# Assignment n°3: Pick You Battles

## Problem A: The Way Out

This problem is to find the shortest path to the end of the maze, represented by a matrix of M x N.

The matrix also contains a character '2', representing the maze exit.

I tried to solve this problem because I already have done it in the school I come from.

To solve this problem, I would use a Dijkstra algorithm on all directions (North, East, South, West).

Each time we're checking a direction, we're storing it in a list. On the next iteration, we're processing once again from the already checked positions.

You can check the code I have already done by clicking this link.

The way I would design it would be the same as the repository given just above. The time complexity of this code is 0(n^2), cause of the double loop used to compute the Dijkstra.

## Problem B: SEND MORE MONEY!

This problem is a casual cryptarithmetic puzzle, where each letter represent a digit. Each digit must make an equation true, following this:

```
String 1 + String 2 = String 3
```

In order to solve this problem, I would use the permutations object with Python that will help me.

Firstly, we must get each letter in the phrase. If we have a higher number of letter than digits, we must have an error.

Then, we will get all possible permutations using x of the 10 digits, where x is the number of letters in the equation. From there, we can get one case:

- If the result of the equation is longer than both the first operands, the first digit of the result is 1. Though, we can exclude every permutation not having the number 1 at the right index.

Then, we're computing each possible value for our letters. When it's done for each word, we can check if the equation is right.

Finally, if the equation is true, we can print the result. If not, we can carry on the next permutation.

```
from typing import List
from itertools import permutations
from sys import argv


class Solver:
    def __init__(self, equationToSolve: str) -> None:
        self._equation = equationToSolve
        self._wordList = []
        self._letterList = []

    def getLettersFromEquation(self) -> bool:
        self._wordList = self._equation.split()

        for word in self._wordList:
            if not word.isalpha():
                ## Remove the operands or elements not having only
letters.
                self._wordList.remove(word)

        ## We're creating a set in order to have only unique values.
        ## Then we're converting the set into a list in order to have
access to indices.
        self._letterList = list(set(''.join(self._wordList)))
        if len(self._letterList) > 10:
            return False
        return True

    def transformValue(self, values: List[int]) -> int:
        """
        Computes the list of integer for each value of a word, into an
integer, using power of ten.

        :param values: List of value for each word
        :return: Computed value for the word
        """
        tenPowersList = [10 ** i for i in range(len(values) - 1, -1, -1)]
        value = 0

        for i in range(len(values)):
            value += values[i] * tenPowersList[i]
        return value

    def findCorrectWordValues(self) -> int:
        it = 0
        resultBiggerLength = False

        def getAllPossibleAssociation():
            return list(permutations([0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
len(self._letterList)))

        associationList = getAllPossibleAssociation()
        if len(self._wordList[2]) > len(self._wordList[0]) and
```

```
len(self._wordList[2]) > len(self._wordList[0]):
            resultBiggerLength = True

        for association in associationList:

            ## Checks if the result is bigger than the both operands. If
true, we're checking
            ## if the element in the 'x' position of the permutation is
not 1, we continue. Because that
            ## permutation will not be the right one.
            if resultBiggerLength and
association[self._letterList.index(self._wordList[2][0])] != 1:
                continue

            ## Creates all values for each letter of each word. When
created, we loop over those
            ## elements, and we're getting the indices of all letters in
our main list and
            ## we're getting its value in the permutation.
            elementValues = [[0] * len(self._wordList[i]) for i in
range(len(self._wordList))]
            for it in range(len(self._letterList)):
                for idx, wordValues in enumerate(elementValues):
                    letterIndices = [_k for _k, _j in
enumerate(self._wordList[idx]) if _j == self._letterList[it]]
                    for index in letterIndices:
                        wordValues[index] = association[it]

            ## When done, we're computing the value for each word. Then,
we're checking if the 2 firsts
            ## are equal to the last one. Then, we're creating the
equation with the transformed words.
            resultList = [self.transformValue(values) for values in
elementValues]
            if sum(resultList[:-1]) == resultList[-1]:
                result = ''
                for idx, value in enumerate(resultList[:-1]):
                    result += f'{value}{" + " if idx !=
len(resultList[:-1]) - 1 else " = "}'
                result += f"{resultList[-1]}"
                print(result)
        return 0

    def run(self) -> int:
        if not self.getLettersFromEquation():
            return 1
        return self.findCorrectWordValues()


def main() -> int:
    if len(argv) != 2:
        return 1
    equationSolver = Solver(argv[1])
    return equationSolver.run()
```

```
if __name__ == "__main__":
    exit(main())
```

