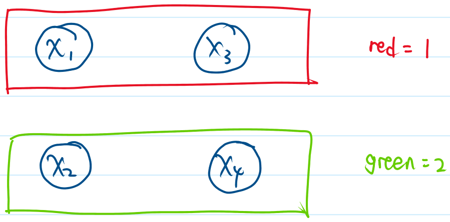
Reference

Wednesday, June 26, 2024 10:06 AM

1. https://www.rareskills.io/zk-book
2. https://www.rareskills.io/post/arithmetic-circuit
3. https://www.rareskills.io/post/circom-tutorial

Problem statement

Given vertices (x_1, x_2, x_3, x_4). Prove the graph is bipartite.



The solution to this problem is a "witness vector":

$$a = [1, \underbrace{x_1, x_2, x_3, x_4}]$$

↑ constant term (public input)

how to "2-color" this graph (private input)

- ➢ Verifier has access to public inputs
- ➢ Prover has access to both public and private inputs
- ➢ Intermediate signals can exist but we don't need them for this specific problem.

Comment:
We assume that verifying a problem is always easier than solving the problem. Otherwise, P = NP.

Q: Is it true that "all NP problems can be represented by arithmetic circuit"?

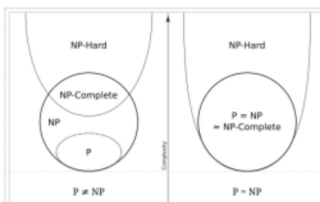
A:

1. Cook–Levin theorem proved that SAT (boolean satisfiability problem) is NP-complete.
2. By definition of NP-completeness, every NP problem can be reduced to NP-complete problem, therefore can be reduced to SAT.
3. Boolean circuit can be transformed into arithmetic circuit.

- ➢ Thus every NP problem should have arithmetic circuit representation.

Resource: P / NP / NP-complete / NP-hard
https://stackoverflow.com/questions/210829/what-is-an-np-complete-in-computer-science

A nice diagram from Wikipedia:
https://en.wikipedia.org/wiki/NP-hardness



Euler diagram for P, NP, NP-complete, and NP-hard set of problems. The left side is valid under the assumption that P≠NP, while the right side is valid under the assumption that P=NP (except that the empty language and its complement are never NP-complete)

Arithmetic circuit

1. Each vertex should have color 1 or 2

$$(x_1-1)(x_1-2)=0 \rightarrow x_1^2-3x_1+2=0$$
$$(x_2-1)(x_2-2)=0 \rightarrow x_2^2-3x_2+2=0$$
$$(x_3-1)(x_3-2)=0 \rightarrow x_3^2-3x_3+2=0$$
$$(x_4-1)(x_4-2)=0 \rightarrow x_4^2-3x_4+2=0$$

2. Vertices from different "groups" should have different colors:

$$x_1x_2-2=0 \quad ①$$
$$x_1x_4-2=0$$
$$x_2x_3-2=0 \quad ②$$

($x_1x_3-2=0$ is implied by ① and ② implicitly)

R1CS

Turn arithmetic circuits into system of quadratic equations (constraints).

1. Each vertex should have color 1 or 2

$$x_1x_1=3x_1-2$$
$$x_2x_2=3x_2-2$$
$$x_3x_3=3x_3-2$$

$x_2 x_2 = 3x_2 - 2$

$x_3 x_3 = 3x_3 - 2$

$x_4 x_4 = 3x_4 - 2$

2. Vertices from different "groups" should have different colors:

$x_1 x_2 = 2$

$x_1 x_4 = 2$

$x_2 x_3 = 2$

Turn system of equations into matrix form

Recall that witness vector $a = [1, x_1, x_2, x_3, x_4]$

We turn systems of equation into $L \cdot a \circ R \cdot a = O \cdot a$

$\uparrow$ Hadamard product

where L, R, O are 3 matrices with same dimension:

- # rows == # constraints
- # columns == dimension of witness vector

What is Hadamard product?

