# Code Crunch v2.0

1. Given two strings word1 and word2, return *the minimum number of operations required to convert word1 to word2*.

You have the following three operations permitted on a word:

- Insert a character
- Delete a character
- Replace a character

**Example 1:**

**Input:** word1 = "horse", word2 = "ros"

**Output:** 3

**Explanation:**

horse -> rorse (replace 'h' with 'r')

rorse -> rose (remove 'r')

rose -> ros (remove 'e')

**Example 2:**

**Input:** word1 = "intention", word2 = "execution"

**Output:** 5

**Explanation:**

intention -> inention (remove 't')

inention -> enention (replace 'i' with 'e')

enention -> exention (replace 'n' with 'x')

exention -> exection (replace 'n' with 'c')

exection -> execution (insert 'u')

**Constraints:**

- 0 <= word1.length, word2.length <= 500

- word1 and word2 consist of lowercase English letters.

2. Write an algorithm to determine if a number n is happy.

A **happy number** is a number defined by the following process:

- Starting with any positive integer, replace the number by the sum of the squares of its digits.
- Repeat the process until the number equals 1 (where it will stay), or it **loops endlessly in a cycle** which does not include 1.
- Those numbers for which this process **ends in 1** are happy.

Return true *if* n *is a happy number, and* false *if not*.

**Example 1:**

**Input:** n = 19

**Output:** true

**Explanation:**

$1^2 + 9^2 = 82$

$8^2 + 2^2 = 68$

$6^2 + 8^2 = 100$

$1^2 + 0^2 + 0^2 = 1$

**Example 2:**

**Input:** n = 2

**Output:** false

**Constraints:**

- $1 <= n <= 2^{31} – 1$

**CODE**

```java
import java.util.*;

public class question2{
    public static int sumOfSquaresOfDigits(int n){
        int ans=0;
        while(n>0){
            int dig=n%10;
            ans+= dig*dig;
            n/=10;
        }
        return ans;
    }
    public static boolean question(int n){
        HashSet<Integer> hs=new HashSet<>();
        while(n!=1 && ! hs.contains(n)){
            hs.add(n);
            n=sumOfSquaresOfDigits(n);
        }
        return n==1;
    }

    public static void main(String[] args) {
        Scanner sc=new Scanner(System.in);
        int n=sc.nextInt();
        System.out.println(question(n));
    }
}
```

3. A linked list of length n is given such that each node contains an additional random pointer, which could point to any node in the list, or null.

Construct a deep copy of the list. The deep copy should consist of exactly n **brand new** nodes, where each new node has its value set to the value of its corresponding original node. Both the next and random pointer of the new nodes should point to new nodes in the copied list such that the pointers in the original list and copied list represent the same list state. **None of the pointers in the new list should point to nodes in the original list**.

For example, if there are two nodes X and Y in the original list, where X.random --> Y, then for the corresponding two nodes x and y in the copied list, x.random --> y.

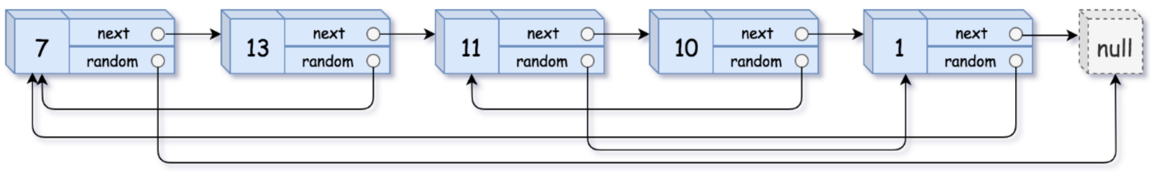
Return *the head of the copied linked list*.

The linked list is represented in the input/output as a list of n nodes. Each node is represented as a pair of [val, random_index] where:

- val: an integer representing Node.val
- random_index: the index of the node (range from 0 to n-1) that the random pointer points to, or null if it does not point to any node.

Your code will **only** be given the head of the original linked list.