All the code above is tested with this:

```
from sources.main import Solver


class TestSolver:
    def test_solverCreation(self):
        solver = Solver('SEND + MORE = MONEY')
        assert solver.equation == 'SEND + MORE = MONEY'
        assert solver.wordList == []
        assert solver.letterList == []

    def test_getLettersFromEq(self):
        solver = Solver('SEND + MORE = MONEY')
        assert solver.getLettersFromEquation() == True
        assert solver.equation == 'SEND + MORE = MONEY'
        assert solver.wordList == ['SEND', 'MORE', 'MONEY']
        assert set(solver.letterList) == set(['S', 'E', 'N', 'D', 'M',
'O', 'R', 'Y'])

    def test_findCorrectEquation(self, capsys):
        solver = Solver('SEND + MORE = MONEY')
        assert solver.run() == 0
        stdout = capsys.readouterr()[0]
        assert stdout == "9567 + 1085 = 10652\n"

    def test_runWithTooMuchLetters(self):
        solver = Solver('ALED + PEUR = SIGNAL')
        assert solver.run() == 1

    def test_getTooManyLetters(self):
        solver = Solver('ALED + PEUR = SIGNAL')
        assert solver.getLettersFromEquation() == False

    def test_valueTransform(self):
        _ = Solver('SEND + MORE = MONEY')
        assert _.transformValue([9, 0, 5, 4]) == 9054
```

The code I wrote here has 3 loops inside one loop. Therefore, the complexity would be O(n^4) because I wanted to make it generic for any number of operands.

By replacing those lines...:

```
elementValues = [[0] * len(self._wordList[i]) for i in
range(len(self._wordList))]
for it in range(len(self._letterList)):
    for idx, wordValues in enumerate(elementValues):
        letterIndices = [_k for _k, _j in enumerate(self._wordList[idx])
if _j == self._letterList[it]]
        for index in letterIndices:
            wordValues[index] = association[it]
```

... by creating and using 3 distincts variables instead of an array of elements, I would be able to lower the time complexity to O(n^2) once again.

## Problem C: From X to Y

***I am sorry I don't know how to explain it without showing an example.***

The problem is to transform X into Y only using 3 operations:

- `X = min(X * m, Y * 2)`
- `X = X - 2`
- `X = X - 1`

For each iteration, I would look at:

- X is lower than Y
- The distance between the result of each operation and the goal Y until I will find Y

As an example:

```
Let's consider X = 4, Y = 12 and m = 25.

X is lower than Y. There, we have X < Y so we're going to use our first
operation.

4 * 25 has a higher distance from Y than Y * 2. So we're going to use our
first operation.

Then, we're lowering the value as we can (-2 until it's not possible
anymore), checking until we're reaching Y.
```

At first, to my mind, that would sometimes be the first operation and then the others operations.

To me the complexity of this algorithm would be O(n), using only one loop composed of conditional statements just looking for the conditions to trigger the operations.

## Problem D: Shortest Paths

We're having a directed graph with a source vertex and a distance function on the arcs.

To it, I would use the Dijkstra algorithm once again. Since we are in a directed weighted graph, it's the most accurate algorithm I know.

From the start vertex, we will try to get the shortest weight for each edge. For each edge, we will keep the vertex we just went to in memory.

Then, we will iterate this for each know vertex. If a vertex is not the start vertex, we will add the weight already used to access this vertex to the other weights we're going to check.

Finally, I will iterate until I find the end vertex, giving me the shortest path.

The way I look at it would be a O(n^2) too. It needs to iterate until our vertices list is empty and we need to iterate over our already checked vertices.