PROBLEM OF THE MONTH

November, 2018

MATHEMATICS

5 points:

A Sigma team consisting of 10 campers forms a team to participate in the Sigma Tournament. Campers have to select 6 people out of their number to solve timed problems at the table during the tournament, so that other 4 campers are sitting on a couch solving written problems. One of the campers sitting at the table should be selected as team captain. In how many different ways one can divide 10 campers into 1 captain + 5 campers at the table + 4 on the couch?

Hint: In how many ways can you select the captain from the team of 10? In how many ways can you select 5 campers out of the remaining 9 campers?

Answer: 1260 or (10 choose 1)*(9 choose 5) or (10 choose 6)*(6 choose 1) → all of these are correct

Solution: We can choose the captain first (10 choose 1) and then choose 5 people to go to the table out of the remaining 9 (9 choose 5) and multiply.
We can also just choose 6 to go to the table first (10 choose 6) and then one of those to be captain (6 choose 1).
Both of these yield the correct number of ways to choose the campers, and the exact number is 1260.

10 points:

There are three campers: one from team alpha, one from team beta and one from team gamma. Every year at Sigma, two of the three campers are randomly chosen and they switch teams.
a) What is the chance that after 2017 SigmaCamps, they will all be on their original teams?
b) After 2018 camps?

Hint: Let us denote the original placement of students as (1, 2, 3), meaning that Student 1 is in team alpha, Student 2 is in team beta, and Student 3 is in gamma. After the first swap you can

end up in (1, 3, 2), (2, 1, 3), and (3, 2, 1) with equal probability. Think about what happens with these probabilities during the second step, and so on.

Answer: a) 0 b) 1/3

Solution:

There are six possible placements of students, and they can be divided in two groups which we will call “even” and “odd”:
Even: (1,2,3), (2, 3, 1), (3,1,2)
Odd: (1,3,2), (2,1,3), (3,2,1)
It is easy to check that each swap moves you from an even position to an odd one and vice versa, which explains the names: we end up in an odd position after an odd numbers of swaps, and in an even position after an even number of swaps.
We know that we start in the position (1,2,3), and after one swap, we end up in an odd position.
Part A:

If the number of swaps is odd (1 swap, 3 swaps… 2017 swaps), the chances of being in the original state is 0.

This is because from any even position, one swap will yield an odd position. So, since we start in an even position, one swap will make an odd position and then another swap will go back to even, and then another is back to odd. So, one can never end up in an even position, let alone the original position, by having an odd number of swaps.

Part B:

If the number of swaps is even (2018 swaps), the chances of being in the original state is ⅓.

This is because at 2017 swaps, we are in an odd position, and if we swap any two campers in an odd position then we get back to an even position. There are three possible swaps, and each of them gives one of the three even positions, so the chances of getting back to the original position is ⅓ because it is one of the three even positions.

PHYSICS

5 points:

An airgun bullet is d=4.4 mm in diameter and weights m=0.5 g. It is propelled by a compressed air. Find the speed of the bullet as it exits the gun, if the length of the barrel is L=0.5 m, and excess air pressure behind the bullet (with respect to the atmospheric pressure outside) stays nearly constant, around P=300 kPa.

Hint: Use the fact that work done by the compressed air all goes to changing the kinetic energy of the bullet.

Answer: v= $d\sqrt{\frac{\pi LP}{2m}} = 95.5 m/s$

Solution: Work done by the compressed air $FL = \pi d^2 LP/4$, should be equal to the kinetic energy of the bullet, $mv^2/2$, therefore