From Wikipedia: https://en.wikipedia.org/wiki/Hadamard_product_(matrices)

Definition [edit]

For two matrices A and B of the same dimension $m \times n$, the Hadamard product $A \odot B$ (sometimes $A \circ B$[4][5][6]) is a matrix of the same dimension as the operands, with elements given by[3]

$$(A \odot B)_{ij} = (A)_{ij}(B)_{ij}.$$

For matrices of different dimensions ($m \times n$ and $p \times q$, where $m \neq p$ or $n \neq q$), the Hadamard product is undefined.

For example, the Hadamard product for two arbitrary 2 × 3 matrices is:

$$\begin{bmatrix} 2 & 3 & 1 \\ 0 & 8 & -2 \end{bmatrix} \circ \begin{bmatrix} 3 & 1 & 4 \\ 7 & 9 & 5 \end{bmatrix} = \begin{bmatrix} 2 \times 3 & 3 \times 1 & 1 \times 4 \\ 0 \times 7 & 8 \times 9 & -2 \times 5 \end{bmatrix} = \begin{bmatrix} 6 & 3 & 4 \\ 0 & 72 & -10 \end{bmatrix}$$

Begin transformation:

Constraint 1: $x_1 x_1 = 3x_1 - 2$

$$\Rightarrow \underbrace{(x_1)}_{L} \cdot \underbrace{(x_1)}_{R} = \underbrace{3x_1 - 2}_{O}$$

$$\underbrace{\begin{array}{c} \begin{array}{ccccc} 1 & x_1 & x_2 & x_3 & x_4 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ \\ & & \text{don't know yet} & & \\ \\ \end{bmatrix} \end{array}}_{L} \cdot \underbrace{\begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}}_{a} \circ \underbrace{\begin{array}{c} \begin{array}{ccccc} 1 & x_1 & x_2 & x_3 & x_4 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ \\ & & \text{don't know yet} & & \\ \\ \end{bmatrix} \end{array}}_{R} \cdot \underbrace{\begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}}_{a} = \underbrace{\begin{array}{c} \begin{array}{ccccc} 1 & x_1 & x_2 & x_3 & x_4 \end{array} \\ \begin{bmatrix} -2 & 3 & 0 & 0 & 0 \\ \\ & & \text{don't know yet} & & \\ \\ \end{bmatrix} \end{array}}_{O} \cdot \underbrace{\begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}}_{a}$$

Check if this transformation is done correctly:

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \circ \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} -2 & 3 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

$$[x_1] \circ [x_1] = [-2 + 3x_1]$$

$$[x_1 \cdot x_1] = [3x_1 - 2]$$

Repeat this process for each constraint in the system of equation, or equivalently, for each row in the matrices.
In the end this is what we get:

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \cdot a \circ \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \cdot a = \begin{bmatrix} -2 & 3 & 0 & 0 & 0 \\ -2 & 0 & 3 & 0 & 0 \\ -2 & 0 & 0 & 3 & 0 \\ -2 & 0 & 0 & 0 & 3 \\ 2 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 \end{bmatrix} \cdot a$$

Implementing R1CS in Python

My implementation: https://github.com/ret2basic/Groth16/blob/main/groth16.py

groth16.py ×

groth16.py > ...

```python
import numpy as np
import galois
from functools import reduce
from py_ecc.bn128 import G1, G2, multiply, add, curve_order, Z1, pairing, neg, final_exponentiate, FQ12

# curve_order = 1151
GF = galois.GF(curve_order) # we work with bn128/bn254 curve

```

galois library handles modular arithmetic over scalar field:

```
>>> import galois
>>> curve_order = 17
>>> GF = galois.GF(curve_order)
>>> operand1 = GF(6)
>>> operand2 = GF(14)
>>> operand1 + operand2
GF(3, order=17)
>>> int(_)
3
>>> operand1 * operand2
GF(16, order=17)
>>> int(_)
16
>>>
```

