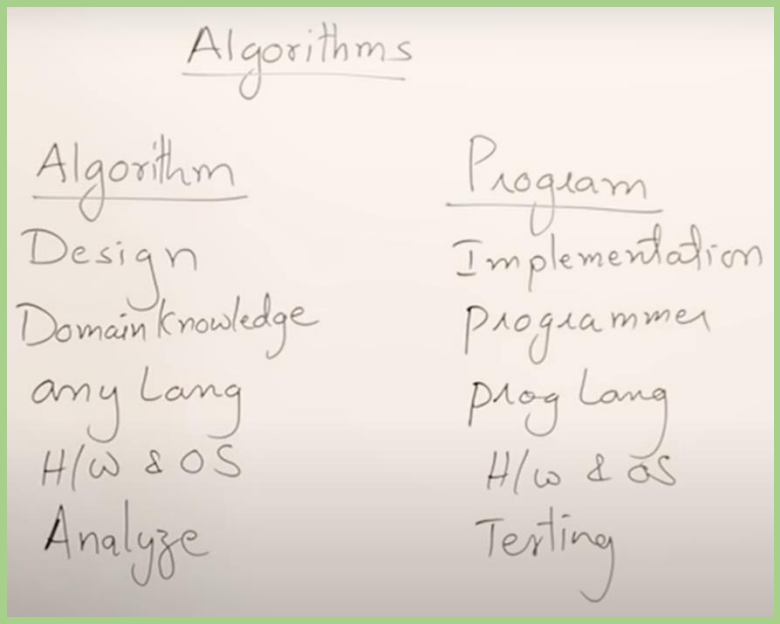
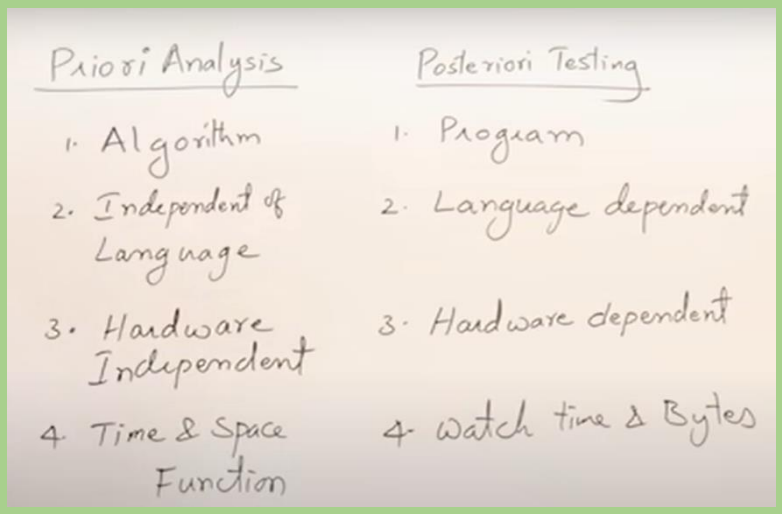
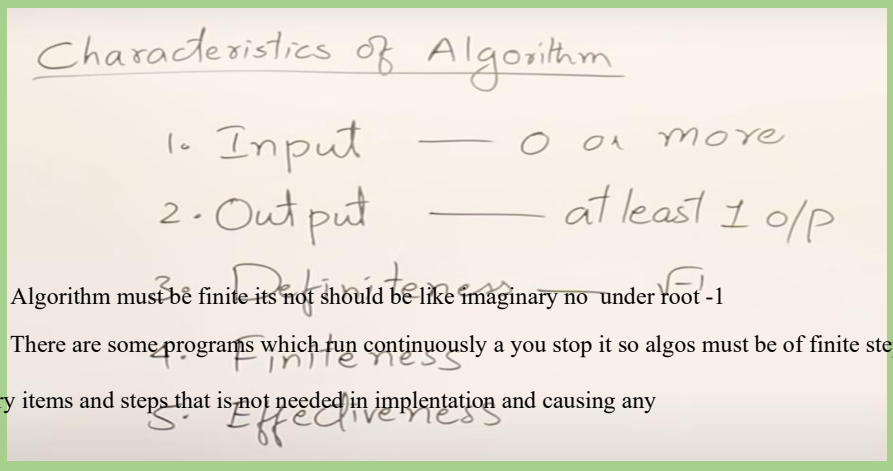
DAA







