and may require additional data-structure such as stacks.
2. Brute-force algorithms generally less efficient.	2. Backtracking algorithms can be more efficient for certain types of problems.
3. Brute-force algorithms try all possible solutions to a problem and return the best one.	3. It works by trying different possibilities, one at a time, until they find a solution.

Write down the control abstraction of greedy method.

Ans:

In the case of greedy algorithm, the control abstraction can be described as follows:-

1. Identify the problem to be solved and define the input and output of the algorithm.
2. Identify the greedy choice property, which is a property that allows to make a locally optimal choice at each step that leads to a globally optimal solution.
3. For each step of the algorithm, make a locally optimal choice based on the greedy choice property.
4. At the end of the algorithm, return the globally optimal solution that has been built up from the locally optimal choices made at each step.

5. Repeat the process as necessary until the problem is fully solved.

Ans:

Write pseudo code for a backtracking algorithm to solve m coloring problem:

Pseudo code:

```
function m_coloring (graph, m, colors, V, c)
    if V is the last vertex
        return true
    end if
    for c = 1 to m
        if is_safe (V, c)
            colors [V] = c
            if m_coloring (graph, m, colors, V+1, c)
                return true
            end if
            colors [V] = 0
        end if
    end for
    return false
end function
```

The pseudo code defines a recursive function 'm-coloring' that takes as input a graph, the number of available colors 'm', an array 'colors', to store the assigned colors for each vertex, the current vertex 'v' being considered and the current color 'c'. The function returns true if it is possible to color the vertices of the graph such that no two adjacent vertices have the same color and false otherwise.

The problem is solved by calling the function recursively which is the backtracking algorithm.

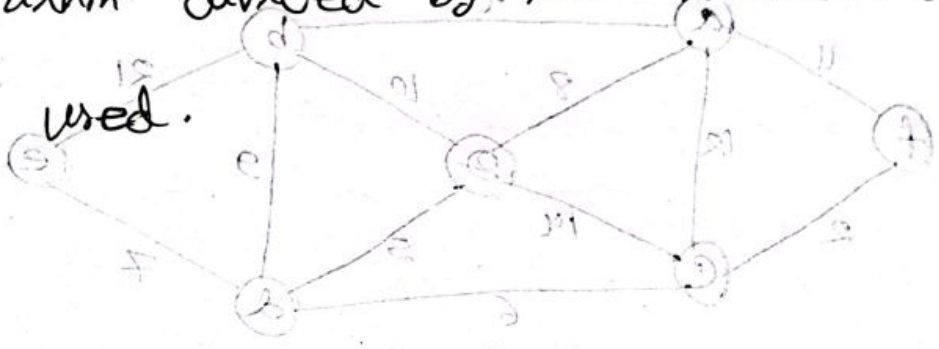
Q1 Define parallel algorithms. Explain Speedup, and Efficiency of parallel algorithms.

Ans:

Parallel algorithms are algorithms that are designed to be executed concurrently on multiple processors or computing devices. The goal of parallel algorithms is to speed up the execution of the algorithm by dividing the workload among multiple processors or devices, thereby reducing the overall runtime.

Speedup is a measure of how much faster a parallel algorithm is compared to its sequential counterpart. It is defined as the ratio of the runtime of the sequential algorithm to the runtime of the parallel algorithm.

Efficiency is a measure of how well the parallel algorithm uses the available processing resources. It is defined as the speedup of the algorithm divided by the number of processors used.

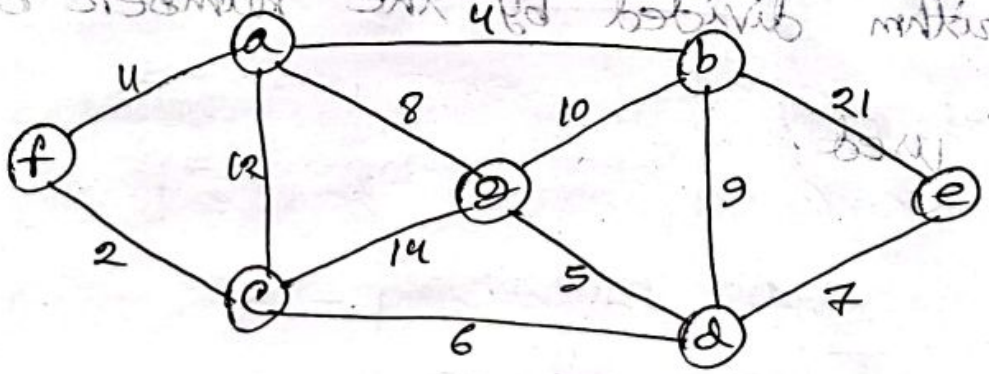


Minimum spanning tree (MST):

A spanning tree of a graph is just a sub-graph that contains all the vertices and is a tree. A graph may have many spanning trees.

The minimum spanning tree for a given graph is the spanning tree of minimum cost for that graph.

