1. **Write a program to implement Horspool's algorithm for String Matching.**

```
import java.util.*;
public class Horspool  {

   public static int SIZE=500;
   public static int table[]=new int[SIZE];

public void shifttable(String pattern) {

 int i,j,m;
 char p[] = pattern.toCharArray();
 m=pattern.length();

 for (i=0;i<SIZE;i++)
    table[i]=m;
 for (j=0;j<m;j++)
    table[p[j]]=m-1-j;
}
public int horspool(String source,String pattern)
  {
      int i,k,pos,m;
      char s[] = source.toCharArray();
      char p[] = pattern.toCharArray();
      m=pattern.length();

      for(i=m-1;i<source.length();)
        {
          k=0;
           while((k<m) && (p[m-1-k] == s[i-k]))
             k++;
          if(k==m)
     {   pos=i-m+2;
             return pos;
     }
          else
             i+=table[s[i]];
        }
       return -1;
  }
     public static void main(String []args){
       int pos;
  String source=args[0];
       String pattern = args[1];
```

```
        Horspool  h = new Horspool ();

        h.shifttable(pattern);
        pos = h.horspool(source,pattern);

        if(pos == -1)
            System.out.println("PATTERN NOT FOUND");
        else
            System.out.println("PATTERN FOUND FROM POSITION: \t"+pos+"\n");
    }
}
```

Output :
 PATTERN FOUND FROM POSITION: **4**

2. Sort a given set of N integer elements using Heap Sort technique and compute its time taken.

```
import java.util.Scanner;

public class HeapSort{
      private static int N;
    public static void sort(int arr[]){
                heapMethod(arr);
        for (int i = N; i > 0; i--){
            swap(arr,0, i);
            N = N-1;
            heap(arr, 0);
        }
    }
    public static void heapMethod(int arr[]){
        N = arr.length-1;
        for (int i = N/2; i >= 0; i--)
            heap(arr, i);
    }
    public static void heap(int arr[], int i){
        int left = 2*i ;
        int right = 2*i + 1;
        int max = i;
        if (left <= N && arr[left] > arr[i])
            max = left;
                if (right <= N && arr[right] > arr[max])
            max = right;
        if (max != i){
```

```
            swap(arr, i, max);
            heap(arr, max);
        }
    }
    public static void swap(int arr[], int i, int j){
        int tmp = arr[i];
        arr[i] = arr[j];
        arr[j] = tmp;
    }
    public static void main(String[] args) {
        Scanner in = new Scanner( System.in );
        int n;
        System.out.println("Enter the number of elements to be sorted:");
        n = in.nextInt();
        int arr[] = new int[ n ];
        System.out.println("Enter "+ n +" integer elements");
        for (int i = 0; i < n; i++)
            arr[i] = in.nextInt();
        sort(arr);
        System.out.println("After sorting ");
        for (int i = 0; i < n; i++)
            System.out.println(arr[i]+" ");
        System.out.println();
    }
}
```

Output is:

```
Enter the number of elements to be sorted:
6
Enter 6 integer elements
99
54
67
32
1
78
After sorting
1
32
54
67
78
99
```

