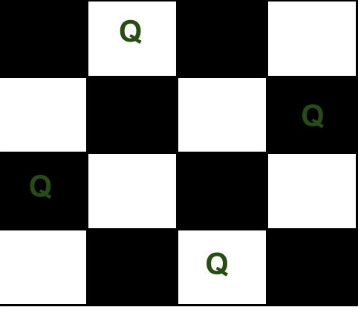
**19I510 Design and Analysis of Algorithms**

**Exercise 9 – Backtracking**

1. **N Queen Problem**

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

The expected output is a binary matrix that has 1s for the blocks where queens are placed. For example, the following is the output matrix for the above 4 queen solution.



The expected output is a binary matrix that has 1s for the blocks where queens are placed. For example, the following is the output matrix for the above 4 queen solution.

{ 0, 1, 0, 0}

{ 0, 0, 0, 1}

{ 1, 0, 0, 0}

{ 0, 0, 1, 0}

Algorithm:

**solveNQueen(board, col)**

**Input − The chess board, the col where the queen is trying to be placed.**

**Output − The position matrix where queens are placed.**

Begin

if all columns are filled, then

return true

for each row of the board, do

if isValid(board, i, col), then

set queen at place (i, col) in the board

if solveNQueen(board, col+1) = true, then

return true

otherwise remove queen from place (i, col) from board.

done

return false

End

**isValid(board, row, col)**

**Input: The chess board, row and the column of the board.**

**Output − True when placing a queen in row and place position is a valid or not.**

Begin

if there is a queen at the left of current col, then

return false

if there is a queen at the left upper diagonal, then

return false

if there is a queen at the left lower diagonal, then

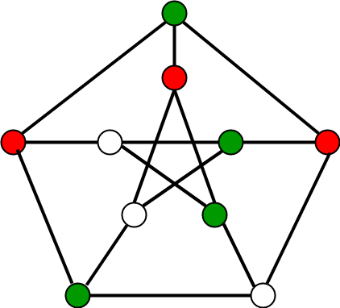
return false;

return true //otherwise it is valid place

End

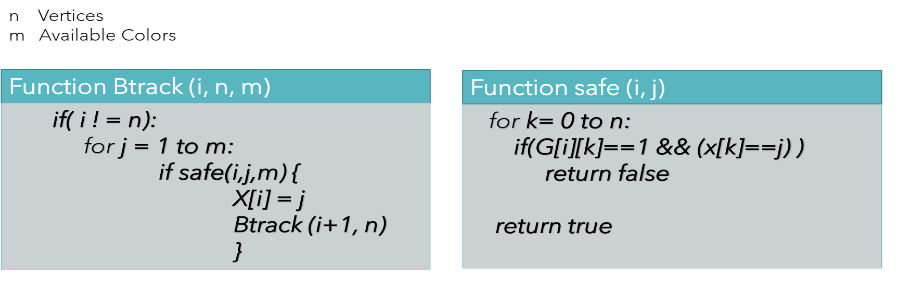
1. **m Coloring Problem**

Given an undirected graph and a number m, determine if the graph can be coloured with at most m colours such that no two adjacent vertices of the graph are colored with the same color. Here coloring of a graph means the assignment of colors to all vertices.



**Algorithm:**

1. Create a recursive function that takes the graph, current index, number of vertices, and output color array.
2. If the current index is equal to the number of vertices. Print the color configuration in output array.
3. Assign a color to a vertex (1 to m).
4. For every assigned color, check if the configuration is safe, (i.e. check if the adjacent vertices do not have the same color) recursively call the function with next index and number of vertices
5. If any recursive function returns true break the loop and return true.
6. If no recursive function returns true then return false.



**Input-Output format:**

***Input:***

1. A 2D array graph[V][V] where V is the number of vertices in graph and graph[V][V] is an adjacency matrix representation of the graph. A value graph[i][j] is 1 if there is a direct edge from i to j, otherwise graph[i][j] is 0.
2. An integer m is the maximum number of colors that can be used.