Algorithm must be finite its not should be like imaginary no under root -1

There are some programs which run continuously a you stop it so algos must be of finite steps

ry items and steps that is not needed in implentation and causing any

Analyse a program

- **Time Complexity:** Time complexity measures how many operations an algorithm completes in relation to the size of the input.
- Operations can be comparisons, assignments, arithmetic operations, or any other fundamental computation that the algorithm carries out.
- Takes a finite amount of time for execution. The time required by the algorithm to solve given problem is called time complexity
- Time complexity is usually expressed using Big O notation, which provides an upper bound on the number of basic operations an algorithm will perform as a function of the input size. It describes how the running time of an algorithm scales with respect to the input size
- Steps for time complexity :
    - Basic operations: assisgnment ,compare,+,-…
    - Input size: like no of lements in an array
    - Asymptic analysis : how no of operation grows on increase of input size often uses big o
    - Big o : n,n2,n3,logn
    - Best,avg,worst:
    - Dominance : if O(n^2 + 3n + 7), we often consider the O(n^2)
    - Comparison of algos : we try to find best optimal algos from diff algos
- Maximum element
    - for (int i = 1; i < arr.length; i++) {
    - if (arr[i] > max) {
        - max = arr[i]; // Update max if a larger element is found
    - }
    - The findMax function iterates through the array once, comparing each element to the current maximum. So, it has a time complexity of O(n), where n is the length of the array.
- Swapping
    - Swap(a,b){
    - Temp=a;
    - A=b;
    - B=temp;
    - }
    - The time complexity of the swap function is O(1) because it involves a fixed number of operations, regardless of the values of a and b. It performs three assignments, which are considered constant-time operations

- **Space Complexity** : The amount of memory required by the algorithm to solve given problem or completion of an algorithm.
- It focuses on understanding how the space usage of an algorithm scales as the input size grows.
- Steps :
    - Memory usage: storage req , sometimes for temp variable or for variable and data structures
    - Input size:
    - Data structures used : many algos need additional ds like array LinkL space is required
    - Recursion and function call : it takes up the stack of calls
    - In-place algos: they have low space they don’t require exy=tra space
    - Asymptic analysis : diff types : O(1) O (n)
    - Worst case : mostly space complexity evaluate algos in worst case which takes extra space
    - Comparison :

- Addition
    - C <— A+B

    - return C
    - The addition of two scalar numbers requires one extra memory location to hold the result. Thus the space complexity of this algorithm is constant, hence S(n) = O(1).
- Reversing array
    - for (int i = 0; i < n; i++) {
    - reversed[i] = arr[n - 1 - i]; // Reverse the order of elements
    - }
    -
    - return reversed;
    - }
    - algorithm creates a new array reversed of the same size. As the input size increases, the space used for the new array also increases linearly. This is why the space complexity of this algorithm is O(n).

Correctness of algorithm

- Proving correctness of algorithm is crucial
- algorithms are very complex. The reliability of an algorithm cannot be claimed unless and until it gives the correct output for each of the valid inputs.
- Tracing the output of each possible input is impossible.
- The correctness of an algorithm can be quickly proven by checking certain condition

There are three ways to find the correctness of algorithm

Proof by Contradiction:

Idea: Assume that the algorithm is incorrect and show that this assumption leads to a contradiction.

Example: To prove the correctness of an algorithm, assume the opposite (i.e., the algorithm is incorrect), and then show that this assumption results in an inconsistency.

Proof by Mathematical Induction:

Idea: Prove that the algorithm works correctly for a base case and then show that if it works for an arbitrary case, it also works for the next case.

Example: Prove that an algorithm correctly computes a factorial by showing that it works for

n=1 (base case) and assuming it works for n=k to prove it works for n=k+1

Loop Invariant Proof:

Idea: Prove the correctness of a loop by defining an invariant (a property that holds before and after each iteration of the loop) and proving that it is maintained throughout the loop's execution.

Example: Prove the correctness of a sorting algorithm by showing that the array is always sorted after each iteration of the loop.