_ = return value from previous computation

operand1/operand2 is equivalent to operand1 * pow(operand2, -1, curve_order).

```
>>> operand1 - operand2
GF(9, order=17)
>>> int(_)
9
>>> operand1 / operand2
GF(15, order=17)
>>> int(_)
15
>>>
```

galois can also handle matrices and polynomials:

ret2basic@PwnieIsland: ~ 80x24

```
>>> import numpy as np
>>> np.array([[1,0,0],[0,1,0],[0,0,1]])
array([[1, 0, 0],
       [0, 1, 0],
       [0, 0, 1]])
>>> GF(_)
GF([[1, 0, 0],
    [0, 1, 0],
    [0, 0, 1]], order=17)
>>> A = _
>>> B = GF(np.array([[0,2,0],[4,0,0],[0,0,8]]))
>>> A + B
GF([[1, 2, 0]
```

```
>>> import numpy as np
>>> np.array([[1,0,0],[0,1,0],[0,0,1]])
array([[1, 0, 0],
       [0, 1, 0],
       [0, 0, 1]])
>>> GF(_)
GF([[1, 0, 0],
    [0, 1, 0],
    [0, 0, 1]], order=17)
>>> A = _
>>> B = GF(np.array([[0,2,0],[4,0,0],[0,0,8]]))
>>> A + B
GF([[1, 2, 0],
    [4, 1, 0],
    [0, 0, 9]], order=17)
>>> A * B
GF([[0, 0, 0],
    [0, 0, 0],
    [0, 0, 8]], order=17)
>>>
```

```
>>> poly1 = galois.Poly([1, 2, 3, 4], field=GF)
>>> poly1
Poly(x^3 + 2x^2 + 3x + 4, GF(17))
>>> poly2 = galois.Poly([5, 2, 3], field=GF)
>>> poly2
Poly(5x^2 + 2x + 3, GF(17))
>>> poly1 + poly2
Poly(x^3 + 7x^2 + 5x + 7, GF(17))
>>> poly1 * poly2
Poly(5x^5 + 12x^4 + 5x^3 + 15x^2 + 12, GF(17))
>>>
```

Back to groth16:

```python
# R1CS matrices

L = np.array([
    [0, 1, 0, 0, 0],
    [0, 0, 1, 0, 0],
    [0, 0, 0, 1, 0],
    [0, 0, 0, 0, 1],
    [0, 1, 0, 0, 0],
    [0, 1, 0, 0, 0],
    [0, 0, 1, 0, 0],
])

R = np.array([
    [0, 1, 0, 0, 0],
    [0, 0, 1, 0, 0],
    [0, 0, 0, 1, 0],
    [0, 0, 0, 0, 1],
    [0, 0, 1, 0, 0],
    [0, 0, 0, 0, 1],
    [0, 0, 0, 1, 0],
])
```

```python
O = np.array([
    [curve_order-2, 3, 0, 0, 0],
    [curve_order-2, 0, 3, 0, 0],
    [curve_order-2, 0, 0, 3, 0],
    [curve_order-2, 0, 0, 0, 3],
    [2, 0, 0, 0, 0],
    [2, 0, 0, 0, 0],
    [2, 0, 0, 0, 0],
])

L_galois = GF(L)
R_galois = GF(R)
O_galois = GF(O)
```

Negative numbers such as -2 must be converted to curve_order - 2.

Prover computes witness:

```python
# In reality this witness is prover's secret, only prover knows it
x1 = GF(1)
x2 = GF(2)
x3 = GF(1)
x4 = GF(2)
# a is the witness
a = GF(np.array([1, x1, x2, x3, x4]))

assert all(np.equal(np.matmul(L_galois, a) * np.matmul(R_galois, a), np.matmul(O_galois, a))), "not equal"
```

Separate public inputs and private inputs (for future use):

From circom doc:

```
pragma circom 2.0.0;
```

```
assert all(np.equal(np.matmul(L_gates, a) * np.matmul(R_gates, a), np.matmul(O_gates, a))), "not equal"
```

Separate public inputs and private inputs (for future use):

```python
# witness = [1, x1, x2, x3, x4]
# Only the first entry [1] is public input
# [x1, x2, x3, x4] are private inputs that only the prover knows
l = 0
public_inputs = a[:l+1]
private_inputs = a[l+1:]
```