***Output:***   
An array color[V] that should have numbers from 1 to m. color[i] should represent the color assigned to the ith vertex. The code should also return false if the graph cannot be colored with m colors.

**Example:**

**Input:**

graph = {0, 1, 1, 1},

{1, 0, 1, 0},

{1, 1, 0, 1},

{1, 0, 1, 0}

**Output:**

Solution Exists:

Following are the assigned colors

1 2 3 2

**Explanation:** By coloring the vertices

with following colors, adjacent

vertices does not have same colors

**Input:**

graph = {1, 1, 1, 1},

{1, 1, 1, 1},

{1, 1, 1, 1},

{1, 1, 1, 1}

**Output:** Solution does not exist.

**Explanation:** No solution exits.

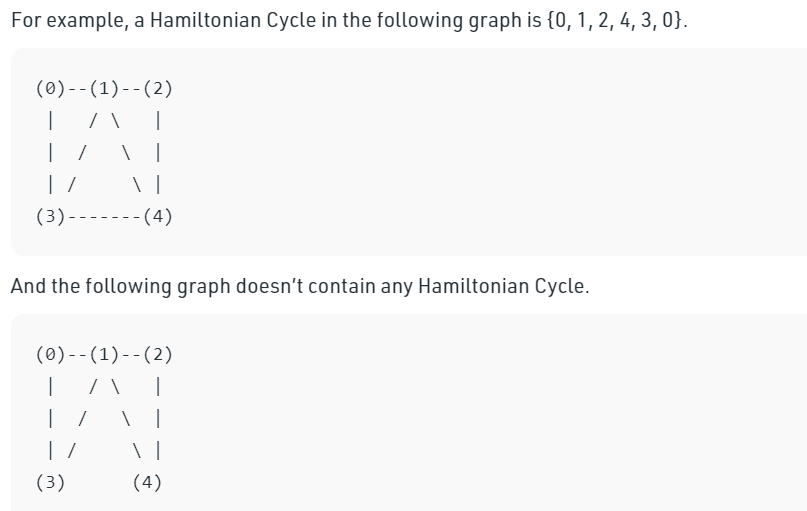
1. **Hamiltonian Cycle**

Hamiltonian Path in an undirected graph is a path that visits each vertex exactly once. A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there is an edge (in the graph) from the last vertex to the first vertex of the Hamiltonian Path. Determine whether a given graph contains Hamiltonian Cycle or not. If it contains, then prints the path. Following are the input and output of the required function.

*Input:*   
A 2D array graph[V][V] where V is the number of vertices in graph and graph[V][V] is adjacency matrix representation of the graph. A value graph[i][j] is 1 if there is a direct edge from i to j, otherwise graph[i][j] is 0.

*Output:*   
An array path[V] that should contain the Hamiltonian Path. path[i] should represent the ith vertex in the Hamiltonian Path. The code should also return false if there is no Hamiltonian Cycle in the graph.  
  
**Backtracking Algorithm**

Create an empty path array and add vertex 0 to it. Add other vertices, starting from the vertex 1. Before adding a vertex, check for whether it is adjacent to the previously added vertex and not already added. If we find such a vertex, we add the vertex as part of the solution. If we do not find a vertex then we return false.