- Initialization: Conditions true before the first iteration of the loop
- Maintenance: If the condition is true before the loop, it must be true before the next iteration.
- Termination: On termination of the loop, the invariant gives us a useful property that helps us prove the correctness of the algorithm.

Randomized Algorithm:

- These algo use some randomness in their logic to improve efficiency time complexity or the total memory used
- Algo picks up a random number from some range and based on the value of random number it makes the decision
- Randomized algorithms are algorithms that use randomization as a key part of their design to solve computational problems.
- randomized algorithms incorporate randomness or probability in their decision-making process.
- use of randomization can lead to more efficient solutions, simpler designs, and better average-case performance.
- **Randomized Choices:**
  - Randomized algorithms make random choices during their execution.
  - These choices might involve selecting random elements from a data structure,
- **Probabilistic Analysis:**
  - The performance of a randomized algorithm is analyzed using probability theory.
- **Applications:**
  - Randomized algorithms find applications in various areas, including optimization, machine learning, cryptography, data streaming, parallel computing, and more.
- **Examples:**
  - **Quicksort:** The choice of the pivot in the quicksort algorithm is often made randomly. While the worst-case time complexity remains the same as the deterministic version, randomization can lead to better average-case performance.
- two broadly classified categories are **Monte Carlo** and **Las Vegas**
- **Monte Carlo Algorithms:**
  - These algorithms may produce an incorrect result with a small probability.
  - The focus is on the expected running time rather than correctness.
  - The probability **of error can be made arbitrarily small by repeating the algorithm.**
  - **Example**: Miller-Rabin Primality Test: It is a Monte Carlo algorithm used to probabilistically determine whether a number is composite or likely prime. It may have a small probability of error but can be made highly accurate by performing multiple tests.
- **Las Vegas Algorithms:**
  - These algorithms always produce the correct result.
  - The running time, however, may vary randomly.
  - The goal is to provide deterministic correctness while leveraging randomization for efficiency gains.
  - **Example:** Quicksort with Randomized Pivot: In the randomized version of quicksort, the pivot element is chosen randomly. While the worst-case time complexity remains the same as the deterministic version, randomization helps avoid worst-case scenarios and achieve better average-case performance.
- **Comparison:**
  - **Monte Carlo algorithms** trade correctness for efficiency, and their results are probabilistic. The focus is on achieving faster average-case performance.
  - **Las Vegas algorithms** prioritize correctness but allow for some randomness in the running time. They provide deterministic correctness while leveraging randomization for efficiency gains.

