

# Analysis of Algorithm

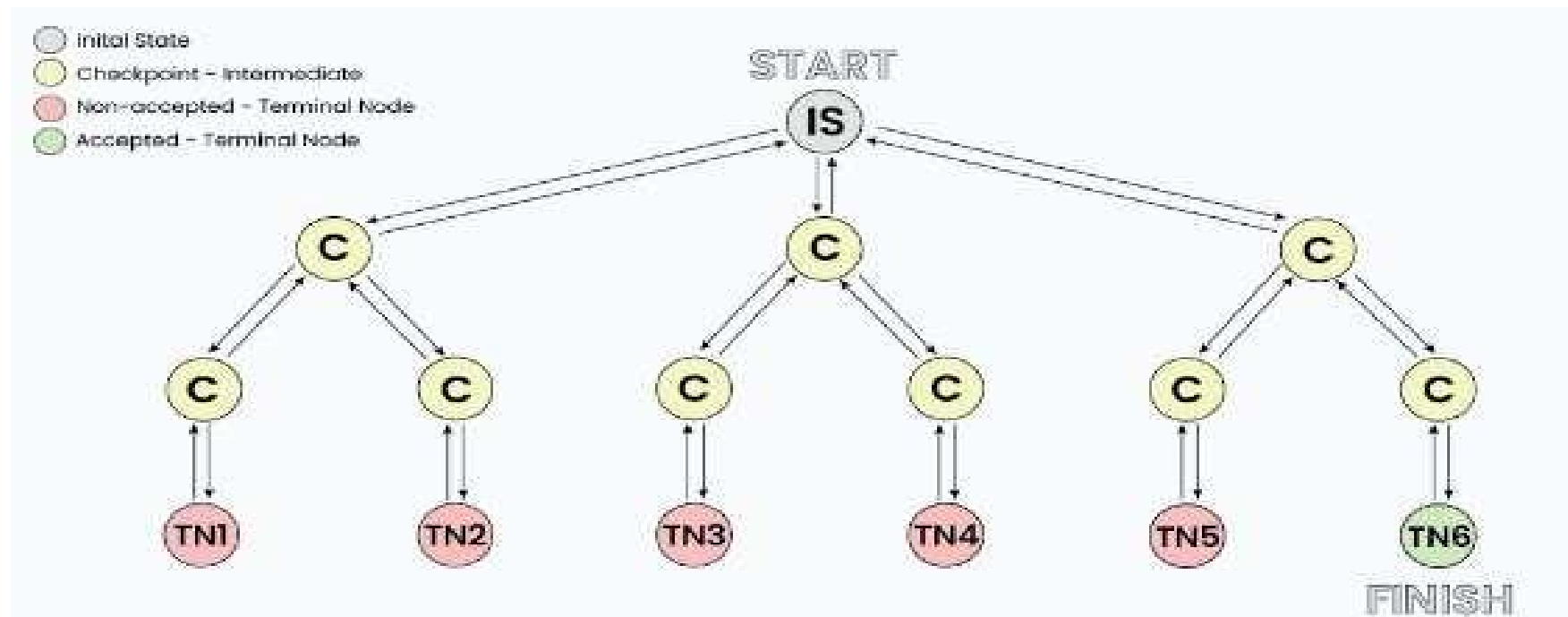
Amypo Technologies Pvt Ltd

## Concepts:

- Backtracking
- Rate in a Maze
- Permutation and Combination
- N Queen Problem
- Subset Sum

# Backtracking

Backtracking is an algorithmic technique where the goal is to get all solutions to a problem using the brute force approach. It consists of building a set of all the solutions incrementally. Since a problem would have constraints, the solutions that fail to satisfy them will be removed.



## Rate in a Maze

Consider a rat placed at  $(0, 0)$  in a square matrix of order  $N * N$ . It has to reach the destination at  $(N - 1, N - 1)$ . Find all possible paths that the rat can take to reach from source to destination. The directions in which the rat can move are 'U'(up), 'D'(down), 'L' (left), 'R' (right). Value 0 at a cell in the matrix represents that it is blocked and rat cannot move to it while value 1 at a cell in the matrix represents that rat can be travel through it. Return the list of paths in lexicographically increasing order.

**Note:** In a path, no cell can be visited more than one time. If the source cell is 0, the rat cannot move to any other cell.

# Example

Input:

Source			
			Dest.

Output: DRDDRR

Explanation:

Source			
			Dest.