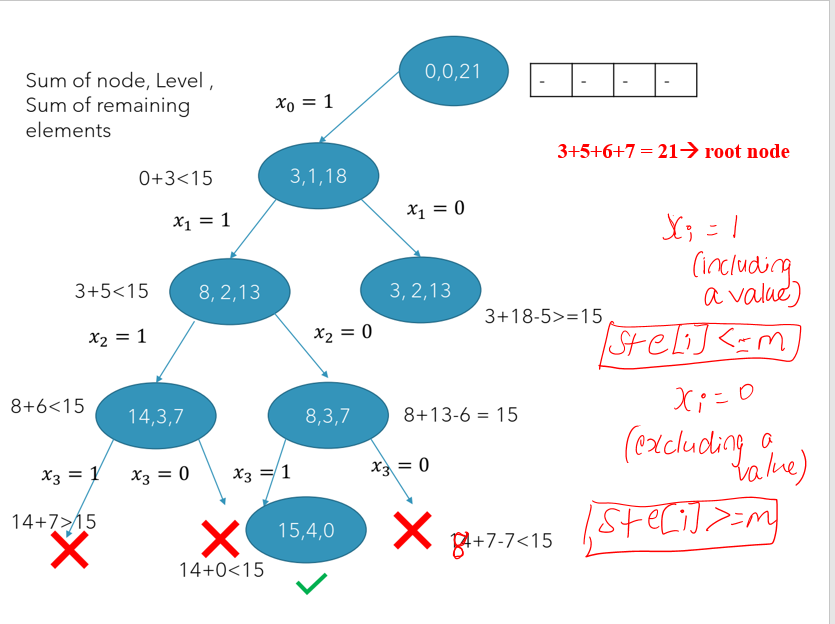


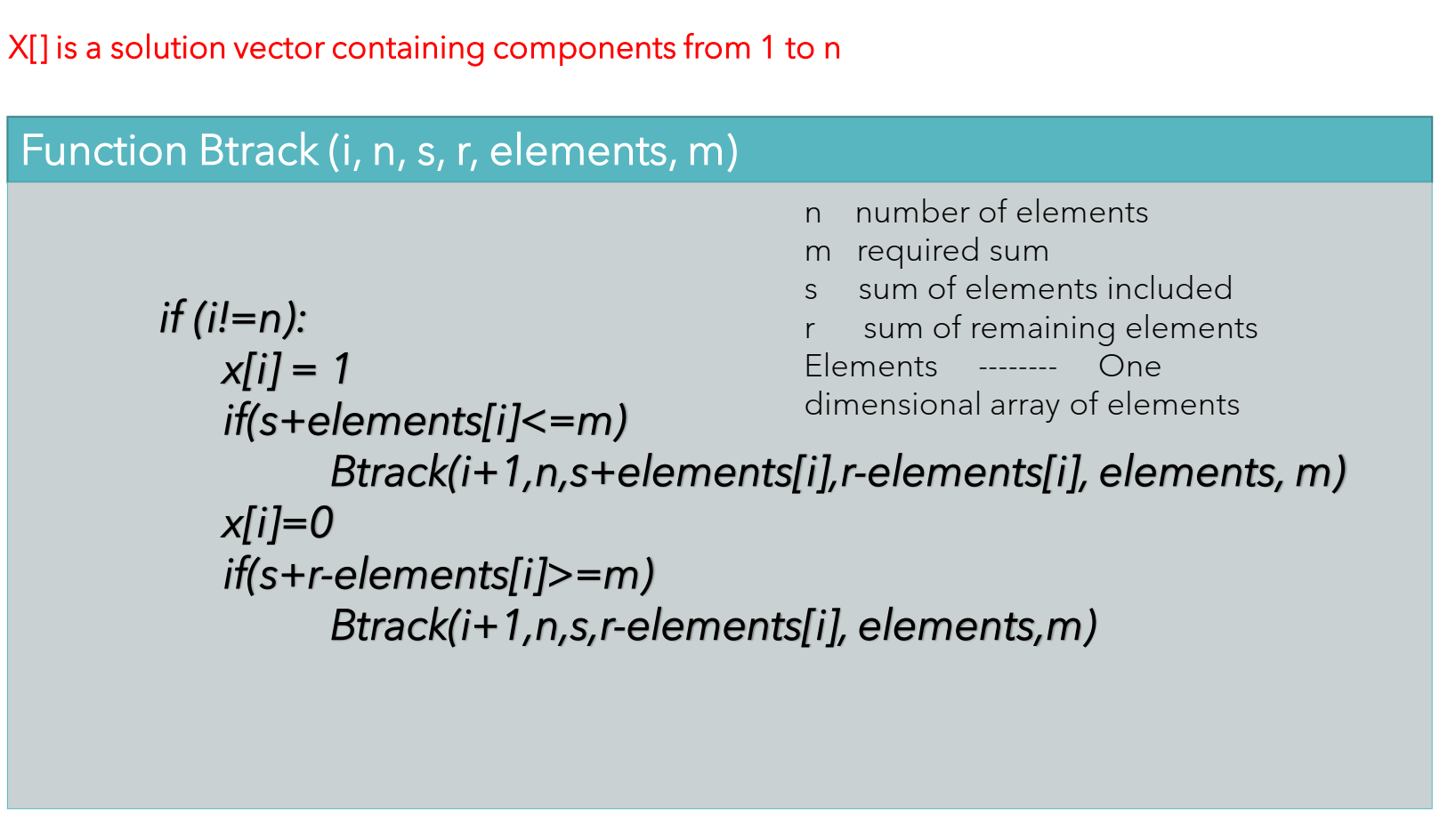
1. **Subset Sum**

Subset sum problem is to find subset of elements that are selected from a given set whose sum adds up to a given number K. We are considering the set contains non-negative values. It is assumed that the input set is unique (no duplicates are presented).

Example:

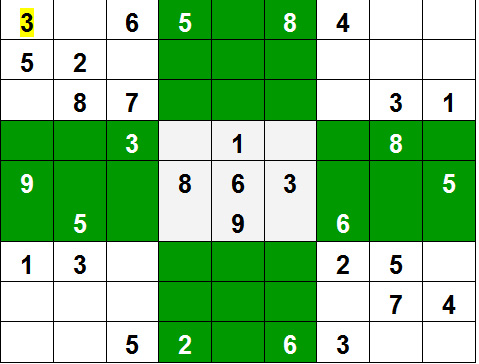
Consider 4 integers {3,5,6,7} and m = 15, then solution is {3,5,7}

Backtracking Approach:



1. **Sudoku**

Problem: 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. 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.



**Examples:**  
***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, 0, 0} }*.

***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.*

***Backtracking Approach:***

* Create a function that checks after assigning the current index the grid becomes unsafe or not. Keep Hashmap for a row, column and boxes. If any number has a frequency greater than 1 in the hashMap return false else return true; hashMap can be avoided by using loops.
* Create a recursive function that takes a grid.
* Check for any unassigned location.
  + If present then assigns a number from 1 to 9.
  + Check if assigning the number to current index makes the grid unsafe or not.
  + If safe then recursively call the function for all safe cases from 0 to 9.
  + If any recursive call returns true, end the loop and return true. If no recursive call returns true then return false.
* If there is no unassigned location then return true.

1. **Rat in a Maze**