From circom doc:

```
pragma circom 2.0.0;

template Multiplier2(){
   //Declaration of signals
   signal input in1;
   signal input in2;
   signal output out;
   out <== in1 * in2;
}

component main {public [in1,in2]} = Multiplier2();
```

in1 and in2 are public inputs

Do it in circom

A simple piece of circom I wrote to represent the bipartite graph problem above: https://zkrepl.dev/?gist=810ac7fb657dc07bd933096cb36b7d5f

main.circom × + Add File

```
pragma circom 2.1.6;

// bipartite graph problem arithmetization

template Bipartite(n) {
    // coloring for 4 vertices x1, x2, x3, x4
    // in[0] -> x1
    // in[1] -> x2
    // in[2] -> x3
    // in[3] -> x4
    signal input in[n];

    // Condition 1: color is either 1 or 2 for each vertex
    (in[0] - 1) * (in[0] - 2) === 0;
    (in[1] - 1) * (in[1] - 2) === 0;
    (in[2] - 1) * (in[2] - 2) === 0;
    (in[3] - 1) * (in[3] - 2) === 0;

    // Condition 2: vertices from different "groups" have different colors
    in[0] * in[1] === 2;
    in[0] * in[3] === 2;
    in[1] * in[2] === 2;

}
component main = Bipartite(4);

/* INPUT = {"in": [1, 2, 1, 2]} */
```

```
template instances: 1
non-linear constraints: 7
linear constraints: 0
public inputs: 0
public outputs: 0
private inputs: 4
private outputs: 0
wires: 5
labels: 5
Written successfully: ./main.r1cs
Written successfully: ./main.sym
Written successfully: ./main_js/main.wasm
Everything went okay, circom safe
Compiled in 0.76s
```

ARTIFACTS:

Finished in 0.87s

- main.wasm (35.02KB)
- main.js (9.18KB)
- main.wtns (0.24KB)
- main.r1cs (1.14KB)
- main.sym (0.07KB)

SAVE:

Saved to Github

```
<iframe src="https://zkrepl.dev/?gist=810ac7fb657dc07bd933096cb36b7d5f" height="400" width="1000" style="border:1px solid #ddd"></iframe>
```

KEYS + SOLIDITY + HTML:

Groth16 | PLONK | Verify

base field (field_modulus)

Note: circom operates in **scalar field (curve_order)** as well: https://docs.circom.io/circom-language/basic-operators/

2G + 3G = 5G

curve_order * G = O -> point at infinity

Field Elements

A field element is a value in the domain of Z/pZ, where p is the prime number set by default to

`p = 21888242871839275222246405745257275088548364400416034343698204186575808495617.`

As such, field elements are operated in arithmetic modulo p.

The circom language is parametric to this number, and it can be changed without affecting the rest of the language (using `GLOBAL_FIELD_P`).

```
ret2basic@PwnieIsland: ~/Desktop/zero-knowledge-puzzles 79x22
>>> from py_ecc.bn128 import field_modulus, curve_order
>>> field_modulus
21888242871839275222246405745257275088696311157297823662689037894645226208583
>>> curve_order
21888242871839275222246405745257275088548364400416034343698204186575808495617
>>>
```

Circom behind the scene

First we check if it actually compiles:

```
ret2basic@PwnieIsland: ~/Desktop/bipartite 80x24
ret2basic@PwnieIsland:~/Desktop/bipartite$ ll
total 12
drwxrwxr-x  2 ret2basic ret2basic 4096 Jun  5 22:47 ./
drwxr-xr-x 30 ret2basic ret2basic 4096 Jun  5 22:46 ../
-rw-rw-r--  1 ret2basic ret2basic  664 Jun  5 22:47 bipartite.circom
ret2basic@PwnieIsland:~/Desktop/bipartite$ circom bipartite.circom
template instances: 1
Everything went okay
ret2basic@PwnieIsland:~/Desktop/bipartite$
```

Generate R1CS file:

```
ret2basic@PwnieIsland: ~/Desktop/bipartite 80x24
ret2basic@PwnieIsland:~/Desktop/bipartite$ circom bipartite.circom --r1cs --sym
template instances: 1
non-linear constraints: 7
linear constraints: 0
public inputs: 0
private inputs: 4
public outputs: 0
wires: 5
labels: 5
Written successfully: ./bipartite.r1cs
Written successfully: ./bipartite.sym
Everything went okay
ret2basic@PwnieIsland:~/Desktop/bipartite$
```

Here we generate the .sym file so that we can provide symbolic input.json later, such as {"in": [1, 2, 1, 2]}.