## Step-by-step approach:



- Create isValid() function to check if a cell at position (row, col) is inside the maze and unblocked.
- Create findPath() to get all valid paths:
  - Base case: If the current position is the bottom-right cell, add the current path to the result and return.
  - Mark the current cell as blocked.
  - Iterate through all possible directions.
    - Calculate the next position based on the current direction.
    - If the next position is valid (i.e, if isValid() return true), append the direction to the current path and recursively call the findPath() function for the next cell.
    - Backtrack by removing the last direction from the current path.
- Mark the current cell as unblocked before returning.

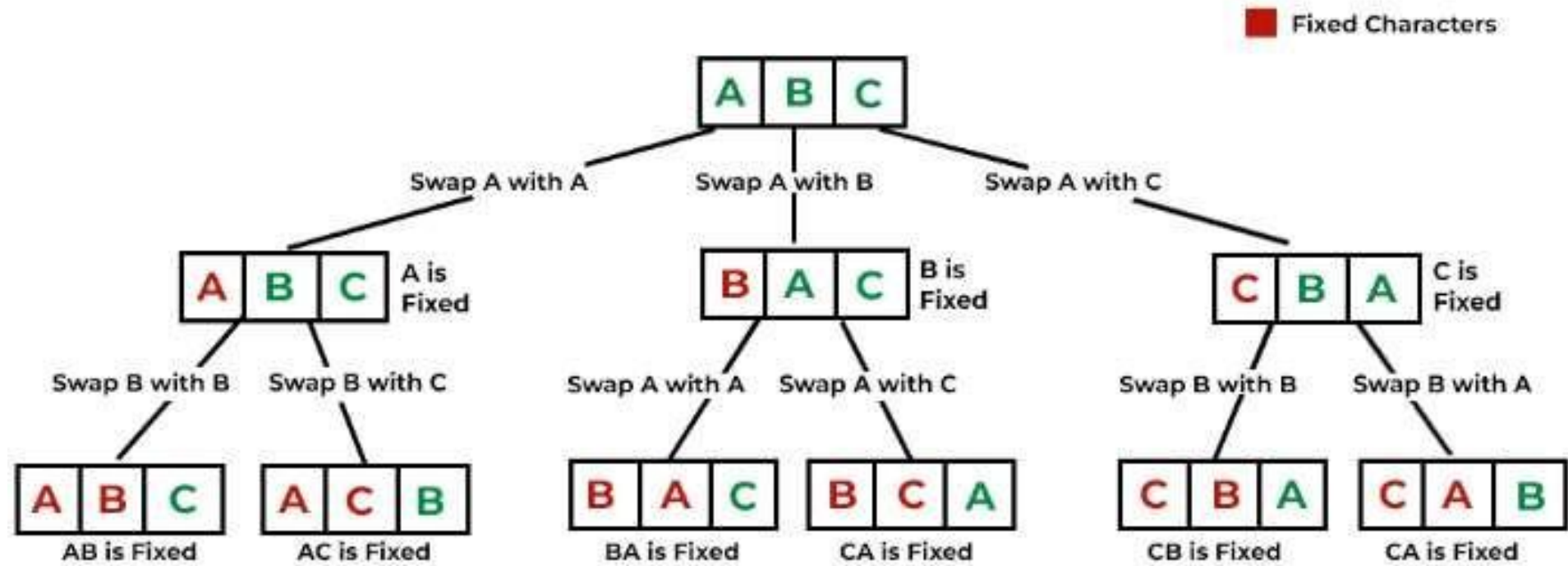
# Permutation and Combination

A permutation also called an “arrangement number” or “order,” is a rearrangement of the elements of an ordered list  $S$  into a one-to-one correspondence with  $S$  itself. A string of length  $N$  has  $N!$  permutations.

Examples:

*Input:  $S = \text{"ABC"}$*

*Output: "ABC", "ACB", "BAC", "BCA", "CBA", "CAB"*





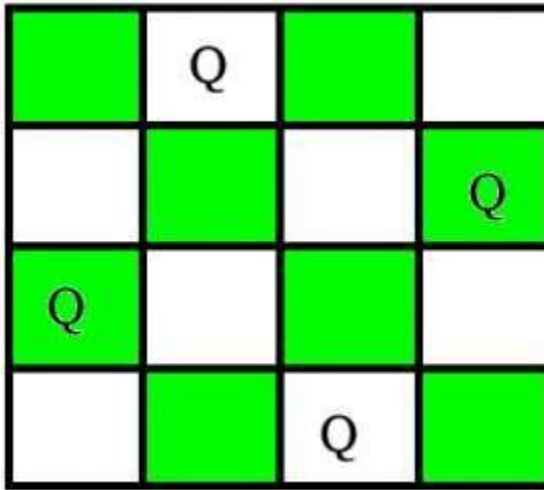
## Step-by-step approach:



- Create a function permute() with parameters as input string, starting index of the string, ending index of the string
- Call this function with values input string, 0, size of string – 1
  - In this function, if the value of L and R is the same then print the same string
    - Else run a for loop from L to R and swap the current element in the for loop with the inputString[L]
    - Then again call this same function by increasing the value of L by 1
    - After that again swap the previously swapped values to initiate backtracking

# N Queen Problem

The N Queen is the problem of placing N chess queens on an  $N \times N$  chessboard so that no two queens attack each other. For example, the following is a solution for 4 Queen problem.



	Q		
			Q
Q			
		Q	

## Step-by-step approach:



- Start in the leftmost column
- If all queens are placed, return true
- Try all rows in the current column. Do following for every tried row.
  - If the queen can be placed safely in this row then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.
  - If placing queen in [row, column] leads to a solution then return true.
  - If placing queen doesn't lead to a solution then unmark this [row, column] (Backtrack) and go to step (a) to try other rows.
- If all rows have been tried and nothing worked, return false to trigger backtracking.

# Subset Sum

Subset sum can also be thought of as a special case of the 0–1 Knapsack problem. For each item, there are two possibilities:

- Include the current element in the subset and recur for the remaining elements with the remaining Sum.
- Exclude the current element from the subset and recur for the remaining elements.

Finally, if Sum becomes 0 then print the elements of current subset. The recursion's base case would be when no items are left, or the sum becomes negative, then simply return.

## Examples:

Given a set[] of non-negative integers and a value sum, the task is to print the subset of the given set whose sum is equal to the given sum.

*Input: set[] = {1,2,1}, sum = 3*

*Output: [1,2],[2,1]*

*Explanation: There are subsets [1,2],[2,1] with sum 3.*

*Input: set[] = {3, 34, 4, 12, 5, 2}, sum = 30*

*Output: []*

*Explanation: There is no subset that add up to 30.*



## Concepts:

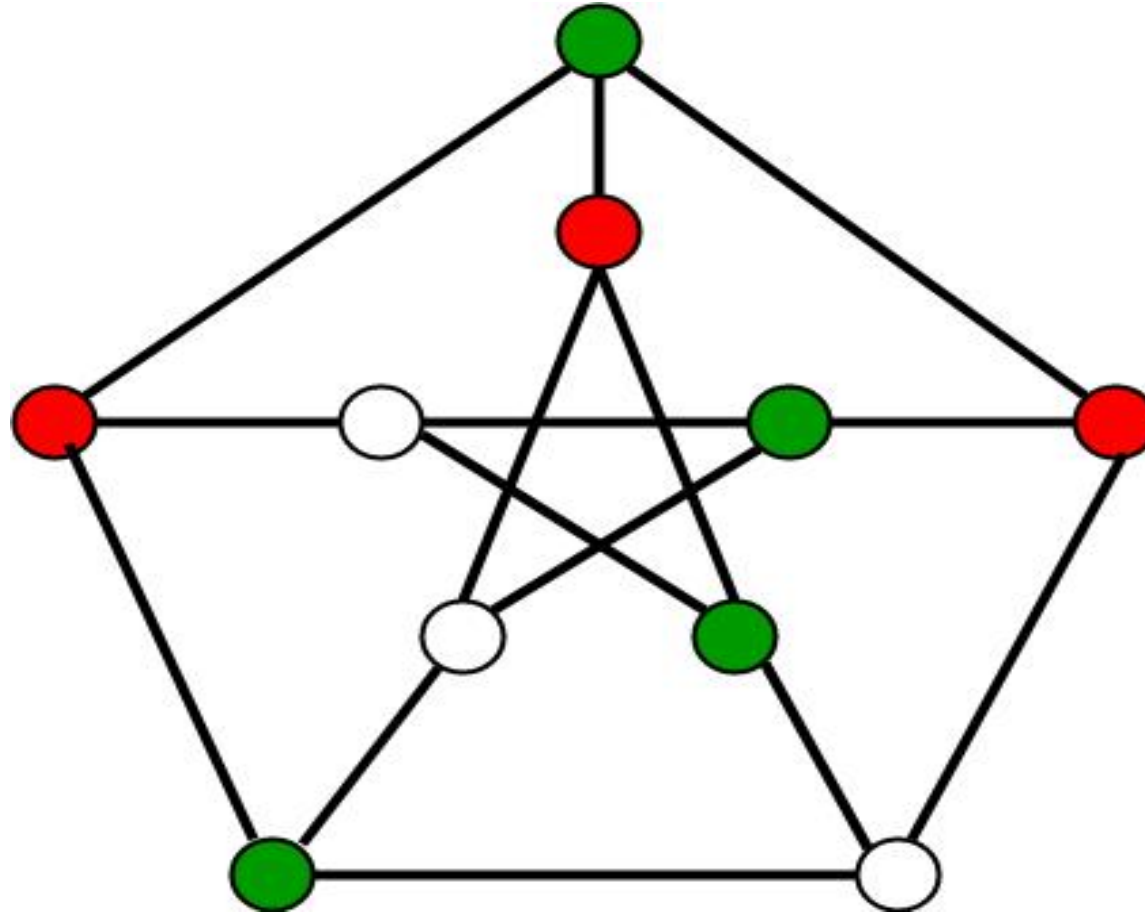
- M – Coloring Problem
- **Hamiltonian Cycle Problem**
- Sudoku Solver
- **Sieve of Sundaram**
- **Prime Numbers after P with Sum S**