Problem: A Maze is given as N\*N binary matrix of blocks where source block is the upper left most block i.e., maze[0][0] and destination block is lower rightmost block i.e., maze[N-1][N-1]. A rat starts from source and has to reach the destination. The rat can move only in two directions: forward and down.

In the maze matrix, 0 means the block is a dead end and 1 means the block can be used in the path from source to destination. Note that this is a simple version of the typical Maze problem. For example, a more complex version can be that the rat can move in 4 directions and a more complex version can be with a limited number of moves.

Gray blocks are dead ends (value = 0).

Following is a binary matrix representation of the above maze.

{1, 0, 0, 0}

{1, 1, 0, 1}

{0, 1, 0, 0}

{1, 1, 1, 1}

Following is a maze with highlighted solution path

Following is the solution matrix (output of program) for the above input matrix.

{1, 0, 0, 0}

{1, 1, 0, 0}

{0, 1, 0, 0}

{0, 1, 1, 1}

All entries in solution path are marked as 1.

**Backtracking Algorithm:**

1. Create a solution matrix, initially filled with 0’s.
2. Create a recursive function, which takes initial matrix, output matrix and position of rat (i, j).
3. if the position is out of the matrix or the position is not valid then return.
4. Mark the position output[i][j] as 1 and check if the current position is destination or not. If destination is reached print the output matrix and return.
5. Recursively call for position (i+1, j) and (i, j+1).
6. Unmark position (i, j), i.e output[i][j] = 0.