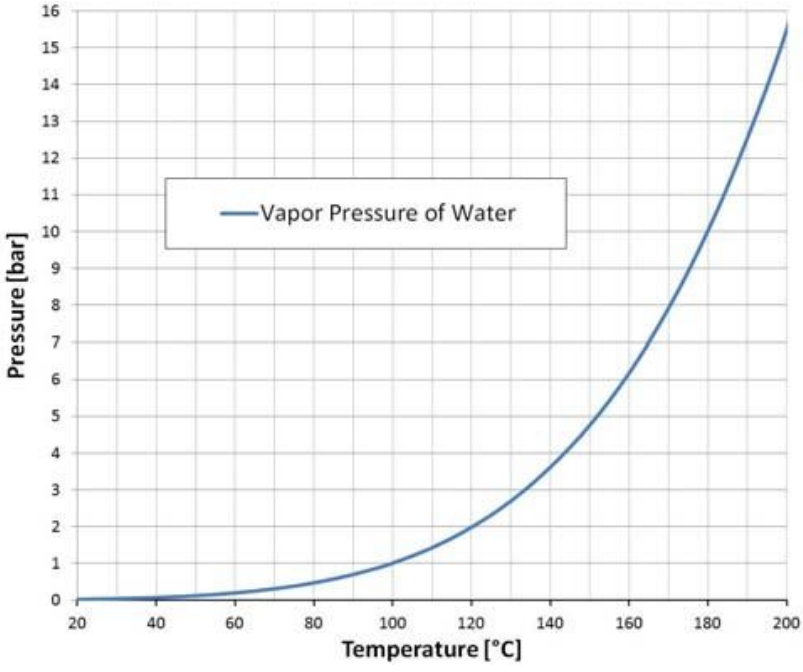
v= $d\sqrt{\frac{\pi LP}{2m}} = 95.5 m/s$

10 points:

A geyser consists of an underground cavity in which water is heated geothermally, and a channel that connects the cavity to the surface. An eruption occurs when the water in the cavity starts boiling. It stops when the temperature of the water drops below the boiling temperature. Note that the temperatures at which the boiling starts and stops are different because the channel is originally fully filled with water, which leads and additional hydrostatic pressure in the cavity. The channel is completely vertical and has a length of 90m.

Estimate the fraction of the total mass of water in the cavity which is vaporized during a single eruption. Specific heat of water is ***C=4.2kJ/kg°C,*** the latent heat of its vaporization is ***L=2230 kJ/kg***.

The figure on the right shows Pressure vs. water boiling temperature. Unit of pressure in the plot is 1 bar =10^5 Pa , which is close to 1 atmosphere (that's why T=100°C corresponds to P=1 bar).



Hint: 1) find the pressure in the cavity in before the eruption, and use the plot to determine the boiling temperature. 2) As the water gets converted from liquid to steam, it consumes heat and cools remaining water in the cavity.

Answer: 15%

Solution: Before eruption, hydrostatic pressure in the cavity is $1bar + \rho gh = 10\ bar$, which means that the boiling starts at $180^0 C$. During eruption, the water is expelled from the channel, and pressure drops to the atmospheric one. So, the eruption will stop at $100^0 C$. The water is cooled down due to the heat absorbed during evaporation:

$$CM\left(180^0 C - 100^0 C\right) = L\Delta M$$

Here M is the total mass of water in the cavity, and ΔM is the mass of evaporated one. Thus,

$$\Delta M/M = C\left(180^0 C - 100^0 C\right)/L = 4.2 \cdot 80/2230 = 0.15$$

(In reality, water of course gets replenished in the cavity).

CHEMISTRY

5 points:

"*We had two bags of Arabica coffee, seventy-five ounces of sodium hydroxide pellets, five kilograms of high purity acetic acid, a saltshaker half-full of aspirin, and a whole galaxy of multi-colored pH papers, rubber balloons, strings etc... Also, a quart of concentrated sulfuric acid, a quart of acetone, a case of Poland Spring water, a pint of raw ether, and two dozen grams of isoamyl alcohol. Not that we needed all that for our graphomaniac exercises, but once you get locked into a serious chemicals collection*, *the tendency is to push it as far as you can. The only thing that really worried me was the ether. There is nothing in the world more helpless and irresponsible and depraved than a man in the depths of an ether binge, and I knew we'd get into that rotten stuff pretty soon.*"

Using the stuff described in this quote, can you prepare a banana smell? Which items listed here are needed for that, and how will you do it?