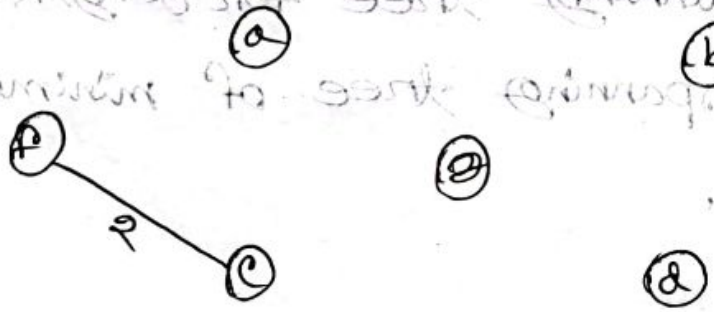
Using Prim's algorithm, the minimum cost spanning tree of the given graph—



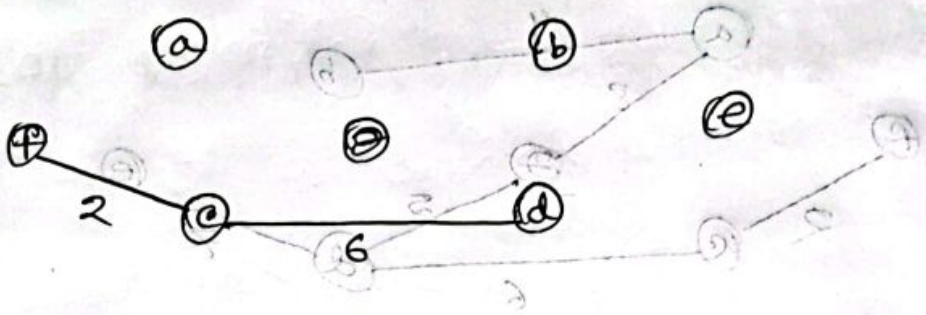
In Prim's algorithm, initially select the smallest weight edge and then always select the connected smallest one. It always maintain a tree.