Unit -4

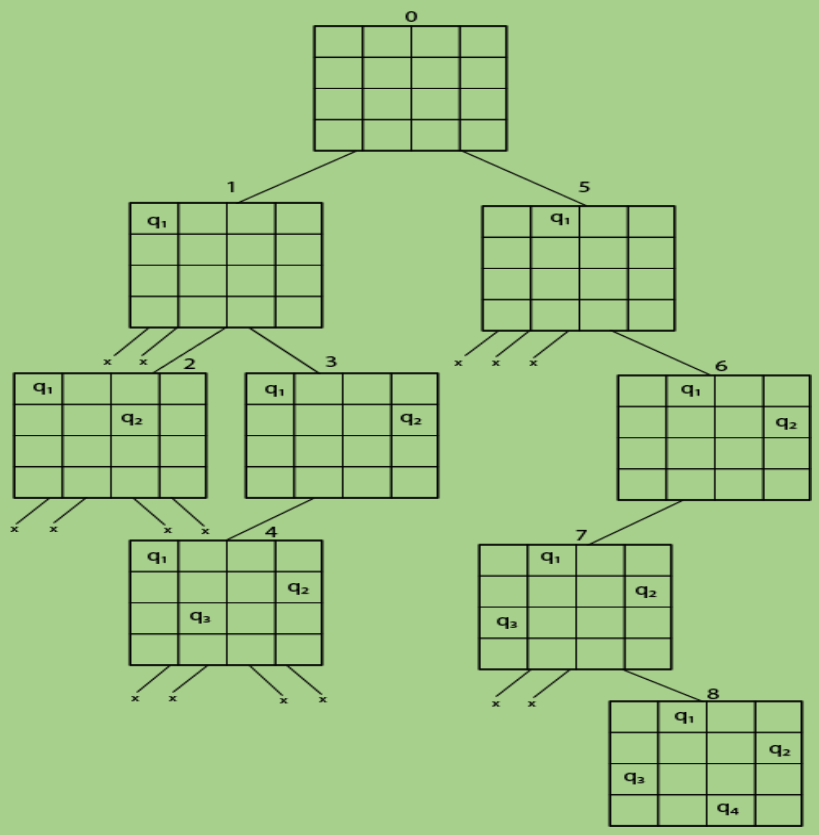
Backtracking

- When a path in an algorithm is not able to determine the solution so as soon as it determines that the path cannot possibly be completed to a valid solution. It abandons the solution(also called as backtrack) and move to some other path
- **Exhaustive search:** Backtracking explores all possible solutions in a systematic manner, ensuring that no potential solution is overlooked. It guarantees finding the optimal solution if one exists within the search space.
- **Efficiency**: Although backtracking involves exploring multiple paths, it prunes the search space by eliminating partial solutions that are unlikely to lead to the desired outcome. This pruning improves efficiency by reducing the number of unnecessary computations.
- **Solution uniqueness**: Backtracking can find multiple solutions if they exist. It can be modified to continue the search after finding the first solution to find additional valid solutions.
- **Flexibility**: Backtracking allows for flexibility in problem-solving by providing a framework that can be customized to various problem domains. It is not limited to specific types of problems and can be applied to a wide range of scenarios.
- **Memory efficiency**: Backtracking typically requires minimal memory usage compared to other search algorithms. It operates in a recursive manner, utilizing the call stack to keep track of the search path. This makes it suitable for solving problems with large solution spaces.

N-queens problem

- **N-Queens problem involves placing N queens on an N×N chessboard in such a way that no two queens threaten each othe**
- **Chessboard Representation:**
  - The chessboard is represented as an N×N grid.
  - Each cell of the grid can either be empty or occupied by a queen.
- **Queen Placement Rules:**
  - Queens can attack along rows, columns, and diagonals.
  - No two queens can share the same row, column, or diagonal.
- **Objective:**
  - The goal is to place N queens on the board in a way that satisfies the placement rules.
- **Backtracking Algorithm:**
  - The algorithm explores possible placements and backtracks when it reaches an invalid configuration.
- **Recursive Exploration:**
  - The backtracking algorithm explores possible placements recursively.
  - It tries different possibilities for each row, moving forward until a solution is found or it backtracks.
- **Solution Space:**
  - The solution space is explored systematically until a valid placement for all queens is found.
  - The algorithm avoids invalid configurations and efficiently finds solutions.
- **All Solutions:**
  - The N-Queens problem often has multiple solutions.

- The algorithm may find all possible arrangements that meet the criteria



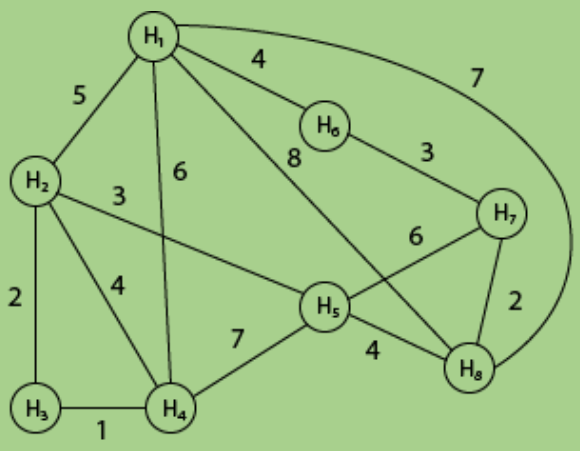
Travelling Sales Person Problem

The traveling salesman problems abide by a salesman and a set of cities. The salesman has to visit every one of the cities starting from a certain one (e.g., the hometown) and to return to the same city. The challenge of the problem is that the traveling salesman needs to minimize the total length of the trip.

Suppose the cities are x_1 x_2..... x_n where cost c_{ij} denotes the cost of travelling from city x_i to x_j. The travelling salesperson problem is to find a route starting and ending at x_1 that will take in all cities with the minimum cost.