Hint:

Banana smell is a compound that is an ester of acetic acid. What ester it is, and how can it be prepared? Explain.

Solution:

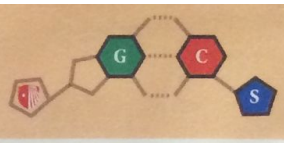
A compound with banana smell is isoamyl acetate, and ester formed by acetic acid and isoamyl alcohol. A standard way to make esters is to heat a mixture of a carboxylic acid alcohol in the presence of a strong acid as a catalyst. A byproduct of this reaction is water, so it is desirable to remove it from the reaction mixture. Concentrated sulfuric acid or phosphoric acid can bind to water (they bind water molecules pretty tightly, although it is not a real covalent bond), so they serve both as a catalyst and water scavenger. After a mixture of acetic acid, isoamyl alcohol and sulfuric acid has been heated for some time, it can be poured into a dilute solution of sodium hydroxide, which neutralises acids (both sulfuric acid and residual amounts of acetic acid), and isoamyl acetate, which is not mixable with water, forms a top layer and can be collected.

10 points:

Alice, a college faculty, came to her lab and found Bob, her technician, reading a Molecular Biology textbook. “What are you doing, Bob”, she asked. “Hi, Alice, I’ve just spoke with one my friend, he told me a lot about recent discoveries in molecular biology. It is so interesting! Ribozymes, the RNA world hypothesis, genome editing using a CRISPR-Cas9 system - all of that sounds like a science fiction. Tomorrow, I meet my friend again, and I have a lot of questions to ask.”

“Who is your friend, Bob?”, Alice asked.

“Oh, he is a really smart guy!”, - Bob exclaimed. “He is a graduate student at Stony Brook university, and he is a member of the Graduate Chemical Society. By the way, he showed me their logo, I like it very much. It looks like some formula, it is so beautiful! Look, Alice”. And Bob showed this photo to Alice:



“Hmm,”- Alice replied, - “Those guys may be good chemists, but I am pretty sure there are no experts in nucleic acid chemistry or biology in this society. I doubt your friend can explain you anything about ribozymes or CRISPR”.

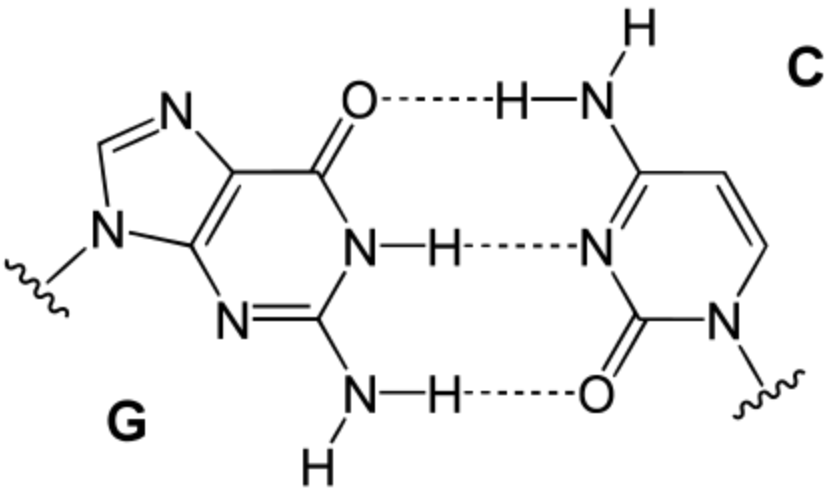
Please, tell why Alice made this conclusion?