Now, for this graph,

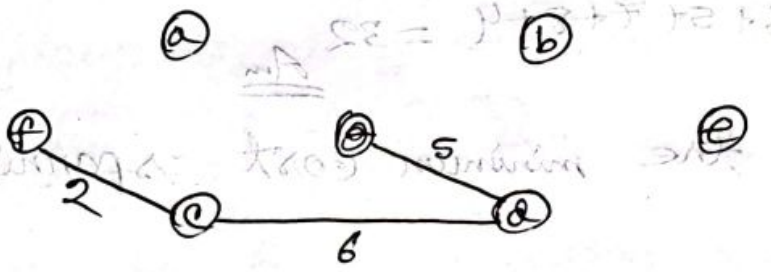
Step-1



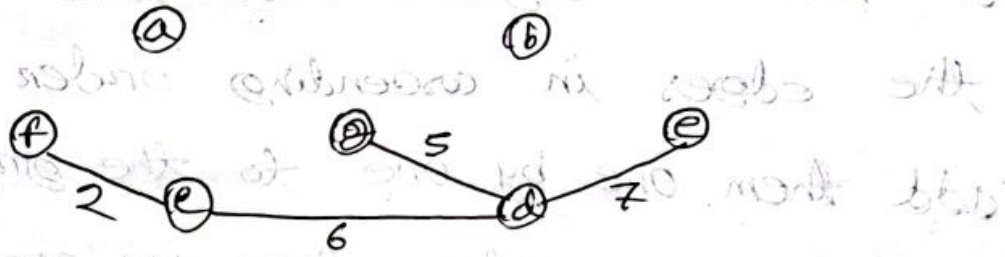
step-2



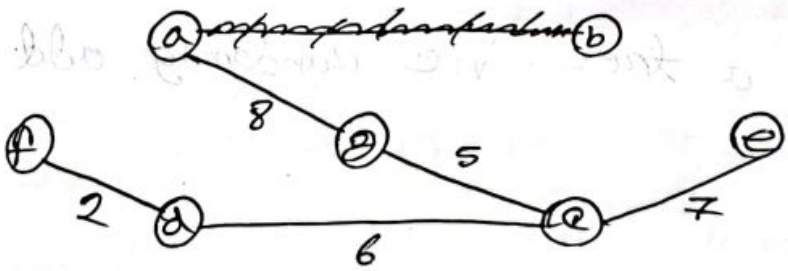
Step-3



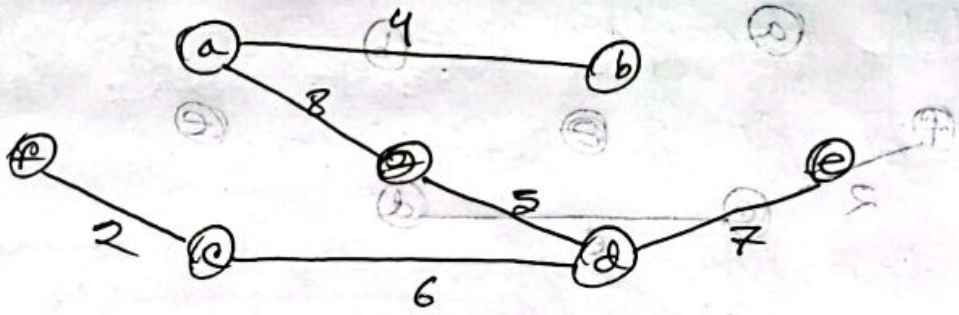
Step -4



Step -5



Step-6



So, the minimum cost is,

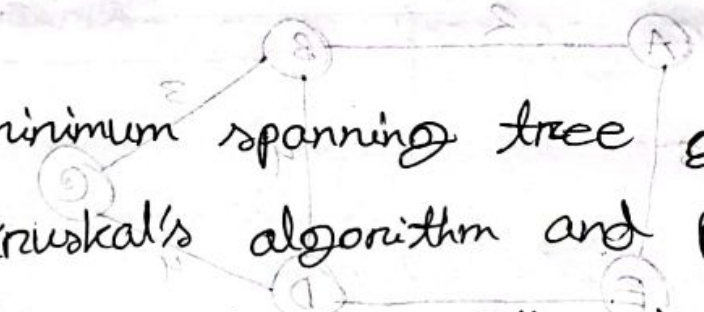
$2+6+5+7+8+4 = 32$ Ans

So, this is the minimum cost spanning tree.

In kruskal's algorithm, we sort all the edges in ascending order then add them one by one to the graph. if adding any edge form any cycle then skip that edge. Here, we don't have to maintain a tree. We randomly add edges.

Is the minimum spanning tree generated by both Kruskal's and Prim's unique?

Ans:



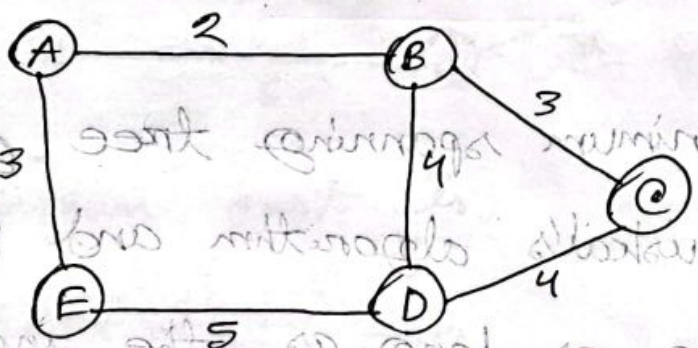
The minimum spanning tree generated using both Kruskal's algorithm and Prim's algorithm is unique as long as the input graph is connected and all the edge weights are unique. If these conditions are met, both algorithms will produce the same minimum spanning tree.

But if there are edges with the same value, some arbitrary decisions must be taken and the exact MST being the output of the algorithm cannot be predicted. There will be differences between different implementations of the same algorithm. Although, they all will have the same total

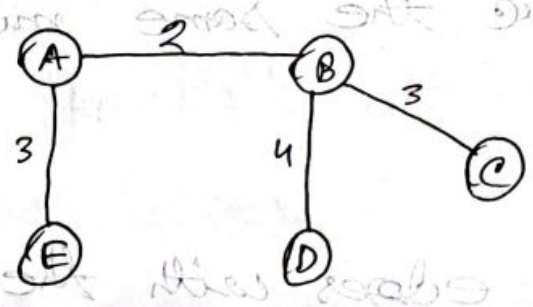
minimum cost. For example,

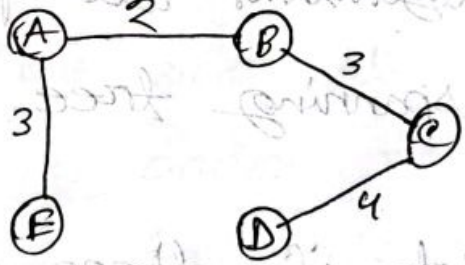
Let,

the given graph,

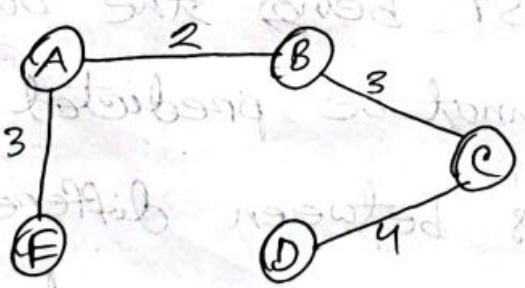


Using Kruskal's algorithm, we can have two different minimum spanning trees-





and using Prim's, the output will be,



So, always the generated minimum spanning tree will not be unique.