3. Implement in Java, the 0/1 Knapsack problem using (a) Dynamic Programming method (b) Greedy method.

(a) Dynamic Programming method

```
import java.util.Scanner;

public class KnapsackDP {
	static final int MAX = 20; // max. no. of objects
	static int w[];  // weights 0 to n-1
	static int p[];  // profits 0 to n-1
	static int n;		// no. of objects
	static int M;		// capacity of Knapsack
	static int V[][];	// DP solution process - table
	static int Keep[][]; // to get objects in optimal solution

	public static void main(String args[]) {
		w = new int[MAX];
		p = new int[MAX];
		V = new int [MAX][MAX];
		Keep = new int[MAX][MAX];
		int optsoln;
		ReadObjects();
		for (int i = 0; i <= M; i++)
			V[0][i] = 0;
		for (int i = 0; i <= n; i++)
			V[i][0] = 0;
		optsoln = Knapsack();
		System.out.println("Optimal solution = " + optsoln);
	}

	static int Knapsack() {
		int r; // remaining Knapsack capacity
		for (int i = 1; i <= n; i++)
			for (int j = 0; j <= M; j++)
				if ((w[i] <= j) && (p[i] + V[i - 1][j - w[i]] > V[i - 1][j])) {
					V[i][j] = p[i] + V[i - 1][j - w[i]];
					Keep[i][j] = 1;
				} else {
					V[i][j] = V[i - 1][j];
					Keep[i][j] = 0;
				}

		// Find the objects included in the Knapsack
		r = M;
		System.out.println("Items = ");
```

```
		for (int i = n; i > 0; i--) // start from Keep[n,M]
			if (Keep[i][r] == 1) {
				System.out.println(i + " ");
				r = r - w[i];
			}
		System.out.println();
		return V[n][M];
	}

	static void ReadObjects() {
		Scanner scanner = new Scanner(System.in);
		System.out.println("Knapsack Problem - Dynamic Programming Solution: 
");
		System.out.println("Enter the max capacity of knapsack: ");
		M = scanner.nextInt();
		System.out.println("Enter number of objects:  ");
		n = scanner.nextInt();
		System.out.println("Enter Weights: ");
		for (int i = 1; i <= n; i++)
			w[i] = scanner.nextInt();
		System.out.println("Enter Profits: ");
		for (int i = 1; i <= n; i++)
			p[i] = scanner.nextInt();
		scanner.close();
	}
}
```

Output

```
Knapsack Problem - Dynamic Programming Solution:
Enter the max capacity of knapsack:
5
Enter number of objects:
4
Enter Weights:
1
2
2
1
Enter Profits:
15
20
10
12
Items =
4
2
```

1

Optimal solution = 47

(b) Greedy method.

```
import java.util.Scanner;
class KObject {                                          // Knapsack object details
        float w;
        float p;
        float r;
}
public class KnapsackGreedy {
        static final int MAX = 20;      // max. no. of objects
        static int n;                           // no. of objects
        static float M;                         // capacity of Knapsack

        public static void main(String args[]) {
                Scanner scanner = new Scanner(System.in);
                System.out.println("Enter number of objects:  ");
                n = scanner.nextInt();
                KObject[] obj = new KObject[n];
                for(int i = 0; i<n;i++)
                        obj[i] = new KObject();// allocate memory for members

                ReadObjects(obj);
                Knapsack(obj);
                scanner.close();
        }

        static void ReadObjects(KObject obj[]) {
                KObject temp = new KObject();
                Scanner scanner = new Scanner(System.in);
                System.out.println("Enter the max capacity of knapsack: ");
                M = scanner.nextFloat();

                System.out.println("Enter Weights: ");
                for (int i = 0; i < n; i++)
                        obj[i].w = scanner.nextFloat();

                System.out.println("Enter Profits: ");
                for (int i = 0; i < n; i++)
                        obj[i].p = scanner.nextFloat();

                for (int i = 0; i < n; i++)
                        obj[i].r = obj[i].p / obj[i].w;
```

```
		// sort objects in descending order, based on p/w ratio
		for(int i = 0; i<n-1; i++)
			for(int j=0; j<n-1-i; j++)
				if(obj[j].r < obj[j+1].r){
					temp = obj[j];
					obj[j] = obj[j+1];
					obj[j+1] = temp;
				}
		scanner.close();
	}

	static void Knapsack(KObject kobj[]) {
		float x[] = new float[MAX];
		float totalprofit;
		int i;
		float U;  // U place holder for M
		U = M;
		totalprofit = 0;
		for (i = 0; i < n; i++)
			x[i] = 0;
		for (i = 0; i < n; i++) {
			if (kobj[i].w > U)
				break;
			else {
				x[i] = 1;
				totalprofit = totalprofit + kobj[i].p;
				U = U - kobj[i].w;
			}
		}
		System.out.println("i = " + i);
		if (i < n)
			x[i] = U / kobj[i].w;
		totalprofit = totalprofit + (x[i] * kobj[i].p);
		System.out.println("The Solution vector, x[]: ");
		for (i = 0; i < n; i++)
			System.out.print(x[i] + " ");
		System.out.println("\nTotal profit is = " + totalprofit);
	}
}
```

Output

Enter number of objects:
4
Enter the max capacity of knapsack:

5
Enter Weights: 1 2 2 1
Enter Profits:
15
20
10
12
i = 3
The Solution vector, x[]:
1.0 1.0 1.0 0.5
Total profit is = 52.0