Hint:

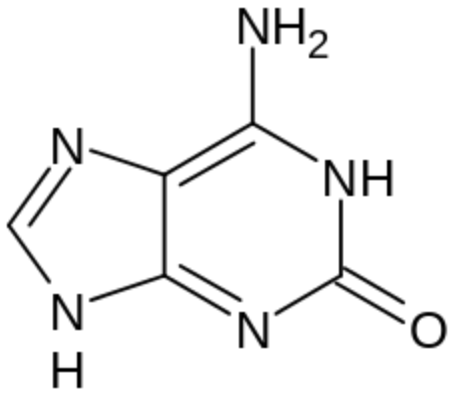
It seems that the logo is supposed to be an imitation of the Watson-Crick base pair (more concretely, a G-C pair, because it is a **G**rad **C**hemistry society). However, the “Watson-Crick” pair on the logo is shown incorrectly: there are *two* errors there. What these two errors are?

Solution:

The two errors are as follows. First, the purine base is attached to the sugar at a wrong position. To see a co



The second error is less obvious. The guanine base is flipped on the logo, but if a guanine is flipped, it cannot form three hydrogen bonds with cytosine: as you can see from the above picture, the NH_2 group of a flipped guanine makes a contact with the NH_2 group of cytosine, and guanine’s oxygen contacts with cytosine’s oxygen. That means, for a flipped purine base to form three hydrogen bonds with cytosine, this base should be not guanine, but something else. This base is called “isoguanine” (i-G), and it has a formula:



(nothe that NH_2 and oxygen are swapped).
In other words, the letters on the logo should be “IG-C-S”, and that is probably an acronym of some “**IG**norant **C**hemists **S**ociety” :)

BIOLOGY

5 points:

A genome of some organism contains the fragment with the following sequence:
5' TTATCCATGTGGCATTAGATGTAAGG 3'
3' AATAGGTACACCGTAATCTACATTCC 5'
How many different proteins can a ribosome synthesize from the DNA that contains this segment? In your answer. assume that the start position for the synthesis of each peptide is situated in this segment. Write all possible amino acid sequences of these peptides.

HINT

Note that DNA is double-stranded, and ORF can be present in both directions.

ANSWER

Double stranded DNA:
5' TTATCCATGTGGCATTAGATGTAAGG 3'
3' AATAGGTACACCGTAATCTACATTCC 5'
Two ORFs:
>
Direct strand: 5' TTATCC ATG TGG CAT TAG ATGTAAGG

Complementary strand: 3' AAT AGG TAC ACC GTA ATCTACATTCC
<
ATG TAA in the direct strand is not an ORF

10 points.

One of the color pattern in rats is called "hooded" - animals have dark fur on heads and top of the back, but a belly and legs are white. It has been determined that rats with hooded color pattern have an 1 kb insert in the intron of a *Kit* gene.

An intron is a DNA segment inside some gene. Introns encode no protein sequence; after RNA is synthesized from the DNA, a special cellular machinery cuts the RNA at the beginning and the

end of the intron segment, removes it, and re-connects the ends of the cut RNA back; only after that procedure (it is called “splicing”) the RNA is ready for translation.
The *Kit* gene encodes some receptor for a protein which regulates cell division and migration, including migration of melanocyte precursors. We know that *Kit* is important for cell migration during early development, and the introduction of the 1 kb insert into its intron has a direct relation to the hooded color pattern in rats.

Please suggest explanations of how inserting inactive sequence in an intron of *Kit* can cause such dramatic change in phenotype.

HINT Although intron is not coding for amino acids, it can affect what happens with DNA or during gene transcription. Consider how intron can change chromatin structure or which important sequences it might introduce between coding exons.

ANSWERS.

- Intron is a non-coding sequence which separates coding parts of a gene, exons. Introns are cut-out from RNA after gene is transcribed.
- Since Kit encodes regulator of cell migration, the hooded pattern may be a result of an insufficient migration of melanocytes during development of a rat.
- Since intron insert does not change sequence of mRNA and protein, we can hypothesize that this insert might change dynamics of production of mRNA. Longer intron would increase transcription time, leading to slower production of protein and making fewer receptors. If this happens in early developmental stages, small delays in cell migration may cause large effects on body features.
- Inserted sequence may have signaling sequences which dramatically change transcription process. For example, it may be enriched in CG dinucleotides and initiate DNA methylation, leading to less accessibility of the DNA and lower transcription rate.
- Inserted sequence may introduce new splice site, creating incorrect mRNA and defective protein.