## M – Coloring Problem

- Assign colors one by one to different vertices, starting from vertex 0.
- Before assigning a color, check for safety by considering already assigned colors to the adjacent vertices i.e check if the adjacent vertices have the same color or not.
- If there is any color assignment that does not violate the conditions, mark the color assignment as part of the solution.
- If no assignment of color is possible then backtrack and return false



# M – Coloring Problem

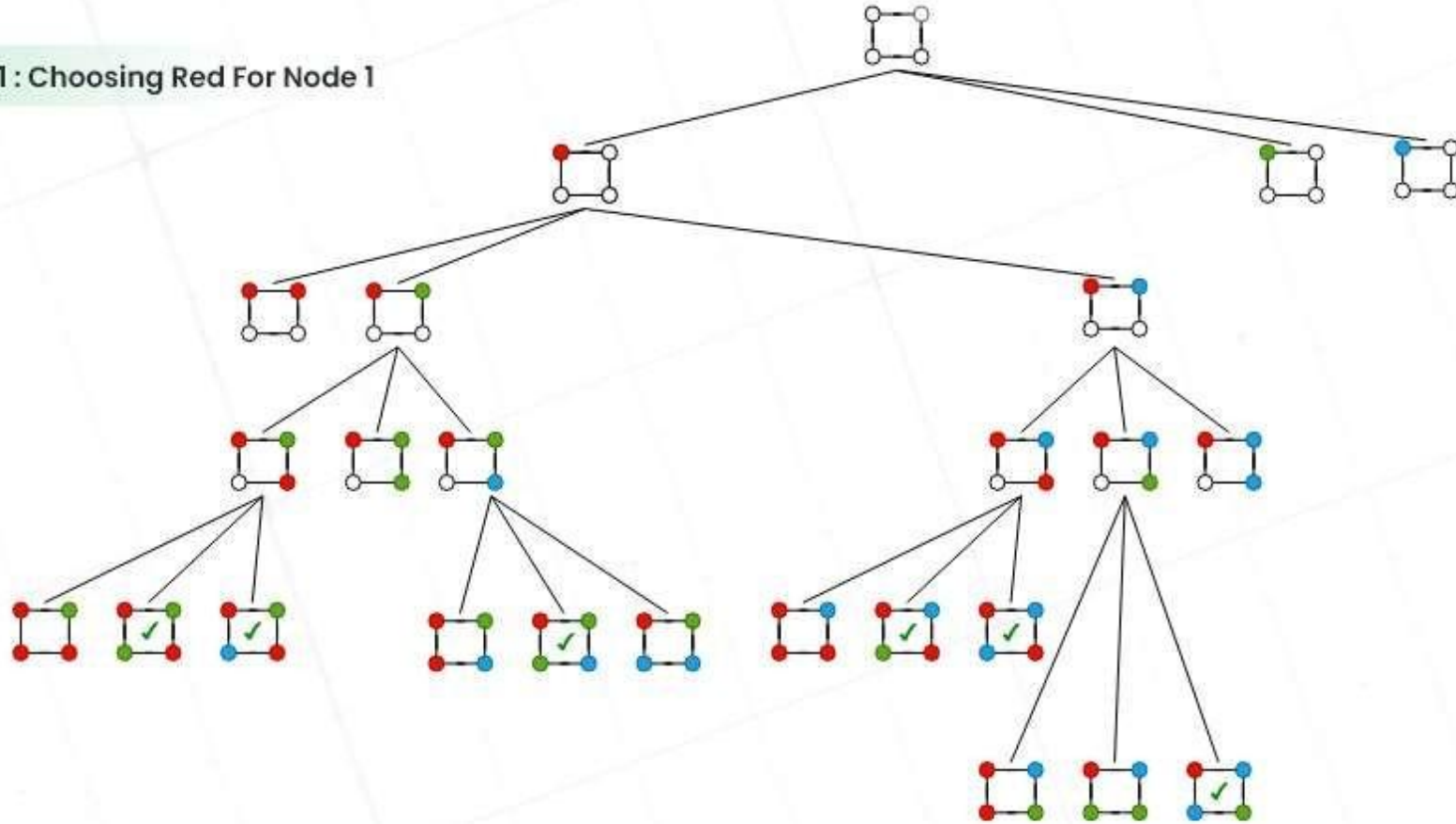


## M – Coloring Problem

- To color the graph, color each node one by one.
- To color the first node there are 3 choices of colors **Red, Green and Blue**, so lets take the **red** color for first node.
- After **Red** color for first node is fixed then we have made choice for second node in similar manner as we did for first node, then for 3rd node and so on.
- There is one important point to remember. while choosing color for the node, it should not be same as the color of the adjacent node.