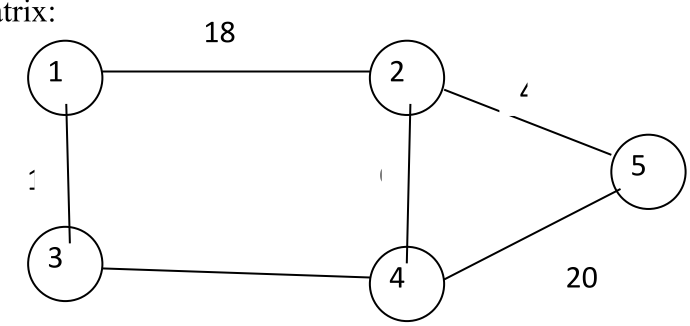
4. From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm. Write the program in Java.

```
import java.util.*;

public class DijkstrasClass {

        final static int MAX = 20;
        final static int infinity = 9999;
        static int n;           // No. of vertices of G
        static int a[][]; // Cost matrix
        static Scanner scan = new Scanner(System.in);

        public static void main(String[] args) {
                ReadMatrix();
                int s = 0;              // starting vertex
                System.out.println("Enter starting vertex: ");
                s = scan.nextInt();
                Dijkstras(s);   // find shortest path
        }

        static void ReadMatrix() {
                a = new int[MAX][MAX];
                System.out.println("Enter the number of vertices:");
                n = scan.nextInt();
                System.out.println("Enter the cost adjacency matrix:");
                for (int i = 1; i <= n; i++)
                        for (int j = 1; j <= n; j++)
                                a[i][j] = scan.nextInt();
        }

        static void Dijkstras(int s) {
                int S[] = new int[MAX];
                int d[] = new int[MAX];
                int u, v;
                int i;
                for (i = 1; i <= n; i++) {
                        S[i] = 0;
                        d[i] = a[s][i];
                }
                S[s] = 1;
                d[s] = 1;
                i = 2;
                while (i <= n) {
                        u = Extract_Min(S, d);
                        S[u] = 1;
```

```
				i++;
				for (v = 1; v <= n; v++) {
					if (((d[u] + a[u][v] < d[v]) && (S[v] == 0)))
						d[v] = d[u] + a[u][v];
				}
			}
			for (i = 1; i <= n; i++)
				if (i != s)
					System.out.println(i + ":" + d[i]);
		}

		static int Extract_Min(int S[], int d[]) {
			int i, j = 1, min;
			min = infinity;
			for (i = 1; i <= n; i++) {
				if ((d[i] < min) && (S[i] == 0)) {
					min = d[i];
					j = i;
				}
			}
			return (j);
		}
}
```

Output
Enter the number of vertices:
5
Enter the cost adjacency matrix:

0 18 1 9999 9999
18 0 9999 6 4
1 9999 0 2 9999
9999 6 2 0 20
9999 4 9999 20 0



Enter starting vertex:
1
2:9
3:1
4:3
5:13

5. **Find Minimum Cost Spanning Tree of a given connected undirected graph using Kruskal'salgorithm. Use Union-Find algorithms in your program.**

```
import java.util.Scanner;

public class KruskalsClass {

        final static int MAX = 20;
        static int n; // No. of vertices of G
        static int cost[][]; // Cost matrix
        static Scanner scan = new Scanner(System.in);

        public static void main(String[] args) {
                ReadMatrix();
                Kruskals();
        }

        static void ReadMatrix() {

                int i, j;
                cost = new int[MAX][MAX];

                System.out.println("Implementation of Kruskal's algorithm");
                System.out.println("Enter the no. of vertices");
                n = scan.nextInt();

                System.out.println("Enter the cost adjacency matrix");
                for (i = 1; i <= n; i++) {
                        for (j = 1; j <= n; j++) {
                                cost[i][j] = scan.nextInt();
                                if (cost[i][j] == 0)
                                        cost[i][j] = 999;
                        }
                }
        }

        static void Kruskals() {
```