Example: A newspaper agent daily drops the newspaper to the area assigned in such a manner that he has to cover all the houses in the respective area with minimum travel cost. Compute the minimum travel cost.

The area assigned to the agent where he has to drop the newspaper is shown in fig:



	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	4	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

Solution: The cost- adjacency matrix of graph G is as follows:

$cost_{ij}$ =

The tour starts from area H_1 and then select the minimum cost area reachable from H_1.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
(H_1)	0	5	0	6	0	(4)	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

Mark area H_6 because it is the minimum cost area reachable from H_1 and then select minimum cost area reachable from H_6.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
(H_1)	0	5	0	6	0	(4)	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
(H_6)	4	0	0	0	0	0	(3)	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

Mark area H_7 because it is the minimum cost area reachable from H_6 and then select minimum cost area reachable from H_7.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
(H1)	0	5	0	6	0	(4)	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
(H6)	4	0	0	0	0	0	(3)	0
(H7)	0	0	0	0	6	3	0	(2)
H_8	7	0	0	0	4	0	2	0

Mark area H_8 because it is the minimum cost area reachable from H_8.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
(H1)	0	5	0	6	0	(4)	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
(H6)	4	0	0	0	0	0	(3)	0
(H7)	0	0	0	0	6	3	0	(2)
(H8)	7	0	0	0	(4)	0	2	0

Mark area H_5 because it is the minimum cost area reachable from H_5.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
(H1)	0	5	0	6	0	(4)	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
(H5)	0	(3)	0	7	0	0	6	4
(H6)	4	0	0	0	0	0	(3)	0
(H7)	0	0	0	0	6	3	0	(2)
(H8)	7	0	0	0	(4)	0	2	0

Mark area H_2 because it is the minimum cost area reachable from H_2.

Mark area H_3 because it is the minimum cost area reachable from H_3.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	(4)	0	7
H_2	5	0	(2)	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	(3)	0	7	0	0	6	4
H_6	4	0	0	0	0	0	(3)	0
H_7	0	0	0	0	6	3	0	(2)
H_8	7	0	0	0	(4)	0	2	0

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	(4)	0	7
H_2	5	0	(2)	4	3	0	0	0
H_3	0	2	0	(1)	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	(3)	0	7	0	0	6	4
H_6	4	0	0	0	0	0	(3)	0
H_7	0	0	0	0	6	3	0	(2)
H_8	7	0	0	0	(4)	0	2	0

Mark area H_4 and then select the minimum cost area reachable from H_4 it is H_1.So, using the greedy strategy, we get the

following.

4 3 2 4 3 2 1 6

$H_1 \rightarrow H_6 \rightarrow H_7 \rightarrow H_8 \rightarrow H_5 \rightarrow H_2 \rightarrow H_3 \rightarrow H_4 \rightarrow {}_{H1}$.