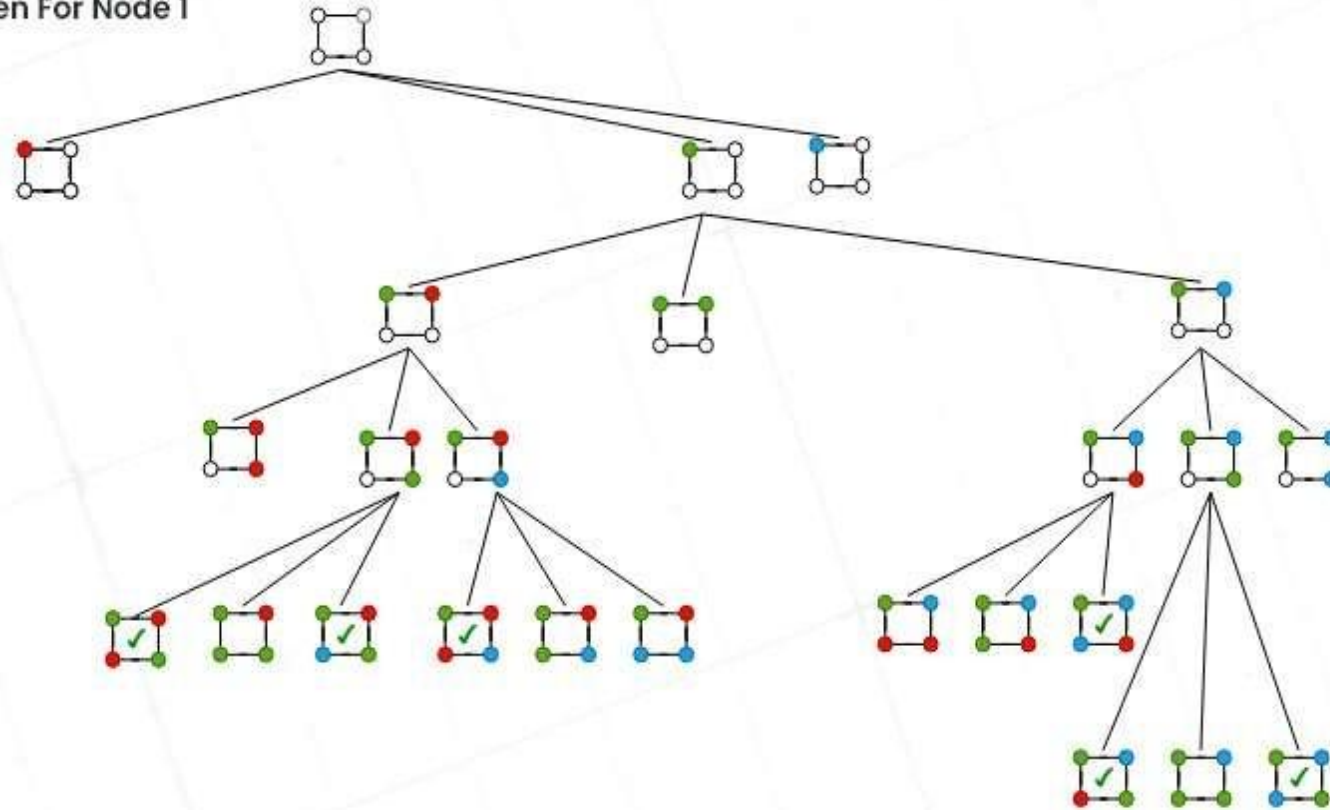
# M – Coloring Problem

Case 1 : Choosing Red For Node 1



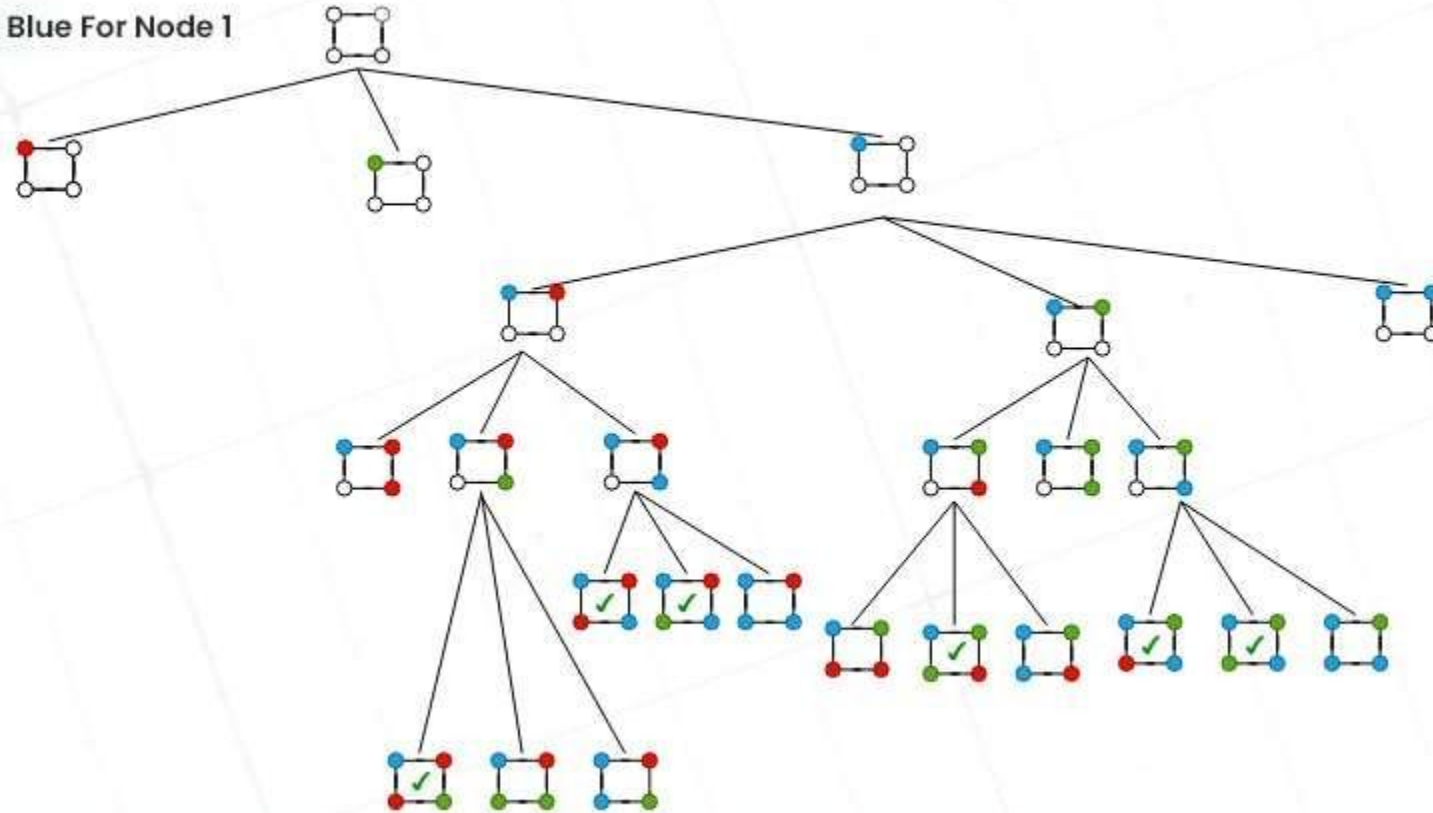
# M – Coloring Problem

Case 2 : Choosing Green For Node 1



# M – Coloring Problem

Case 3 : Choosing Blue For Node 1



# Hamiltonian Cycle Problem

- **Hamiltonian Cycle or Circuit** in a graph **G** is a cycle that visits every vertex of **G** exactly once and returns to the starting vertex.
- If graph contains a Hamiltonian cycle, it is called **Hamiltonian graph** otherwise it is **non-Hamiltonian**.