Check out R1CS file:

```
ret2basic@PwnieIsland: ~/Desktop/bipartite 80x24
ret2basic@PwnieIsland:~/Desktop/bipartite$ ll
total 20
drwxrwxr-x  2 ret2basic ret2basic 4096 Jun  5 22:48 ./
drwxr-xr-x 30 ret2basic ret2basic 4096 Jun  5 22:46 ../
-rw-rw-r--  1 ret2basic ret2basic  664 Jun  5 22:47 bipartite.circom
-rw-rw-r--  1 ret2basic ret2basic 1136 Jun  5 22:48 bipartite.r1cs
-rw-rw-r--  1 ret2basic ret2basic   68 Jun  5 22:48 bipartite.sym
ret2basic@PwnieIsland:~/Desktop/bipartite$ snarkjs r1cs print bipartite.r1cs
[INFO]  snarkJS: [ 2188824287183927522224640574525727508854836440041603434369820
41865758084956161 +main.in[0] ] * [ 2188824287183927522224640574525727508854836
4004160343436982041865758084956151 +main.in[0] ] - [  ] = 0
[INFO]  snarkJS: [ 2188824287183927522224640574525727508854836440041603434369820
41865758084956161 +main.in[1] ] * [ 2188824287183927522224640574525727508854836
4004160343436982041865758084956151 +main.in[1] ] - [  ] = 0
[INFO]  snarkJS: [ 2188824287183927522224640574525727508854836440041603434369820
41865758084956161 +main.in[2] ] * [ 2188824287183927522224640574525727508854836
4004160343436982041865758084956151 +main.in[2] ] - [  ] = 0
[INFO]  snarkJS: [ 2188824287183927522224640574525727508854836440041603434369820
41865758084956161 +main.in[3] ] * [ 2188824287183927522224640574525727508854836
4004160343436982041865758084956151 +main.in[3] ] - [  ] = 0
[INFO]  snarkJS: [ main.in[0] ] * [ main.in[1] ] - [ 21 ] = 0
[INFO]  snarkJS: [ main.in[0] ] * [ main.in[3] ] - [ 21 ] = 0
[INFO]  snarkJS: [ main.in[1] ] * [ main.in[2] ] - [ 21 ] = 0
ret2basic@PwnieIsland:~/Desktop/bipartite$
```

Could be circom bug. Circom is printing 1 at the end of each huge number.

Generate wasm file as preparation for computing witness:

```
ret2basic@PwnieIsland: ~/Desktop/bipartite 80x24
ret2basic@PwnieIsland:~/Desktop/bipartite$ circom bipartite.circom --r1cs --sym
--wasm
template instances: 1
non-linear constraints: 7
linear constraints: 0
public inputs: 0
private inputs: 4
public outputs: 0
wires: 5
labels: 5
Written successfully: ./bipartite.r1cs
Written successfully: ./bipartite.sym
Written successfully: ./bipartite_js/bipartite.wasm
Everything went okay
ret2basic@PwnieIsland:~/Desktop/bipartite$
```

In ./bipartite_js directory, create input.json:

```
ret2basic@PwnieIsland: ~/Desktop/bipartite/bipartite_js 80x24
  GNU nano 6.2                          input.json *
{"in": [1, 2, 1, 2]}
```

Generate witness and check out:

```
ret2basic@PwnieIsland: ~/Desktop/bipartite/bipartite_js 80x24
ret2basic@PwnieIsland:~/Desktop/bipartite/bipartite_js$ node generate_witness.js
 bipartite.wasm input.json witness.wtns
ret2basic@PwnieIsland:~/Desktop/bipartite/bipartite_js$ snarkjs wtns export json
 witness.wtns
ret2basic@PwnieIsland:~/Desktop/bipartite/bipartite_js$ cat witness.json
[
 "1",
 "1",
 "2",
 "1",
 "2"
]ret2basic@PwnieIsland:~/Desktop/bipartite/bipartite_js$
```

Circomlib - comparators.circom

https://github.com/iden3/circomlib/blob/master/circuits/comparators.circom

```
template IsZero() {
    signal input in;
    signal output out;

    signal inv;

    inv <-- in!=0 ? 1/in : 0;

    out <== -in*inv +1;
    in*out === 0;
}
```

Idea: non-zero field element has multiplicative inverse.