COMPUTER SCIENCE

- You can write and compile your code here: http://www.tutorialspoint.com/codingground.htm
- Your program should be written in Java or Python
- No GUI should be used in your program: eg., easygui in Python. All problems in POM require only text input and output. GUI usage complicates solution validation, for which we are also using *codingground* site. Solutions with GUI will have points deducted or won't receive any points at all.
- Please make sure that the code compiles and runs on http://www.tutorialspoint.com/codingground.htm before submitting it.
- Any input data specified in the problem should be supplied as user input, not hard-coded into the text of the program.
- Submit the problem in a plain text file, such as .txt, .dat, etc. **No .pdf, .doc, .docx, etc!**

This month we are visiting Sigma2D, two-dimensional Sigma Universe. In Sigma2D, in one of the remote areas, there is a square minefield. The minefield has the shape of a grid with a side of 10, and therefore it has 100 cells in total. Each of the 100 cells can contain one mine or be empty. The minefield has markings on it: there is a number on each cell indicating how many mines there are in the 8 cells (or less, if the cell is at the edge of the minefield) immediately next to it.

5 points:

Write a program that takes on input a 10-by-10 array of 1's and 0's (1 stands for a mine, and 0 for an empty space) and returns an array of the same size, but containing numbers for how many mines there are at each cell's eight (or less) immediate neighbors. The program should terminate with a complaint if the input contains anything other than mines, separators and empty spaces (1s and 0s).

For example, for a mine array of:
1, 1, 1, 1, 1, 1, 1, 0, 1, 0
1, 0, 0, 0, 1, 1, 0, 0, 0, 1
1, 0, 1, 1, 0, 1, 0, 0, 0, 0
1, 1, 0, 1, 1, 0, 0, 0, 1, 1
1, 0, 1, 0, 1, 1, 1, 1, 0, 1
1, 1, 1, 0, 1, 1, 1, 1, 0, 1
0, 1, 0, 1, 1, 1, 1, 0, 1, 1

0, 0, 0, 0, 1, 0, 1, 0, 0, 0
1, 1, 1, 1, 0, 1, 1, 0, 0, 1
1, 0, 0, 1, 0, 0, 1, 1, 0, 0

the program should return the following numbers representing the number of mines around each cell:
2, 3, 2, 3, 4, 4, 2, 2, 1, 2
3, 6, 5, 6, 6, 5, 4, 2, 2, 1
3, 5, 3, 4, 6, 3, 2, 1, 3, 3
3, 5, 5, 5, 5, 5, 4, 3, 3, 2
4, 7, 4, 6, 5, 6, 5, 4, 6, 3
3, 5, 4, 6, 6, 8, 7, 5, 6, 3
3, 3, 4, 4, 5, 7, 5, 5, 3, 2
3, 4, 5, 5, 5, 7, 4, 4, 3, 3
2, 3, 3, 3, 4, 4, 4, 4, 2, 0
2, 4, 4, 2, 3, 3, 3, 2, 2, 1

Hint:

It is convenient for each cell to iterate through all cells with coordinates ranging from the current coordinate-1 to the current coordinate+1. In addition, a helper function for determining whether a particular pair of integers is a legitimate coordinate on the mine field might prove useful.

Solution:

Python:

```
import re

n = 10 # int(input("enter number of rows: ").strip())
f = []
for i in range(n):
  line = input("enter %dth row: " % (i+1))
  xs = list(map(str.strip, line.split(',')))
  if len(xs) != n:
    raise Exception("number of columns must be equal to the number of rows")
  for j in range(len(xs)):
    if not re.match(r"[01,\s]", xs[j]):
      raise Exception("invalid field symbol '%s' at position %d" % (xs[j], j))
  f.append(list(map(int, xs)))

r = []
for i in range(n):
  r.append([])
  for j in range(n):
    c = 0
    for di in [-1, 0, 1]: # search in the immediate vicinity
      for dj in [-1, 0, 1]:
        if di == 0 and dj == 0: continue # don't count the center
        if 0 <= i+di < n and 0 <= j+dj < n:
          c += f[i+di][j+dj]
```

```
    r[i].append(c)
print(r)
print("end.")
```