Differences between Kruskal's and Prim's algorithm :-

Prim's	Kruskal's
1. It starts to build the MST from any vertex in the graph.	1. It starts to build the MST from the vertex carrying minimum weight in the graph.
2. It traverses one node more than one time to get the minimum distance.	2. It traverses one node only once.
3. It is a vertex-based approach.	3. It is an edge-based approach.
4. It has a time complexity of $O(V^2)$, which can be improved upto $O(E \log V)$ using fibonacci heaps.	4. Its time complexity is $O(E \log V)$, V being the number of vertices.
5. It works only on connected graph.	5. It can work on disconnected graph also.
6. It runs faster in dense graph.	6. It runs faster in sparse graph.

How to analyze an algorithm with an example.

Ans:

There are several ways to analyze an algorithm -

1. Time complexity analysis
2. Space complexity analysis.
3. Experimental analysis.
4. Empirical analysis.
5. Asymptotic analysis.

For example,

If we want to find the maximum value in an array of integers, the algorithm is -

1. set max to the first element of the array
2. Loop through the remaining elements of the array.
3. For each elements, if it is greater than max, set max to that element.

4. Return max.

To analyze the time complexity of this algorithm, the algorithm has a single for loop that iterates through the entire array.

So, the time complexity $O(n)$.

Hence, we can analyze an algorithm.

What are Implicit and Explicit constraints?

Ans:

Implicit constraints refer to limitations or rules that are not explicitly stated within the algorithm, but are still present and affect the outcome of the algorithm. For example, the time complexity of an algorithm is an implicit constraints.

Explicit constraints are limitations or rules that are clearly stated within the algorithm. For example, a specific input range or output format can be explicit constraints in an algorithm.

So,

implicit constraints in algorithm are defined by the algorithm itself or its properties, while explicit constraints are defined by the problem or task and are clearly stated in the algorithm.

Implicit and Explicit constraints for n-queens and sum of subset problems.

Ans:

N-queens problem –

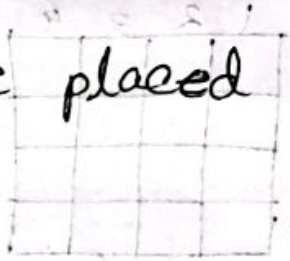
Implicit constraints:

① Only one queen can be placed on a single row and column.

(ii) Queens cannot be placed on the same diagonal.

Explicit constraints:

(i) The number of queens to be placed on the board (n).



Sum of subset problem –

Implicit constraints:

(i) The subsets must be mutually exclusive

(ii) The subsets must be collectively exhaustive

(iii) The integers in the set can only be used once.

Explicit constraints:

(i) The set of integers from which the subsets are to be chosen.

(ii) The desired sum that the subsets should add up to.

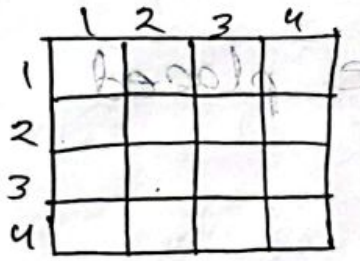
(iii) The size of the subsets. (if specified)

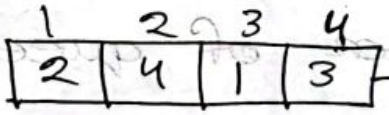
(iv) The number of subsets

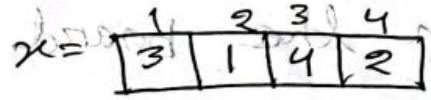
Recursive backtracking or State-space tree for 4-queen problem :-

Here,

x = Queen , 4-columns







Here, is the state-space tree for 4-queens problem. We find 2 solutions.

The solution to the problem is reduced

from $O(4^4)$ to optimal solution $O(4!)$ by using the backtracking algorithm, which is a type of DFS algorithm.

In the case of the n-queen problem, the algorithm starts by placing a queen in the first column and first row, and then moving on to the next column. It backtracks and tries the new solutions. The algorithm backtracks whenever a dead end is encountered, it doesn't waste time exploring solutions that are not valid. As a result, the time complexity of the algorithm is reduced from $O(n^n)$ to $O(n!)$ which is optimal.

Apply backtracking technique to solve the following instance of subset sum problem:

Given,

$S = \{1, 3, 4, 5, 8\}$

$d = 16$

Here,

x = element number

1 = taken

0 = not taken

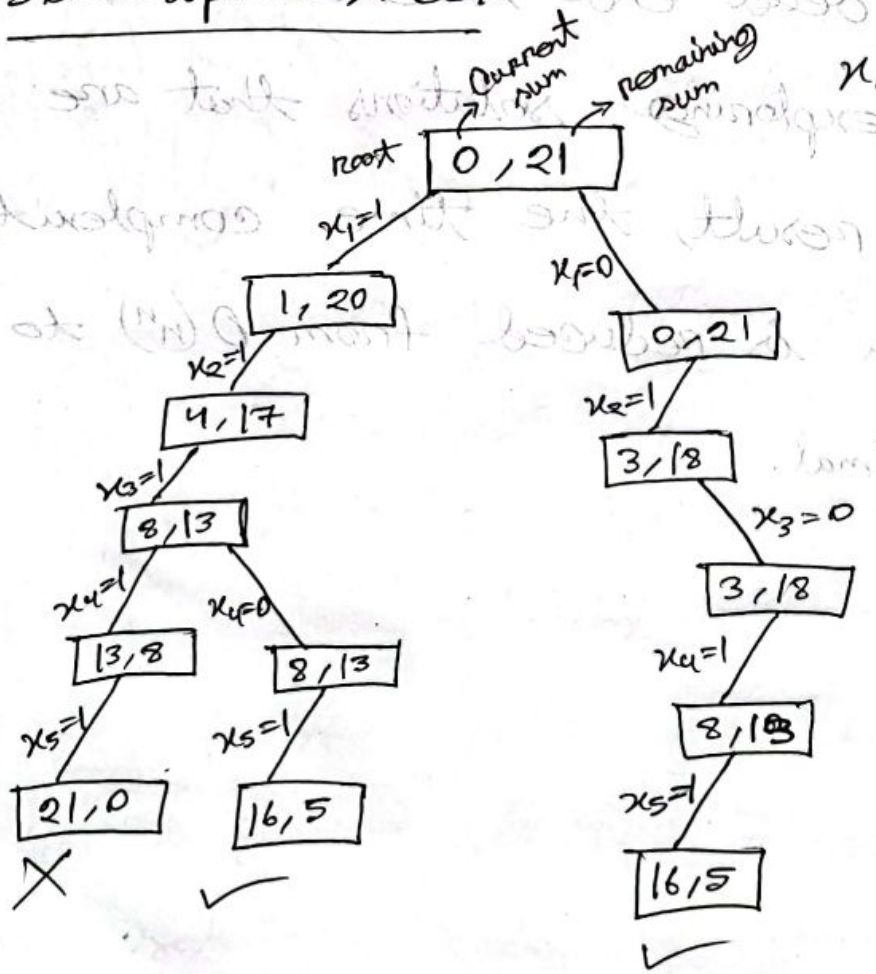
$x_i =$

1	2	3	4	5
1	1	1	0	1

$x_i =$

1	2	3	4	5
0	1	0	1	1

State-space tree:



Use example to distinguish between feasible solution and optimal solution for the case of knapsack problem.

Ans:

The knapsack problem is an optimization problem where given a set of items, each with weight and a value, and a knapsack with a limited weight capacity. The goal is to select a subset of items that maximizes the total value of the items while not exceeding the weight capacity.

A feasible solution for the knapsack problem is a solution that meets the constraint of the problem.

for example, a set of item,

Items weight : {2, 2, 3, 4, 5}

" value : {6, 10, 12, 15, 18}

and capacity = 10

A feasible solution would be,

item 1, 2, 3 with total weight = 7,

and total value = 28

An optimal solution is a feasible solution that is the best among all possible solutions.

In this case, the feasible solution above is not optimal, there is another solution that is better:

item: 2, 3 and 4 with total weight = 9

total value = 37

So, the optimal solution is a subset of feasible solutions, it's a feasible solution but it's also the best among all feasible solutions.

⊞ Considering a 0/1 knapsack problem:-

In a greedy algorithm, we always tend to pick an object that will make our profit more than the other objects. As, this is a 0/1 knapsack problem, we can not take any fraction of an object. So in greedy method, at first we sort all the objects in respect to their profit in descending order. Then we pick the objects from the very front until the total weight of an knapsack becomes equal or less than capacity.

In greedy method we may find a feasible solution, but that will not be an optimal solution.

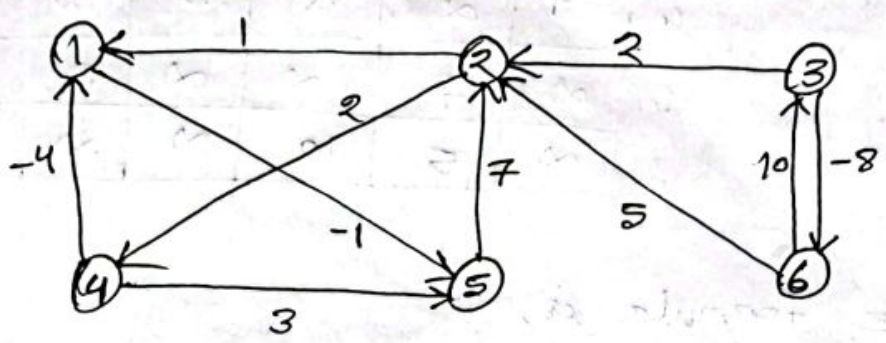
In dynamic programming, we either take an object or don't take. As we solve the same sub problems again and again, we can store

the solution to the subproblems in a 2D array dp[n][w]. where n is the number of elements and w is the maximum capacity. The state DP[i][j] will denote maximum value of 'j-weight' considering all values from '1 to ith'.