<-- assign to signal

<== assign and add constraint

```
main.circom  ×    + Add File

pragma circom 2.1.6;

template IsZeroTheWrongWay() {
    signal input in;
    signal output out;

    if (in == 0) {
        out <== 1;
    } else {
        out <== 0;
    }
}

component main = IsZeroTheWrongWay();

/* INPUT = {
    "in": "5"
} */
```

```
SHIFT-ENTER TO RUN
CMD-S TO SAVE AS GITHUB GIST

STDERR:
error[T2005]: Typing error found
    "main.circom":7:9

 7 |     if (in == 0) {
   |        ^^^^^^^^ There are constraints depending on the value of the condition and it can
be unknown during the constraint generation phase

previous errors were found

STDOUT:
Compiled in 1.75s

FAIL:
Too many values for input signal in

KEYS + SOLIDITY + HTML:
Groth16    PLONK    Verify
```

Not that easy

```
template IsEqual() {
    signal input in[2];
    signal output out;

    component isz = IsZero();

    in[1] - in[0] ==> isz.in;

    isz.out ==> out;
}
```

Common pattern: When there are multiple inputs, store them into an array. Usually it is called in[].

component: instantiate another template and "wire" inputs.

Arrow direction can be <== or ==>

```
template LessThan(n) {
    assert(n <= 252);
    signal input in[2];
    signal output out;

    component n2b = Num2Bits(n+1);

    n2b.in <== in[0]+ (1<<n) - in[1];

    out <== 1-n2b.out[n];
}
```

```
template Num2Bits(n) {
    signal input in;
    signal output out[n];
    var lc1=0;

    var e2=1;
    for (var i = 0; i<n; i++) {
        out[i] <-- (in >> i) & 1;
        out[i] * (out[i] -1 ) === 0;
        lc1 += out[i] * e2;
        e2 = e2+e2;
    }

    lc1 === in;
}
```

accumulator

Compare 5 = 0101 $n=4$
and 7 = 0111

$e_2 = 1, 2, 4, 8, \cdots$

0101
+ 10000 ← 1 << 4
10101

10101
− 0111
01110

If MSB is 0, then $a < b$
If MSB is 1, then $a > b$

1. 5 = 0101 $e_2 = 1$ $lc1 = 0$
& 1
1 → out[0]

$lc1 \mathrel{+}= 1 * 1 \to lc1 = 1$
$e_2 = 2$

2. 0010 $e_2 = 2$ $lc1 = 1$
& 1
0 → out[1]

$lc1 \mathrel{+}= 0 * 1 \to lc1 = 1$
$e_2 = 4$

⋮

```
// N is the number of bits the input  have.
// The MSF is the sign bit.
template LessEqThan(n) {
    signal input in[2];
    signal output out;

    component lt = LessThan(n);

    lt.in[0] <== in[0];
    lt.in[1] <== in[1]+1;
    lt.out ==> out;
}
```

```
// N is the number of bits the input  have.
// The MSF is the sign bit.
template GreaterThan(n) {
    signal input in[2];
    signal output out;

    component lt = LessThan(n);

    lt.in[0] <== in[1];
    lt.in[1] <== in[0];
    lt.out ==> out;
}
```

```
// N is the number of bits the input  have.
// The MSF is the sign bit.
template GreaterEqThan(n) {
    signal input in[2];
    signal output out;

    component lt = LessThan(n);

    lt.in[0] <== in[1];
    lt.in[1] <== in[0]+1;
    lt.out ==> out;
}
```