Deadlines:

For all Groups: October 18 2021

Grading system:

- 1 problem - 6

- 2 problems - 7

- 3 problems - 8

- 4 problems - 9

- 5 problems and bonus– 10

1. Subsets

Given an integer array set of **unique** elements, return *all possible subsets (the power set)*.

Return the solution in **any order**.

Example 1:

Input: set = [1,2,3]

Output: [[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]

Example 2:

Input: set = [0]

Output: [[],[0]]

2. XNOR

Create a program that would ask for two boolean values (true or false, 0 or 1) and would output the result for the XNOR operation performed on them.

You're allowed to use only `and`, `or` and `not` operations.

3. Regular Expression Matching

Given an input string **string** and a pattern **pattern**, implement regular expression matching with support for **’ . ’** and **’ * ’** where:

- **‘ - ‘** Matches any single character.
- **‘ * ‘** Matches zero or more of the preceding element.

The matching should cover the **entire** input string (not partial).

Example 1:

Input: string = "aa", pattern = "a"

Output: false

Explanation: "a" does not match the entire string "aa".

Example 2:

Input: string = "aa", pattern = "a*"

Output: true

Explanation: '*' means zero or more of the preceding element, 'a'. Therefore, by repeating 'a' once, it becomes "aa".

Example 3:

Input: string = "ab", pattern = ".*"

Output: true

Explanation: ".*" means "zero or more (*) of any character (.)".

Example 4:

Input: string = "aab", pattern = "c*a*b"

Output: true

Explanation: c can be repeated 0 times, a can be repeated 1 time. Therefore, it matches "aab".

Example 5:

Input: string = "mississippi", pattern = "mis*is*p*."

Output: false

4. Truth table solver

You have to write a program that computes the truth table for various expressions. The set of expressions are limited to:

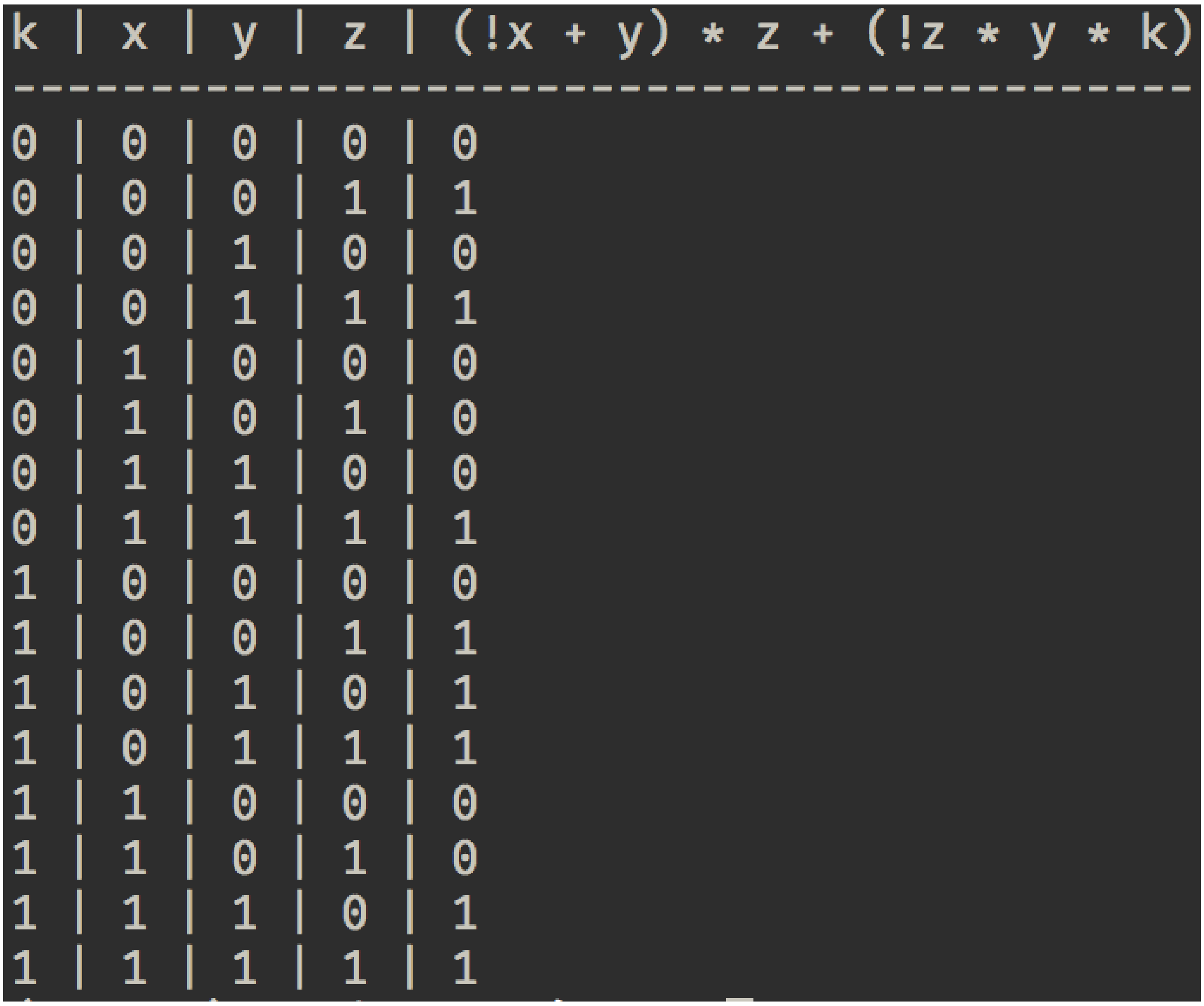
- `and` operation

- `or` operation

- `not` operation

- supports parenthesis

An example of your program input is `(!x + y) * z + (!z * y * k)` and it should print out:



```
k | x | y | z | (!x + y) * z + (!z * y * k)
-------------------------------------------
0 | 0 | 0 | 0 | 0
0 | 0 | 0 | 1 | 1
0 | 0 | 1 | 0 | 0
0 | 0 | 1 | 1 | 1
0 | 1 | 0 | 0 | 0
0 | 1 | 0 | 1 | 0
0 | 1 | 1 | 0 | 0
0 | 1 | 1 | 1 | 1
1 | 0 | 0 | 0 | 0
1 | 0 | 0 | 1 | 1
1 | 0 | 1 | 0 | 1
1 | 0 | 1 | 1 | 1
1 | 1 | 0 | 0 | 0
1 | 1 | 0 | 1 | 0
1 | 1 | 1 | 0 | 1
1 | 1 | 1 | 1 | 1
```

Here are some examples of input that your program should support

```

x + y
```

```
!x * y

(!x + y) * x + y * !k
```

Note:

I strongly recommend to use the python **`eval`** function. Inventing math operations and their execution priority is **not** the aim of this exercise.

5. Leibniz harmonic triangle

Write a program that prints the harmonic triangle for the depth `n`, where `n` is an input value.

Tip:

If you're using Python you might look into **`fractions`** module.

Bonus: A game of life foreplay (aka [Elementary cellular automaton]

In this problem we're going to take a look at elementary cellular automaton. Every cell is like a small micro organism with a few primitive rules. When combining with other cells they form interesting patterns. There also is an interesting (ted talk) given by Stephen Wolfram that touches on this topic.

Your task is to randomly generate a list (let's say of length 200, it's up to you in the end, just make sure to be long enough) containing only the numbers `0` and `1`. Then you start iterating over the list in order to compute the *next generation*. The rules that apply for the next generation are the following.

111	110	101	100	011	010	001	000
0	1	1	0	1	1	1	0

For instance if the cells `1`, `2` and `3` have the value `1 1 0` the 2nd cell of the next generation will be `1`.

PRO Tip:

READ THIS LINK: https://natureofcode.com/book/chapter-7-cellular-automata/

Note:

For computing the first and the last cell you can consider the missing parent to be `0`.

Now you have to compute the next 100 generations and print the resulting matrix with color for value `1` and with white for value `1`. Once You've done that try to change the first generation from randomly generated numbers to all values to be `0` and the last element is `1`. Observe the result.

The rule applied above is called rule 110, there is actually a list of rules that renders quite interesting patterns.

Change arbitrary the initial rule and observe the differences.

Maybe you can find a new interesting pattern for Bunica's covor.

Bonus task:

Make your program in a way that it would be easy to change the number of pixels rendered for every cell. For instance my cell is 1 x 1 pixels. And by changing one or two variables my cell would change to 5 x 5 pixels.