In greedy method, we have to sort the objects. So, the time complexity will be O(nlogn), where n is the total objects.

In dp, we use recursion, so, time complexity O(n·w) where n is the number of objects and w is the capacity of the knapsack.

Floyd-Warshall algorithm – all pair Matrix D^k is given below:-



The corresponding matrix:-

$A^0 =$

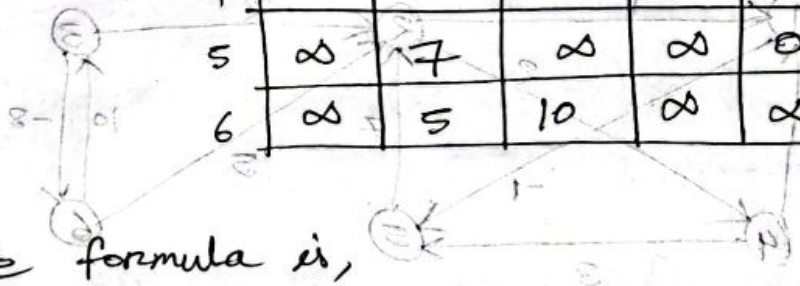
Node	1	2	3	4	5	6
1	0	∞	∞	∞	-1	∞
2	1	0	∞	2	∞	∞
3	∞	2	0	∞	∞	-8
4	-4	∞	∞	0	3	∞
5	∞	7	∞	∞	0	∞
6	∞	5	10	∞	∞	0

Outer loop will iterate for 6 times.

Now,

A^1 =

	1	2	3	4	5	6
1	0	∞	∞	∞	-1	∞
2	1	0	∞	2	0	∞
3	∞	2	0	∞	∞	-8
4	-4	∞	∞	0	-5	∞
5	∞	-7	∞	∞	0	∞
6	∞	5	10	∞	∞	0



The formula is,

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

Similarly,

A^6 =

	1	2	3	4	5	6
1	0	6	∞	8	-1	∞
2	-2	0	∞	2	-3	∞
3	-5	-6	0	-1	-6	-8
4	-4	2	∞	0	-5	∞
5	5	7	∞	9	0	∞
6	3	2	10	7	2	0

Ans

2018/1(b)

Let,

int = 2 bytes in memory

time = m sec. = ms

Now,

The step counts of the above program w.r.to space and time,

Here,

Total space required is,

(9×2) + (9×2) = 36 bytes [for two matrix]

2+2+2 = 6 bytes [for 3 variables]

9×2 = 18 bytes [for another matrix dynamic]

∴ total space = (36+6+18)

= 60 bytes Ans

For time,

1+1+1+3+(3×3)+(3×3)+1 = 25 ms Ans

2018/2@

Given,

$$T(n) = \begin{cases} a & , n=1 \\ 2T\left(\frac{n}{2}\right)+cn & , n>1 \end{cases}$$

Now,

$$T(n) = 2T\left(\frac{n}{2}\right)+cn$$

$$= 2\left[2T\left(\frac{n}{2^2}\right)+\frac{cn}{2}\right]+cn$$

$$= 2^2 T\left(\frac{n}{2^2}\right)+cn+cn$$

$$\therefore T(n) = 2^2 T\left(\frac{n}{2^2}\right)+2cn$$

$$T(n) = 2^2\left[2T\left(\frac{n}{2^3}\right)+\frac{cn}{2^2}\right]+2cn$$

$$T(n) = 2^3 T\left(\frac{n}{2^3}\right)+3cn$$

Similarly,

$$T(n) = 2^k T\left(\frac{n}{2^k}\right)+kcn$$

Assume,

$$T\left(\frac{n}{2^k}\right) = T(1)$$

$$\Rightarrow \frac{n}{2^k} = a$$

$$\Rightarrow n = 2^k \cdot a$$

$$\therefore \log n = ka$$

$$\therefore k = \frac{\log n}{a}$$

So,

$$T(n) = \frac{n}{a} \cdot T(1) + \frac{\log n}{a} \cdot cn$$

$$T(n) = \frac{n}{a} \cdot a + \frac{1}{a} cn \log n$$

$$T(n) = n + \frac{c}{a} n \log n$$

$$\therefore O(n \log n)$$

Ans

What is the principle of optimality in Dynamic programming?

Ans:

A problem is said to satisfy the principle of optimality if the subsolutions of an optimal solutions of the problem are themeselves optimal solutions for their sub problems.

The shortest path problem satisfies the principle of optimality.

2017/3@

Given,

Capacity (m) = 10

weight = {10, 3, 5}

profit = {40, 20, 30}

Using Dynamic programming,

	0	1	2	3	4	5	6	7	8	9	10
(40, 10)	0	0	0	0	0	0	0	0	0	0	40
(20, 3)	0	0	0	20	20	20	20	20	20	20	40
(30, 5)	0	0	0	20	20	30	30	30	50	50	50

So, the answer is 50 which is optimal.