- Finding a Hamiltonian Cycle in a graph is a well-known NP Problem, which means that there's no known efficient algorithm to solve it for all types of graphs.
- However, it can be solved for small or specific types of graphs.  
The Hamiltonian Cycle problem has practical applications in various fields, such as **logistics, network design, and computer science**.

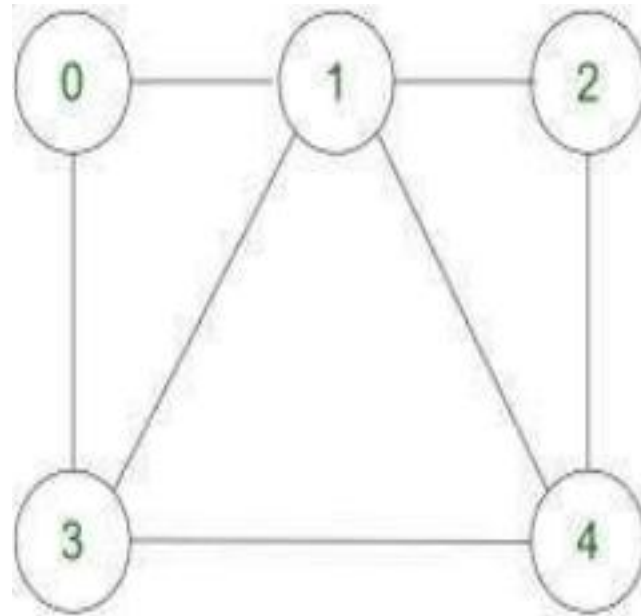
# Hamiltonian Cycle Problem

**Input :**

`graph[][] = {{0, 1, 0, 1, 0},{1, 0, 1, 1, 1},{0, 1, 0, 0, 1},{1, 1, 0, 0, 1},{0, 1, 1, 1, 0}}`