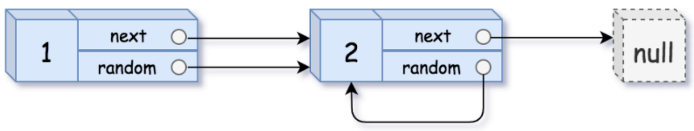
**Example 1:**



**Input:** head = [[7,null],[13,0],[11,4],[10,2],[1,0]]

**Output:** [[7,null],[13,0],[11,4],[10,2],[1,0]]

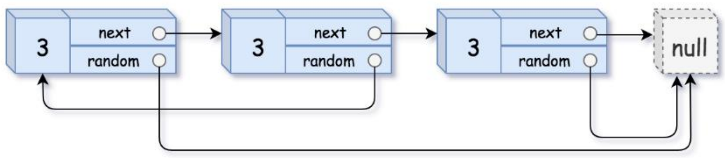
**Example 2:**



**Input:** head = [[1,1],[2,1]]

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

**Example 3:**



**Input:** head = [[3,null],[3,0],[3,null]]

**Output:** [[3,null],[3,0],[3,null]]

**CODE**

```
class Node{
    int val;
    Node next;
    int random_index;
    public Node(int val, Node next, int
random_index){
        this.val=val;
        this.next=next;
        this.random_index=random_index;
    }
}
public class question3 {
    public static Node question(Node head){
        Node dummy=head;
        Node ans=new Node(dummy.val, dummy.next,
dummy.random_index);
        Node ansHead=ans;
        dummy=dummy.next;
        while(dummy!=null){
            ans.val= dummy.val;
            ans.next=dummy.next;
            ans.random_index=dummy.random_index;
            ans=ans.next;
            dummy=dummy.next;
        }
        ans.next=null;
        return ansHead;
    }
}
```

4. According to Wikipedia's article: "The **Game of Life**, also known simply as **Life**, is a cellular automaton devised by the British mathematician John Horton Conway in 1970."

The board is made up of an m x n grid of cells, where each cell has an initial state: **live** (represented by a 1) or **dead** (represented by a 0). Each cell interacts with its eight neighbors (horizontal, vertical, diagonal) using the following four rules (taken from the above Wikipedia article):

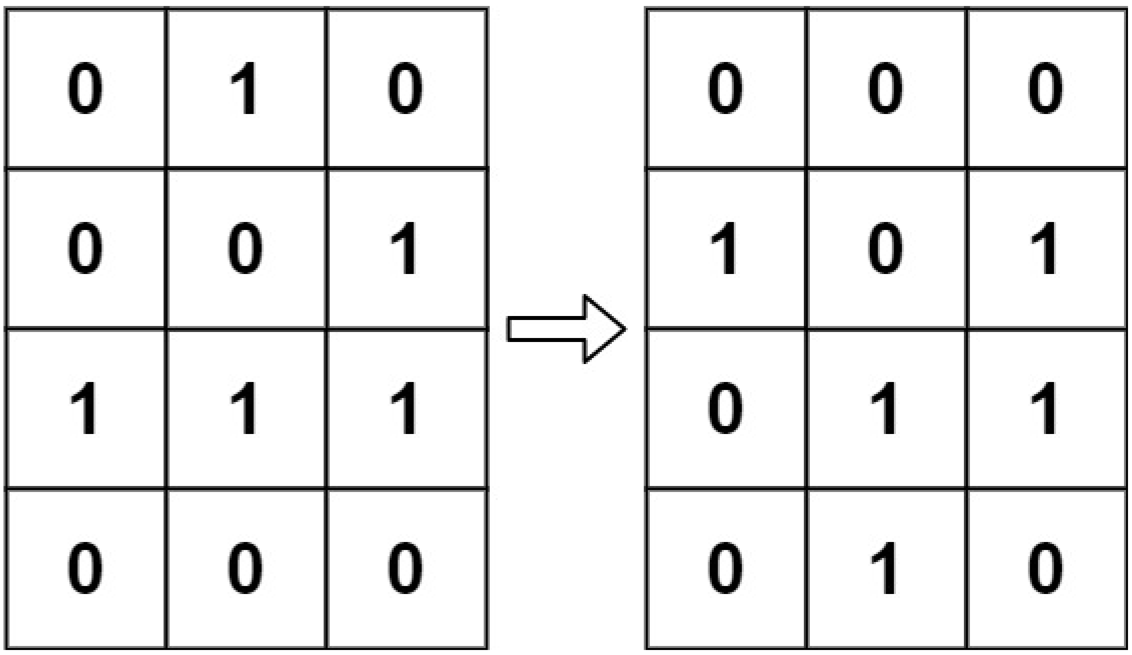
1. Any live cell with fewer than two live neighbors dies as if caused by under-population.
2. Any live cell with two or three live neighbors lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by over-population.
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

The next state of the board is determined by applying the above rules simultaneously to every cell in the current state of the m x n grid board. In this process, births and deaths occur **simultaneously**.