If we use greedy algorithm, then we don't find the optimal solution. We may find any feasible solution.

So, we find the result by dynamic programming.

Algorithm for solving fractional knapsack problem using greedy method:-

Ans:

```
function greedy_knapsack (P, W, M, X, n)
// P(1:n) and w(1:n) contain the profits and
// weights respectively.
// Objects ordered so that P(i)/w(i) ≥ P(j+1)/w(i+1)
// M is the knapsack size and X(1:n) is the
// solution vector

P(1:n), w(1:n), X(1:n), M, cur;
integer i, n;
X ← 0    // Initialize solution to zero
cur ← M  // cur = Remaining knapsack capacity.
for i ← 0 to n do
        if w(i) > cur then exit
        endif
        X(i) ← 1
        cur ← cur - W(i)
```

repeat

if $i \leq n$ then $x(i) \leftarrow cur : w(i)$

end if

end greedy_knapsack.

Given,

	Total steps
for $x = 1$ to n do	$n+1$
begin	
$y = x$	n
while $(y > 1)$ do	$n*(\log n+1)$
$y = y/2$	$n*\log n$
end	

$\therefore$ Total $= 3n + 2n\log n + 1$

$\therefore$ Here, $f(n) = 3n + 2n\log n + 1$

So, we can say that,

$$n\log n \leq 3n + 2n\log n + 1 \leq 3n\log n + 2n\log n + n\log n$$

$$\Rightarrow n\log n \leq 3n + 2n\log n + 1 \leq 6n\log n$$

$\therefore f(n) = \theta(n\log n)$ Ans

Prove that,

$$n \log n = O(n^{3/2})$$

$\therefore \quad f(n) = n \log n$

$\therefore \; n \log n \leq n \sqrt{n}$

$\Rightarrow n \log n \leq n^{3/2}$

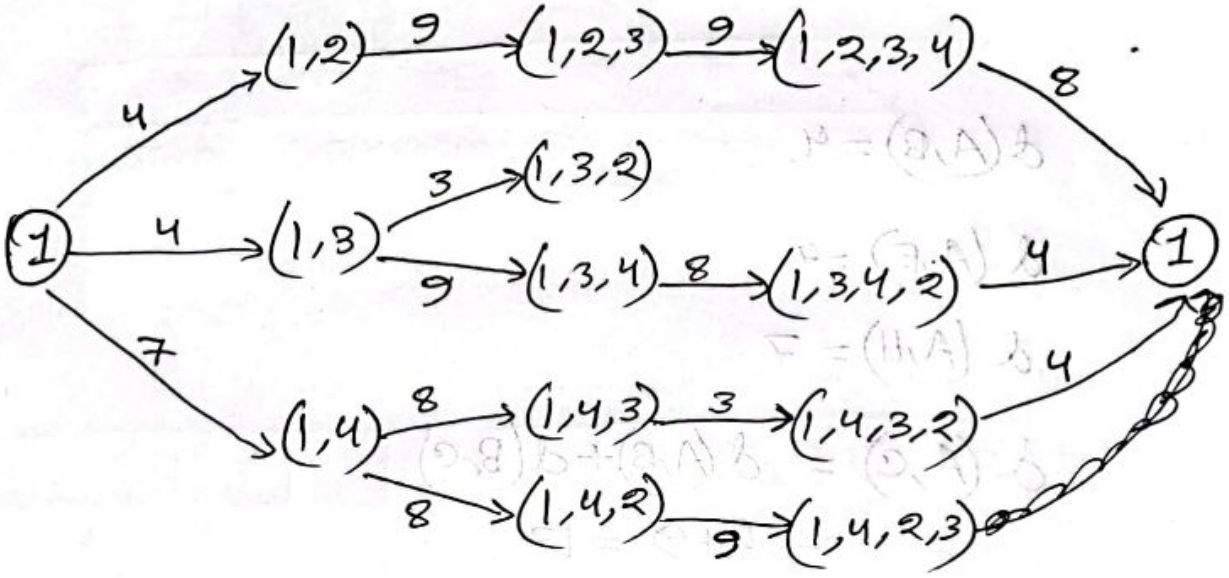
$\therefore f(n) \leq c \cdot g(n)$

$\therefore n \log n = O(n^{3/2})$

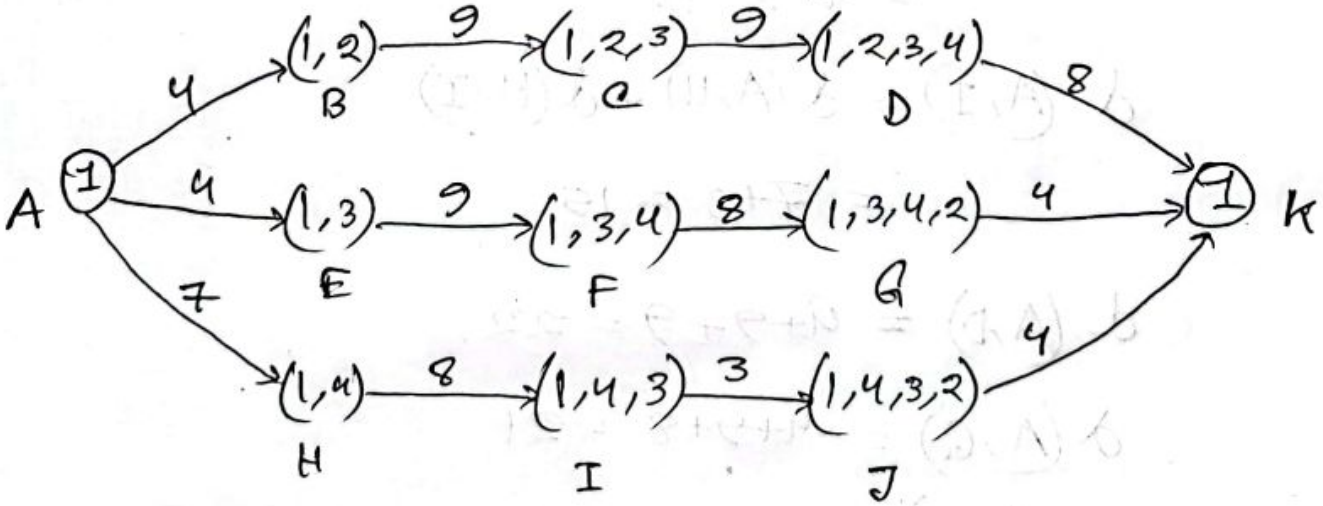