Thus the minimum travel cost = 4 + 3 + 2 + 4 + 3 + 2 + 1 + 6 =

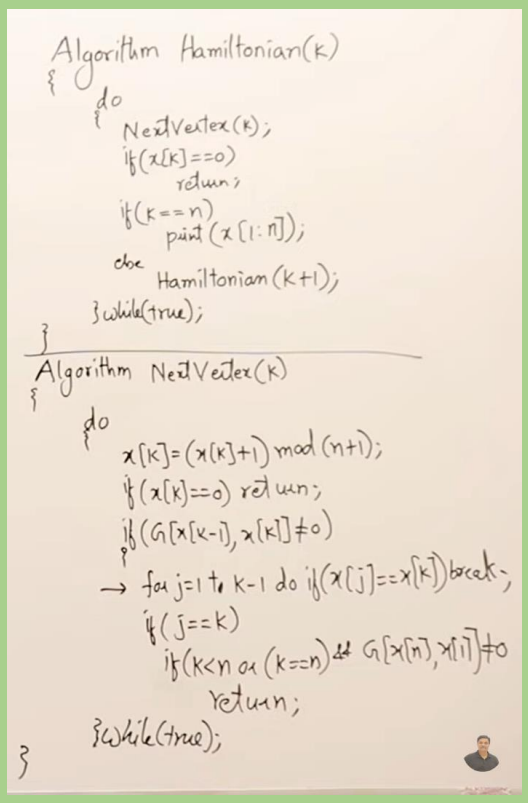
25

Hamilton circuit

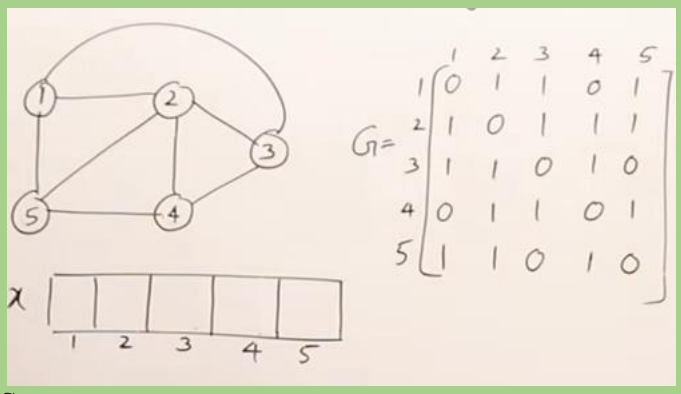
- Can ne Directed or undirected
- A Hamiltonian circuit is a cycle that passes through every vertex of a graph exactly once.
- The cycle is closed, meaning it starts and ends at the same vertex.
- **Hamiltonian Graph:**
  - A graph that contains a Hamiltonian circuit is called a Hamiltonian graph.
  - Not all graphs are Hamiltonian;
- **Hamiltonian Path:**
  - If a graph contains a path that visits each vertex exactly once (but doesn't necessarily form a closed loop), it's called a Hamiltonian path.
- **Necessary Conditions:**

- If a Hamiltonian circuit exists in a graph with N vertices, then every vertex must have a degree of at least N/2 (where N is even) or (N-1)/2 (where N is odd).
- This is a necessary condition but not a sufficient one, as there are graphs that meet this condition but don't have Hamiltonian circuits.

- **Traveling Salesman Problem:**
  - The Hamiltonian circuit problem is related to the Traveling Salesman Problem (TSP), where the goal is to find the shortest possible Hamiltonian circuit in a weighted graph.
- **NP-Completeness:**
  - Determining whether a Hamiltonian circuit exists in a graph is known to be an NP-complete problem, meaning that it is unlikely to have an efficient (polynomial-time) algorithm to solve it for all possible cases.
- **Examples:**
  - Hamiltonian circuits can be found in various contexts, including transportation planning, circuit design, and network optimization.
- **Algorithmic Approaches:**
  - There are heuristic approaches and algorithms for finding Hamiltonian circuits, such as backtracking algorithms, branch-and-bound, and genetic algorithms**.**
- **Algorithm**
-

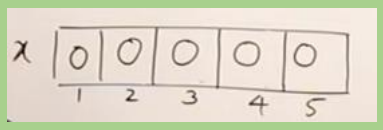


Given:

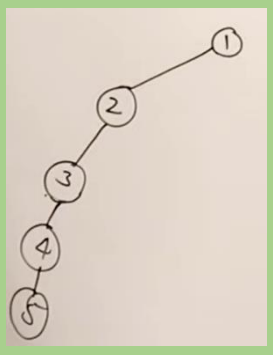


Steps :

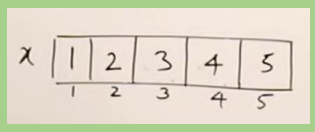
- Step 1:



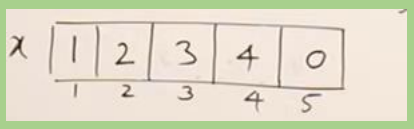
-



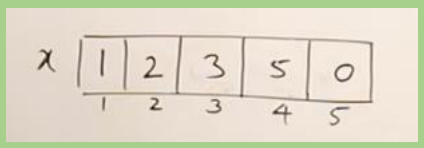
- Step 2: now keep moving and edges should not be repeated now check if there is a edge between 5 and 1 if yes then there exist a cycle and print result