Given the current state of the board, **update** the board to reflect its next state.

**Note** that you do not need to return anything.

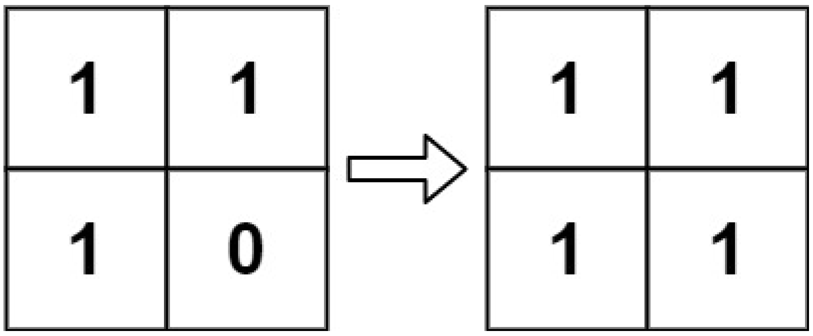
**Example 1:**



**Input:** board = [[0,1,0],[0,0,1],[1,1,1],[0,0,0]]

**Output:** [[0,0,0],[1,0,1],[0,1,1],[0,1,0]]

**Example 2:**



**Input:** board = [[1,1],[1,0]]

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

**Constraints:**

- m == board.length
- n == board[i].length
- 1 <= m, n <= 25
- board[i][j] is 0 or 1.

**CODE**

```
import java.util.*;
class Pair1{
    int i;
    int j;
    int val;
    public Pair1(int i, int j, int val){
        this.i=i;
        this.j=j;
        this.val=val;
    }
}
public class question4 {
    public static void question(int [][]grid){
        int []rowDir={-1,-1,-1,0,0,0,1,1,1};
        int []colDir={-1,0,1,-1,0,1,-1,0,1};
        int m=grid.length;
        int n=grid[0].length;
//          for (int i = 0; i < m; i++) {
//              for (int j = 0; j < n; j++) {
//                  int live=0;
//                  for (int k = 0; k < 8; k++) {
//                      int row=i+rowDir[k];
//                      int col=j+colDir[k];
//                      if(row<m && row>=0 && col>=0
&& col<n){
//                          if(grid[row][col]==1){
//                              live++;
//                          }
//                      }
//                  }
//                  if(grid[i][j]==1){
//                      if(live<2 || live>3)
grid[i][j]=0;
//                  } else{
//                      if(live==3){
//                          grid[i][j]=1;
//                      }
//                  }
//              }
//          }

        Queue<Pair1> q=new LinkedList<>();
        q.add(new Pair1(0,0, grid[0][0]));
        while(!q.isEmpty()){
            Pair1 p=q.poll();
            int i=p.i;
            int j=p.j;
            int status=p.val;
            HashSet<Integer[]> hs =new HashSet<>();
            int live=0;
            for (int k = 0; k < 8; k++) {
                int row=i+rowDir[k];
                int col=j+colDir[k];
                if(row<m && row>=0 && col>=0 &&
col<n){
                    if(grid[row][col]==1){
                        live++;
                    }
                    q.add(new
Pair1(row,col,grid[row][col]));
                }

            }
            if(status==1){
                if(live<2 || live>3) grid[i][j]=0;
            } else{
                if(live==3){
                    grid[i][j]=1;
                }
            }
        }
    }

    public static void main(String[] args) {
        int
[][]grid={{0,1,0},{0,0,1},{1,1,1},{0,0,0}};
        question(grid);

System.out.println(Arrays.deepToString(grid));
    }
}
```