(Proved)

2020/2@

(i) Here, the Multi-stage Graph is given below;-



So, the multi-stage graph is,



(ii) Now,

From above graph,

Using the backward approach,

$d(A,B) = 4$

$d(A,E) = 4$

$d(A,H) = 7$

$d(A,C) = d(A,B) + d(B,C)$

$= 4+9 = 13$

$d(A,F) = d(A,E) + d(E,F)$

$= 4+9 = 13$

$d(A,I) = d(A,H) + d(H,I)$

$= 7+8 = 15$

$d(A,D) = 4+9+9 = 22$

$d(A,G) = 4+9+8 = 21$

$d(A,J) = 7+8+3 = 18$

$\therefore d(A,K) = \min [d(A,D)+d(D,K), d(A,G)+d(G,K)+d(A,J)+d(J,K)]$

$= \min (22+8, 21+4, 18+4)$

$= \min (30, 25, 22) = 22$

∴ The minimum cost path is,

$A \to H \to I \to J \to K$

$1 \to 4 \to 3 \to 2 \to 1$

For forward approach,

$$d(A,K) = \min[4+d(B,K), 4+d(E,K), 7+d(H,K)]$$

$d(B,K) = 9+d(C,K) = 9+17 = 26$

$d(E,K) = 9+d(F,K) = 9+12 = 21$

$d(H,K) = 8+d(I,K) = 7+8 = 15$

$d(C,K) = 9+d(D,K) = 9+8 = 17$

$d(F,K) = 8+d(G,K) = 8+4 = 12$

$d(J,K) = 3+d(J,K) = 3+4 = 7$

$\therefore d(A,K) = \min(4+26, 4+21, 07+15)$

$= \min(30, 25, 22)$

$= 22$ Ans

2019/4ⓑ

Given,

i	1	2	3	4	5
P_i	0.24	0.22	0.23	0.3	0.01

Now,

	0	1	2	3	4	5
0	0	0.24	0.68			
1		0	0.22	0.67		
2			0	0.23	0.76	
3				0	0.3	0.32
4					0	0.01
5						0

Cost matrix

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

Root matrix

$$c[0,2] = \min\left[c[0,0]+c[1,2],\ c[0,1]+c[2,2]\right]+w[0,2]$$

$$= 0.68$$

Similarly,

—

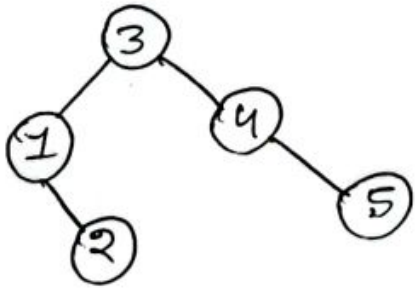
—

$$c[1,3] = \min\{c[1,1]+c[2,3],\ c[1,2]+c[3,3]\} + w[1,3]$$

$$= 0.67$$

$$c[0,3] = \min\{c[0,0]+c[1,3],\ c[0,1]+c[2,3],\ c[0,2]+c[3,3]\} + w[0,3]$$

$$=$$



Ans