**Output:**

`{0, 1, 2, 4, 3, 0}.`



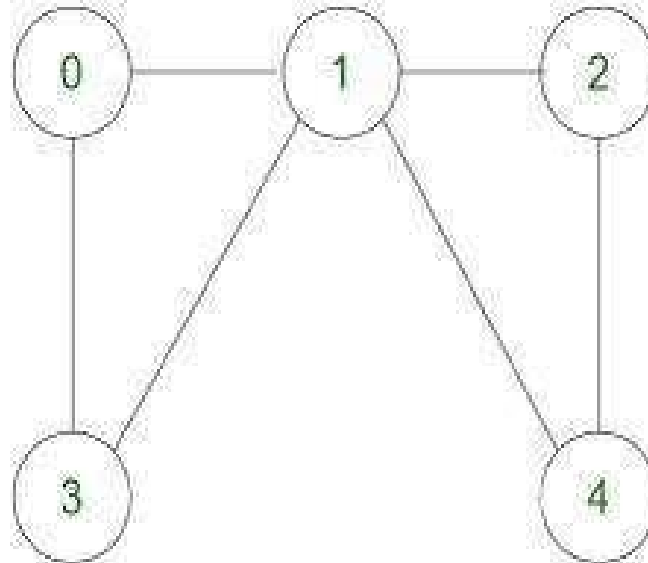
# Hamiltonian Cycle Problem

Input :

$\text{graph}[][] = \{\{0, 1, 0, 1, 0\}, \{1, 0, 1, 1, 1\}, \{0, 1, 0, 0, 1\}, \{1, 1, 0, 0, 0\}, \{0, 1, 1, 0, 0\}\}$

Output:

Solution does not exist





# Sudoku Solver

Given a partially filled 9×9 2D array 'grid[9][9]', the goal is to assign digits (from 1 to 9) to the empty cells so that every row, column, and subgrid of size 3×3 contains exactly one instance of the digits from 1 to 9.