Java:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Arrays;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.util.stream.Stream;

public class MineSweeper5 {
  private int[][] f;
  private int n;
  private int[][] r;
  private int[] vicinity = {-1, 0, 1};
  private String validSymbols = "[01,\\s]+";

  private void input() throws IOException {
    n = 10;
    f = new int[n][n];
    r = new int[n][n];
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    Pattern pattern = Pattern.compile(validSymbols);
    for(int i=0; i<n; i++) {
      System.out.printf("enter %dth row: ", i+1);
      String line = br.readLine();
      Matcher matcher = pattern.matcher(line);
      if(!matcher.matches())
        throw new IllegalArgumentException("invalid input string");
      f[i] = Stream.of(line.trim().split("\\s*,\\s*")).mapToInt(Integer::parseInt).toArray();
      if(f[i].length != n)
         throw new IllegalArgumentException("number of columns must be equal to the number of
rows");
    }
  }

  private void test() throws IOException {
    input();

    for(int i=0; i<n; i++) {
      for(int j=0; j<n; j++) {
        int c = 0;
        for(int di : vicinity) {
          for(int dj : vicinity) {
            if(di == 0 && dj == 0)
              continue; // don't count the center
            if(i+di >= 0 && i+di < n && j+dj >= 0 && j+dj < n)
              c += f[i+di][j+dj];
          }
        }
        r[i][j] = c;
      }
    }
    System.out.println(Arrays.deepToString(r));
```

```
  }

  public static void main(String[] args) throws IOException {
    MineSweeper5 mineSweeper5 = new MineSweeper5();
    mineSweeper5.test();
    System.out.println("end.");
  }
}
```

10 points:

For this problem, your program should perform the "inverse" of what the previous problem's solution did. The input to your program should be a 10-by-10 array containing number of mines in each cell's immediate neighborhood of cells. The program should calculate and return the corresponding array of 1’s and 0’s, depending on whether there is a mine in the current location or not. This array, when given as an input to the 5 point problem above, would have produced the input array of this problem. As an example, for an input of the array of integers shown in the 5 point problem as a sample output, your program should return the array presented in the 5 point problem as a sample input.

Hint:

When there are "uncertain" cells around the current cell, look for the situations when the number of definitively placed mines around the current cell already equal to this cell's value (and therefore all "uncertain" cells definitely do not have mines), or, on opposite, the number of definitively placed mines plus the number of "uncertain" cells equal to the cell's value (and then all "uncertain" cells definitely have mines). Keep doing it iteratively. When you can't make any more progress, start "experimenting" with placing mines in the next "uncertain"cell.

Solution:

Python-3:

```
import itertools as it

# some generic, useful functions

def list_flatten(list_of_list):
    '''
    flattens a list of lists
    '''
    return list(it.chain(*list_of_list))

def add_lists_element_wise(list_1, list_2):
    '''
    element-wise addition of 2 lists of equal length
    '''
    if len(list_1)==len(list_2):
        return [sum(pair) for pair in zip(list_1, list_2)]
    else:
```

```
        print('unequal lengths!')

def pair2string(pair):
    '''
    pair of integers to a string
    '''
    return '-'.join([str(integer) for integer in pair])

def string2pair(string):
    '''
    string to a pair of integers
    '''
    return [int(integer) for integer in string.split('-')]