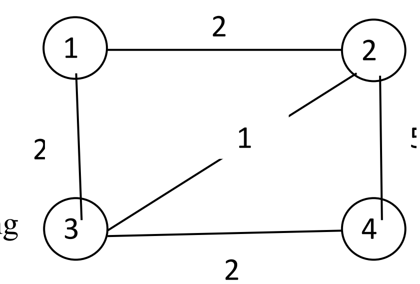
```
        int a = 0, b = 0, u = 0, v = 0, i, j, ne = 1, min, mincost = 0;

        System.out.println("The edges of Minimum Cost Spanning Tree are");
        while (ne < n) {
            for (i = 1, min = 999; i <= n; i++) {
                for (j = 1; j <= n; j++) {
                    if (cost[i][j] < min) {
                        min = cost[i][j];
                        a = u = i;
                        b = v = j;
                    }
                }
            }
            u = find(u);
            v = find(v);
            if (u != v)
            {
                uni(u, v);
            Syste.out.println(ne++ + "edge (" + a + "," + b + ") =" + min);
                mincost += min;
            }
            cost[a][b] = cost[b][a] = 999;
        }
        System.out.println("Minimum cost :" + mincost);
    }

    static int find(int i) {
        int parent[] = new int[9];
        while (parent[i] == 1)
            i = parent[i];
        return i;
    }

    static void uni(int i, int j) {
        int parent[] = new int[9];
        parent[j] = i;
    }
}
```

Output

Enter the number of vertices: 4

Enter the cost adjacency matrix:

0	20	2	999
20	0	15	5
2	15	0	25
999	5	25	0



The edges of Minimum Cost Spanning Tree are

1edge (1,3) =2

2edge (2,4) =5

3edge (2,3) =15

Minimum cost :22

6. Find Minimum Cost Spanning Tree of a given connected undirected graph using Prim's algorithm.

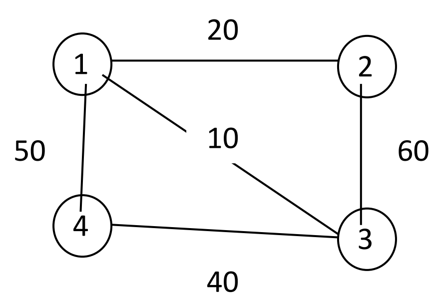
```
import java.util.Scanner;

public class PrimsClass {

        final static int MAX = 20;
        static int n; // No. of vertices of G
        static int cost[][]; // Cost matrix
        static Scanner scan = new Scanner(System.in);

        public static void main(String[] args) {
                ReadMatrix();
                Prims();
        }

        static void ReadMatrix() {
                int i, j;
                cost = new int[MAX][MAX];

                System.out.println("\n Enter the number of nodes:");
                n = scan.nextInt();
                System.out.println("\n Enter the cost matrix:\n");
                for (i = 1; i <= n; i++)
                        for (j = 1; j <= n; j++) {
                                cost[i][j] = scan.nextInt();
                                if (cost[i][j] == 0)
                                        cost[i][j] = 999;
                        }
        }

        static void Prims() {

                int visited[] = new int[10];
                int ne = 1, i, j, min, a = 0, b = 0, u = 0, v = 0;
                int mincost = 0;
```

```
        visited[1] = 1;
        while (ne < n) {
            for (i = 1, min = 999; i <= n; i++)
                for (j = 1; j <= n; j++)
                    if (cost[i][j] < min)
                        if (visited[i] != 0) {
                            min = cost[i][j];
                            a = u = i;
                            b = v = j;
                        }
            if (visited[u] == 0 || visited[v] == 0) {
    System.out.println("Edge" + ne++ + ":(" + a + "," + b + ")" + "cost:" + min);
                mincost += min;
                visited[b] = 1;
            }
            cost[a][b] = cost[b][a] = 999;
        }
        System.out.println("\n Minimun cost" + mincost);
    }

}
```

Output

Enter the number of nodes: 4
Enter the cost matrix:

0	20	10	50
20	0	60	999
10	60	0	40
50	999	40	0



Enter Source: 1
1 --> 3 = 10   Sum = 10
1 --> 2 = 20   Sum = 30
3 --> 4 = 40   Sum = 70

Total cost: 70

7. **Write Java programs to**
(a) Implement All-Pairs Shortest Paths problem using Floyd's algorithm.