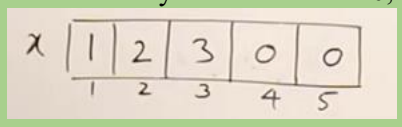


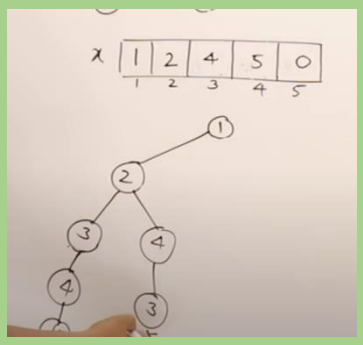
- Step 3: Continue the algo consider last as 0 and backtrack to find more solutions



- Step 4: Now move back and check at 4th place can we place 5 for placing 5 threre must be edge between 3 and 5 no so not possible now 4 already taken so its like 0,1,2,3,4,5 now 0 will be taken

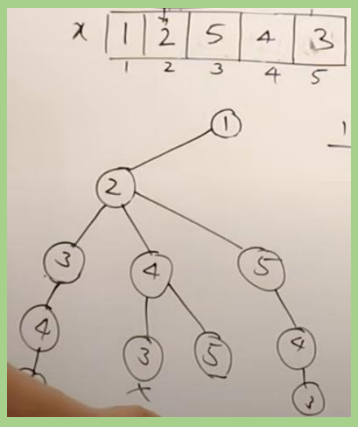






- Step 5 Now at place of 3 check any other which have par with 2 so take 4 now move forward take 5 as its not there in list now only option is 3 but there is no edghe bet9ween 3 and 5 put 0 and move backward again in this way find multiple sol

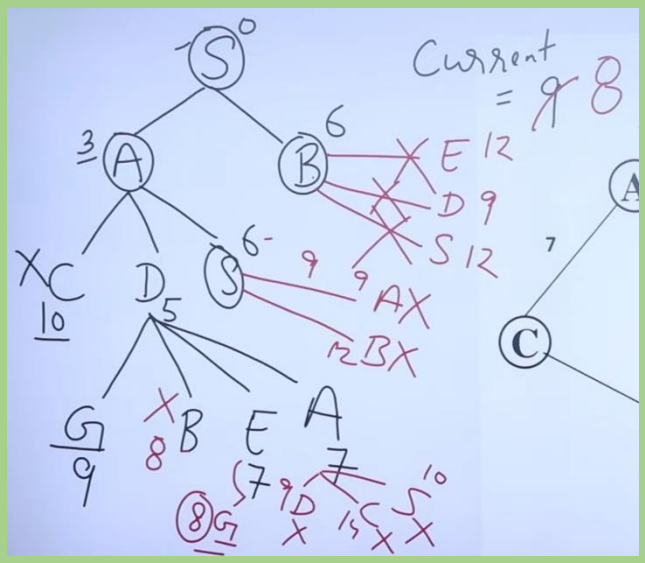
- Final ans is

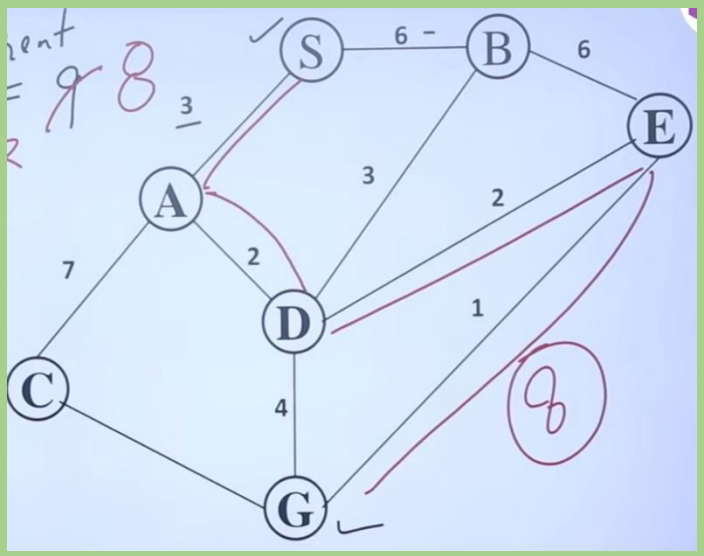


Time complexity is (n-1)!