# some minesweeper-specific functions

def get_nearest_neighbors(location, n):
    '''
    obtains the list of locations that are nearest neighbors of a given location,
    and given the bounding box size n
    '''

    nearest_neighbors = [add_lists_element_wise(delta, location) for delta in deltas]
    # filtering to exclude the neighbors outside the bounding box
    nearest_neighbors = list(it.filterfalse(
        lambda location: max(location)>=n or min(location)<0,
        nearest_neighbors
    ))

    return nearest_neighbors

def inaccessibleQ(proximityMap, locations, location_i):
    '''
    Checks whether any locations are "inaccessible", meaning there is no way of modifying their
proximity map
    by changing the minesVector values for locations that have not been visited yet.
    Takes union of nearest_neighbors of all locations not visited yet and takes its complement
    in the set of all locations. If the complement isn't empty, then these are the inaccessible
locations.
    '''
    nearest_neighbors = set([
        pair2string(pair) for pair in
        list_flatten([
            get_nearest_neighbors(location, n) for location in
            locations[location_i:]
        ])
    ])
    locations_inaccessible = [
                    string2pair(string)  for  string  in  set([pair2string(pair)  for  pair  in
locations]).difference(set(nearest_neighbors))
    ]
    if len(locations_inaccessible)==0:
        return False
    else:
                        return  max([proximityMap[location[0]][location[1]]  for  location  in
locations_inaccessible])>0
```

```
def negativeQ(proximityMap):
    '''
    simply determines if any of the entries are below 0
    '''
    return min(list_flatten(proximityMap))<0

def minesVector2minesArray(locations, minesVector):
    '''
    this turns the solution (a vector of mines inferred) into an array,
    using the order of locations visited while solving
    '''
    if len(locations)!=len(minesVector):
        print('unequal lengths of locations and minesVector!')
    else:
        minesSolution = [n*[0] for i in range(n)]
        for location, mine in zip(locations, state.minesVector):
            minesSolution[location[0]][location[1]] = mine
        return minesSolution

class State(object):

    def __init__(self, locations, proximityMap, printing=False):
        self.locations = locations
        self.proximityMap = proximityMap
        self.minesVector = []
        self.exhausted = []
        self.steps_back = 0
        # print(self.proximityMap)

    def advance(self):
        '''
        Going forward, we automatically assign a "mine" or 1 to the cell.
        '''
        # print(get_nearest_neighbors(self.locations[len(self.minesVector)], n))
        self.minesVector.append(1)
        self.exhausted.append(0)
        for location in get_nearest_neighbors(self.locations[len(self.minesVector)-1], n):
            self.proximityMap[location[0]][location[1]] -= 1
        # print(self.proximityMap)

    def reverse(self):
        '''
        Reversal looks for the last currently "mine" (1) location to flip to "no mine" (0) and
        rolls back all the changes to proximityMap going back to this location
        The indexing is admittedly somewhat tedious...
        '''
        self.steps_back = 1
        # print('exhausted:  ', self.exhausted)
        while self.exhausted[-self.steps_back] == 1:
            # print(self.steps_back
            self.steps_back += 1
        # print('steps_back:', self.steps_back)
        for i in range(self.steps_back):
            if self.minesVector[-(i+1)] == 1:
                                                                    for    location    in
get_nearest_neighbors(self.locations[len(self.minesVector)-(i+1)], n):
                    # print(location)
```

```
                    self.proximityMap[location[0]][location[1]] += 1
        if self.steps_back > 1:
            self.minesVector = self.minesVector[:-(self.steps_back-1)]
            self.exhausted = self.exhausted[:-(self.steps_back-1)]
        self.minesVector[-1] = 0
        self.exhausted[-1] = 1
        # print(self.proximityMap)

'''
all possible changes to neighboring cells
'''
deltas = [(-1,-1),(-1,0),(-1,1),(0,-1),(0,1),(1,-1),(1,0),(1,1)]
print(deltas)