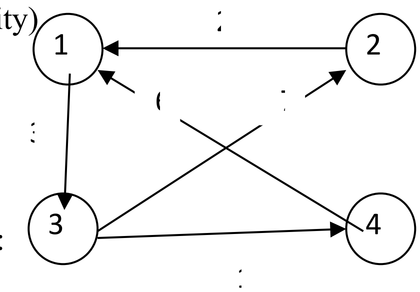
```
import java.util.Scanner;
public class FloydsClass {
static final int MAX = 20;      // max. size of cost matrix
static int a[][];               // cost matrix
static int n;                           // actual matrix size

public static void main(String args[]) {
        a = new int[MAX][MAX];
        ReadMatrix();
        Floyds();                               // find all pairs shortest path
        PrintMatrix();
}

static void ReadMatrix() {
        System.out.println("Enter the number of vertices\n");
        Scanner scanner = new Scanner(System.in);
        n = scanner.nextInt();
        System.out.println("Enter the Cost Matrix (999 for infinity) \n");
        for (int i = 1; i <= n; i++) {
                for (int j = 1; j <= n; j++) {
                        a[i][j] = scanner.nextInt();
                }
        }
        scanner.close();
}

static void Floyds() {
        for (int k = 1; k <= n; k++) {
                for (int i = 1; i <= n; i++)
                        for (int j = 1; j <= n; j++)
                                if ((a[i][k] + a[k][j]) < a[i][j])
                                        a[i][j] = a[i][k] + a[k][j];
        }
}

static void PrintMatrix() {
        System.out.println("The All Pair Shortest Path Matrix is:\n");
        for(int i=1; i<=n; i++)
        {
                for(int j=1; j<=n; j++)
                        System.out.print(a[i][j] + "\t");
                System.out.println("\n");
        }
```

```
        }
}
```

Output

Enter the number of vertices: 4
EEnter the Cost Matrix (999 for infinity)

0	999	3	999
2	0	999	999
999	7	0	1
6	999	999	0



TThe All Pair Shortest Path Matrix is:

0	10	3	4
2	0	5	6
7	7	0	1
6	16	9	0

(b) **Implement Travelling Sales Person problem using Dynamic programming.**

```
import java.util.Scanner;

public class TravSalesPerson {
static int MAX = 100;
static final int infinity = 999;

public static void main(String args[]) {
		int cost = infinity;
	int c[][] = new int[MAX][MAX];	// cost matrix
	int tour[] = new int[MAX];		// optimal tour
	int n;						// max. cities
	System.out.println("Travelling Salesman Problem using Dynamic
Programming\n");
	System.out.println("Enter number of cities: ");
	Scanner scanner = new Scanner(System.in);
	n = scanner.nextInt();
	System.out.println("Enter Cost matrix:\n");
	for (int i = 0; i < n; i++)
		for (int j = 0; j < n; j++) {
			c[i][j] = scanner.nextInt();
			if (c[i][j] == 0)
				c[i][j] = 999;
		}
	for (int i = 0; i < n; i++)
		tour[i] = i;
	cost = tspdp(c, tour, 0, n);
	// print tour cost and tour
	System.out.println("Minimum Tour Cost: " + cost);
	System.out.println("\nTour:");
	for (int i = 0; i < n; i++) {
		System.out.print(tour[i] + " -> ");
	}
	System.out.println(tour[0] + "\n");
	scanner.close();
}

static int tspdp(int c[][], int tour[], int start, int n) {
	int i, j, k;
	int temp[] = new int[MAX];
	int mintour[] = new int[MAX];
	int mincost;
	int cost;
	if (start == n - 2)
		return c[tour[n - 2]][tour[n - 1]] + c[tour[n - 1]][0];
```

```
        mincost = infinity;
        for (i = start + 1; i < n; i++) {
                for (j = 0; j < n; j++)
                        temp[j] = tour[j];
                temp[start + 1] = tour[i];
                temp[i] = tour[start + 1];
                if (c[tour[start]][tour[i]] + (cost = tspdp(c, temp, start + 1, n)) < mincost) {
                        mincost = c[tour[start]][tour[i]] + cost;
                        for (k = 0; k < n; k++)
                                mintour[k] = temp[k];
                }
        }
        for (i = 0; i < n; i++)
                tour[i] = mintour[i];
        return mincost;
}
}
```