Branch & Bound Algorithm

- Branching is the process of generating subproblems.
- Bounding refers to ignoring partial solutions that cannot be better than the current best solution.
- It combines the concepts of divide-and-conquer and intelligent search to systematically search for the best solution while avoiding unnecessary computations.
- The key idea behind Branch and Bound is to prune or discard certain branches of the search tree based on bounding information.
- It is a search procedure to find the optimal solution.
- It eliminates those parts of a search space which does not contain better solution(Pruning).
- In this method we basically extend the cheapest partial path.
- The branch and bound algorithm can be used to solve a wide variety of optimization problems, including:
    - The knapsack problem
    - The traveling salesman problem
    - The scheduling problem
    - Graph coloring
- Example: jab hm G tak first time phuch gye by S->A->D->G hme 9 solution ilgya to ab hm isse greater value ko prune krdenege jase ki C ko ab hm rukenge nhi ab next solution dhudhenge ab hm dekheneg sare leaf nodes me se smallest kon hai B usko dekha sab 9 se bade the prune krdiya aese he ek ek krke sabko check krebe to end me hme 8 bhi mila best solution





B&B algorithm operates according to two principles:

- Branching: The algorithm recursively branches the search space into smaller and smaller subproblems. Each subproblem is a subset of the original problem that satisfies some constraints.
- Bounding: The algorithm maintains a set of upper and lower bounds on the objective function value for each subproblem. A subproblem is eliminated from the search if its upper bound is greater than or equal to its lower bound.

Algo
Start

- Starts with exploring the solution space by dividing it into smaller subproblems or branches. Each branch represents a potential solution path.
- The algorithm estimate its potential for improvement. This estimation is often based on a lower and upper bound.

Lower bound

- lower bound provides a minimum value that the objective function can have.
- It helps in determining whether a branch can potentially lead to a better solution than the best one found so far.
- If the lower bound of a branch is worse than the best solution found that branch can be pruned, as it cannot contribute to the optimal solution

Upper bound

- upper bound, provides best possible value
- It helps in identifying branches that can potentially lead to an optimal solution.
- If the upper bound of a branch is worse than the best solution found, it implies that the branch cannot contain the optimal solution, and thus it can be discarded.

Branching

- Involves dividing the current branch into multiple subbranches by making a decision at a particular point.
- Each subbranch represents a different choice or option for that decision.
- The algorithm explores these subbranches in a systematic manner, typically using depth-first or breadth-first search strategies.

Reducibility,

- It refers to the ability to transform one computational problem into another in a way that preserves the solution.
- The notion of reducibility is crucial for classifying problems based on their computational difficulty and relationships with each other.
- There are different types of reducibility, including polynomial-time reducibility and many-one reducibility.
    - **Polynomial-Time Reducibility (Cook Reducibility):**
        - **Definition**: A problem A is polynomial-time reducible to problem B if there exists a polynomial-time computable function f such that for every instance x of problem A the value F(x) s an instance of problem B, and the solution to A can be efficiently obtained from the solution to B.
        - **Notation:** $A <=_p B$ read as "A is polynomial-time reducible to B

    - **Many-One Reducibility:**
        - **Definition**: A problem A is many-one reducible to problem B if there exists a computable function f (not necessarily polynomial-time) such that for every instance x of problem A, the value f(x) is an instance of problem B, and the solution to A can be efficiently obtained from the solution to B.
        - **Notation:** $A <=_m B$, read as "A is many-one reducible to B.

- The concept of reducibility is fundamental for several reasons:
- **Classification of Problems**: Reducibility helps classify problems into complexity classes based on their computational relationships.
- **Hardness and Completeness**: It allows the identification of problems that are at least as hard as others. If problem A is reducible to problem B, and B is hard, then A is also hard.
- **NP-Completeness**: The notion of reducibility plays a central role in defining NP-completeness. A problem is NP-complete if it is in NP, and every problem in NP is polynomial-time reducible to it.
- **Understanding Computational Relationships**: Reducibility provides a way to understand the computational relationships between problems and helps establish their relative difficulty.
- **For example**, when proving the NP-completeness of a problem, one typically starts by showing that a known NP-complete problem can be polynomial-time reduced to the problem at hand. This establishes the new problem's membership in the NP-complete class.
-