'''
Locations is the order in which the cells are to be visited.
They are chosen to find issues/inconsistencies along the solution as quickly as possible
by placing the next cell to visit to have the most neighbors already visited.
The routine is agnostic to choice of path of locations to visit, as long as they cover the
entire square
'''

locations = [(0, 0), (1, 0), (0, 1), (1, 1), (2, 1), (2, 0), (1, 2), (2, 2), (0, 2), (3, 1),
(3, 2), (1, 3), (2, 3), (3, 3), (2, 4), (3, 0), (3, 4), (4, 2), (4, 3), (4, 1), (5, 2), (5, 3),
(4, 4), (5, 4), (3, 5), (4, 5), (4, 0), (5, 1), (5, 0), (6, 3), (6, 2), (6, 1), (2, 5), (1, 4),
(0, 3), (0, 4), (1, 5), (2, 6), (3, 6), (4, 6), (5, 5), (6, 4), (7, 3), (7, 2), (3, 7), (0, 5),
(1, 6), (2, 7), (7, 1), (6, 0), (8, 2), (7, 0), (8, 1), (0, 6), (1, 7), (6, 5), (5, 6), (4, 7),
(7, 4), (8, 3), (9, 2), (7, 5), (8, 4), (9, 3), (6, 6), (5, 7), (8, 5), (9, 4), (7, 6), (6, 7),
(7, 7), (8, 6), (9, 5), (0, 7), (4, 8), (5, 8), (6, 8), (3, 8), (2, 8), (1, 8), (5, 9), (4, 9),
(3, 9), (2, 9), (0, 8), (1, 9), (9, 6), (8, 7), (7, 8), (6, 9), (8, 8), (9, 7), (7, 9), (9, 1),
(8, 0), (9, 8), (8, 9), (0, 9), (9, 0), (9, 9)]

'''
proximity map of how many cells are in the 8 neighboring cells in total for each cell
this is the input to the routine
proximity = [
    [1, 1, 4, 2, 3, 2, 1, 3, 3, 3],
    [2, 4, 5, 4, 4, 3, 4, 6, 6, 5],
    [1, 2, 4, 6, 5, 5, 3, 4, 6, 5],
    [2, 3, 5, 4, 5, 5, 6, 6, 6, 4],
    [2, 2, 5, 4, 7, 5, 4, 3, 6, 4],
    [3, 3, 4, 4, 5, 6, 7, 5, 5, 3],
    [2, 2, 3, 4, 5, 6, 5, 4, 6, 3],
    [1, 1, 2, 4, 6, 7, 7, 4, 3, 2],
    [0, 1, 2, 3, 5, 5, 4, 3, 3, 2],
    [0, 1, 1, 3, 4, 3, 3, 1, 1, 0]
]
'''
n = 10
proximity = []
for i in range(n):
     input_row = input("enter the row number %i as %i elements separated by comma: " % (i+1,
n)).rstrip("\r\n")
    try:
        input_row_items = [int(item.strip()) for item in input_row.split(',')]
        if len(input_row_items) != n:
            print("number of elements in", input_row_items, "is not %i" % (n))
```

```
            exit(1)
        proximity.append(input_row_items)
    except ValueError:
        print(input_row, "has a non-integer element")
        exit(1)
print("proximity map was entered as:\n", proximity)
'''
'''

state = State(locations, [list(proximity_row) for proximity_row in proximity])

t = 0
impossible = 0

'''
We go on until either we reduce the proximityMap to all 0s (all mines are taken care of)
or we find that it is impossible to do so.
'''

while max(list_flatten(state.proximityMap))>0 and not(impossible):
    print(t)
    state.advance()
    # print('minesVector:', state.minesVector)
         while(negativeQ(state.proximityMap)  or  inaccessibleQ(state.proximityMap,  locations,
len(state.minesVector))):
        '''
        2 conditions for reversal/backtracking:
        either there is a negative entries in the proximityMap (we put down more mines that the
problem asked for)
        or there is a location on the proximityMap that has some mines left to take care of,
        but nothing can be done about them based on the locations to be visited.
        '''
        try:
            state.reverse()
        except IndexError:
            impossible = 1
            break
        # print('minesVector:', state.minesVector)
    t += 1
print("final proximity map:", state.proximityMap)
print("impossible =", impossible)
print("solution:\n", minesVector2minesArray(locations, state.minesVector))
```