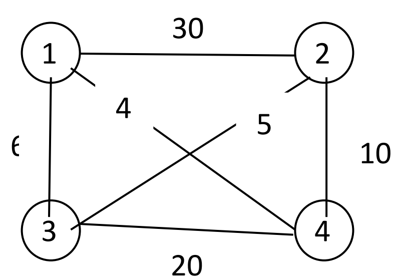
Output

Travelling Salesman Problem using Dynamic Programming

Enter number of cities:

Enter cost matrix:

0	30	6	4
30	0	5	10
6	5	0	20
4	10	20	0



Minimum Tour Cost: 25

Tour:

0 -> 2 -> 1 -> 3 -> 0

8. **Design and implement in Java to find a subset of a given set S = {Sl, S2,.....,Sn} of *n* positive integers whose SUM is equal to a given positive integer *d*. For example, if S ={1, 2, 5, 6, 8} and *d*= 9, there are two solutions {1,2,6}and {1,8}. Display a suitable message, if the given problem instance doesn't have a solution.**

```
import java.util.Scanner;

public class SumOfsubset {
	final static int MAX = 10;
	static int n;
	static int S[];
	static int soln[];
	static int d;

	public static void main(String args[]) {
		S = new int[MAX];
		soln = new int[MAX];
		int sum = 0;
		Scanner scanner = new Scanner(System.in);
		System.out.println("Enter number of elements: ");
		n = scanner.nextInt();

		System.out.println("Enter the set in increasing order: ");
		for (int i = 1; i <= n; i++)
			S[i] = scanner.nextInt();
		System.out.println("Enter the max. subset value(d): ");
		d = scanner.nextInt();
		for (int i = 1; i <= n; i++)
			sum = sum + S[i];
		if (sum < d || S[1] > d)
			System.out.println("No Subset possible");
		else
			SumofSub(0, 0, sum);
		scanner.close();
	}

	static void SumofSub(int i, int weight, int total) {
		if (promising(i, weight, total) == true)
```

```
                if (weight == d) {
                        for (int j = 1; j <= i; j++) {
                                if (soln[j] == 1)
                                        System.out.print(S[j] + "  ");
                        }
                        System.out.println();
                }
                else {
                        soln[i + 1] = 1;
                        SumofSub(i + 1, weight + S[i + 1], total - S[i + 1]);
                        soln[i + 1] = 0;
                        SumofSub(i + 1, weight, total - S[i + 1]);
                }
        }

        static boolean promising(int i, int weight, int total) {
        return ((weight + total >= d) && (weight == d || weight + S[i + 1] <= d));
        }
}
```

Output

```
Enter number of elements:
5
Enter the set in increasing order:
2
3
4
5
6
Enter the max. subset value(d): 9

2  3  4
3  6
4  5
```

9. Implement “N-Queens Problem” using Backtracking

```
package com.JournalDev;
public class Main {
    static final int N = 4;

   // print the final solution matrix
    static void printSolution(int board[][])
    {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++)
                System.out.print(" " + board[i][j]
                        + " ");
            System.out.println();
        }
    }

    // function to check whether the position is safe or not
    static boolean isSafe(int board[][], int row, int col)
    {
        int i, j;
        for (i = 0; i < col; i++)
            if (board[row][i] == 1)
                return false;


        for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
            if (board[i][j] == 1)
                return false;

        for (i = row, j = col; j >= 0 && i < N; i++, j--)
            if (board[i][j] == 1)
                return false;

        return true;
    }

    // The function that solves the problem using backtracking
    public static boolean solveNQueen(int board[][], int col)
    {
        if (col >= N)
            return true;

        for (int i = 0; i < N; i++) {
            //if it is safe to place the queen at position i,col -> place it
            if (isSafe(board, i, col)) {
```

```
            board[i][col] = 1;

            if (solveNQueen(board, col + 1))
                return true;

            //backtrack if the above condition is false
            board[i][col] = 0;
        }
    }
    return false;
}

public static void main(String args[])
{
    int board[][] = { { 0, 0, 0, 0 },
            { 0, 0, 0, 0 },
            { 0, 0, 0, 0 },
            { 0, 0, 0, 0 } };

    if (!solveNQueen(board, 0)) {
        System.out.print("Solution does not exist");
        return;
    }

    printSolution(board);

}
}
```

Output :

```
0  0  1  0
1  0  0  0
0  0  0  1
 1  0  0
```