3		6	5		8	4		
5	2							
	8	7					3	1
		3		1			8	
9			8	6	3			5
	5			9		6		
1	3					2	5	
							7	4
		5	2		6	3		

# Sudoku Solver

0, 0} }

**Input:** grid

```
{ {3, 0, 6, 5, 0, 8, 4,  
0, 0},  
{5, 2, 0, 0, 0, 0, 0,  
0, 0},  
{0, 8, 7, 0, 0, 0, 0,  
3, 1},  
{0, 0, 3, 0, 1, 0, 0,  
8, 0},  
{9, 0, 0, 8, 6, 3, 0,  
0, 5},  
{0, 5, 0, 0, 9, 0, 6,  
0, 0},  
{1, 3, 0, 0, 0, 0, 2,  
5, 0},  
{0, 0, 0, 0, 0, 0, 0,  
7, 4},  
{0, 0, 5, 2, 0, 6, 3,
```

# Sudoku Solver

Output:

```
3 1 6 5 7 8 4 9 2
5 2 9 1 3 4 7 6 8
4 8 7 6 2 9 5 3 1
```

```
2 6 3 4 1 5 9 8 7
9 7 4 8 6 3 1 2 5
8 5 1 7 9 2 6 4 3
1 3 8 9 4 7 2 5 6
6 9 2 3 5 1 8 7 4
7 4 5 2 8 6 3 1 9
```

**Explanation:** Each row, column and 3\*3 box of the output matrix contains unique numbers.

# Sieve of Sundaram

printPrimes(n) [Prints all prime numbers smaller than n]

1) In general Sieve of Sundaram, produces primes smaller than  $(2*x + 2)$  for given number  $x$ . Since we want primes smaller than  $n$ , we reduce  $n-1$  to half. We call it  $nNew$ .  $nNew = (n-1)/2$ ;

For example, if  $n = 102$ , then  $nNew = 50$ . if  $n = 103$ , then  $nNew = 51$

2) Create an array `marked[n]` that is going to be used to separate numbers of the form  $i+j+2ij$  from others where  $1 \leq i \leq j$

3) Initialize all entries of `marked[]` as false.

4) // Mark all numbers of the form  $i + j + 2ij$  as true // where  $1 \leq i \leq j$  Loop for  $i=1$  to  $nNew$  a)  $j = i$ ;

5) Loop While  $(i + j + 2*i*j) \leq nNew$ , then print  $2i + 1$  as first prime.

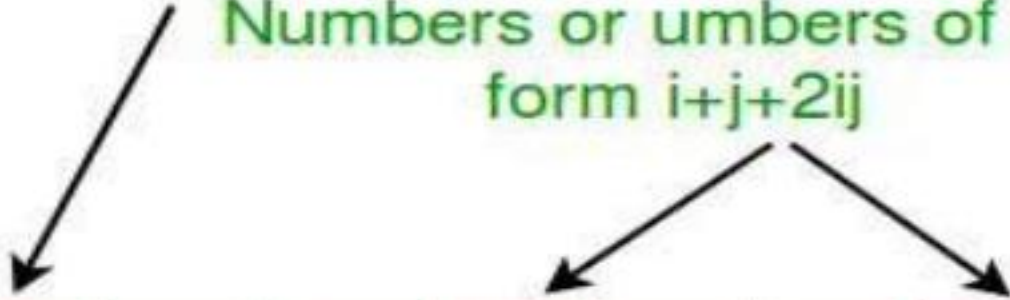
6) Remaining primes are of the form  $2i + 1$  where  $i$  is index of NOT marked numbers. So print  $2i + 1$  for all  $i$  such that `marked[i]` is false.

# Sieve of Sundaram

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

# Sieve of Sundaram

Marked or Removed  
Numbers or umbers of the  
form  $i+j+2ij$



1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

# Prime Numbers after P with Sum S

**Input :**

N = 2, P = 7, S = 28

**Output :**

11 17

**Explanation :**

11 and 17 are  
primes after  
prime 7 and (11 +  
17 = 28)

23

7 11 17 19

**Explanation :**

**Input :**

N = 4, P = 3, S  
= 54

**Output :**

5 7 11 31

5 7 13 29

5 7 19 23

5 13 17

19

7 11 13



## Prime Numbers after P with Sum S

Input: N = 3, P = 2, S = 23

Output: 3 7 13

Explanation: 3, 5, 7, 11 and 13 are primes after prime 2. And  $(3 + 7 + 13 = 5 + 7 + 11 = 23)$   
All are prime numbers and their sum is 54



# Prime Numbers after P with Sum S

Prime less than 10 and greater than 2 are: 3, 5, 7

