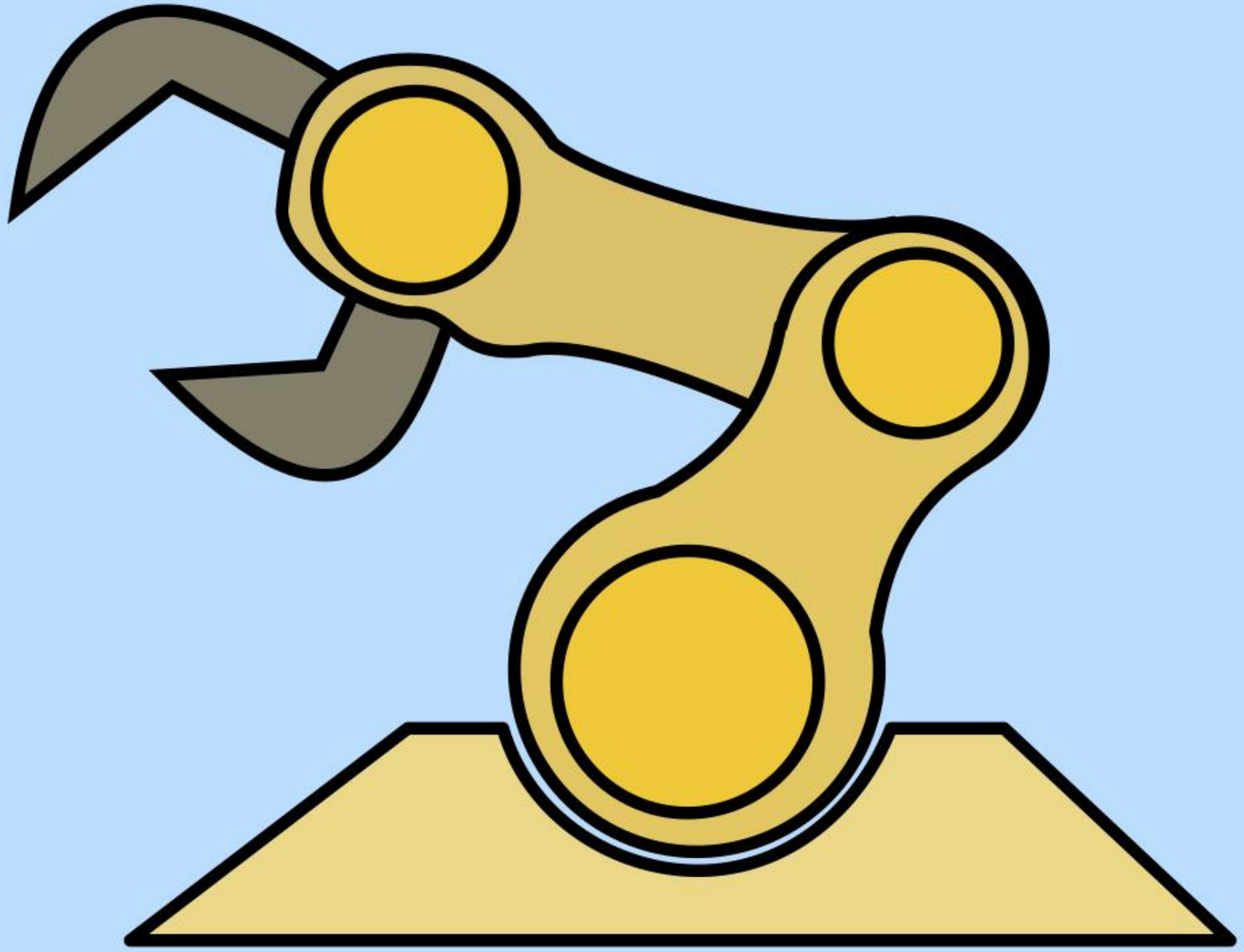
General Recursive Solving Engines

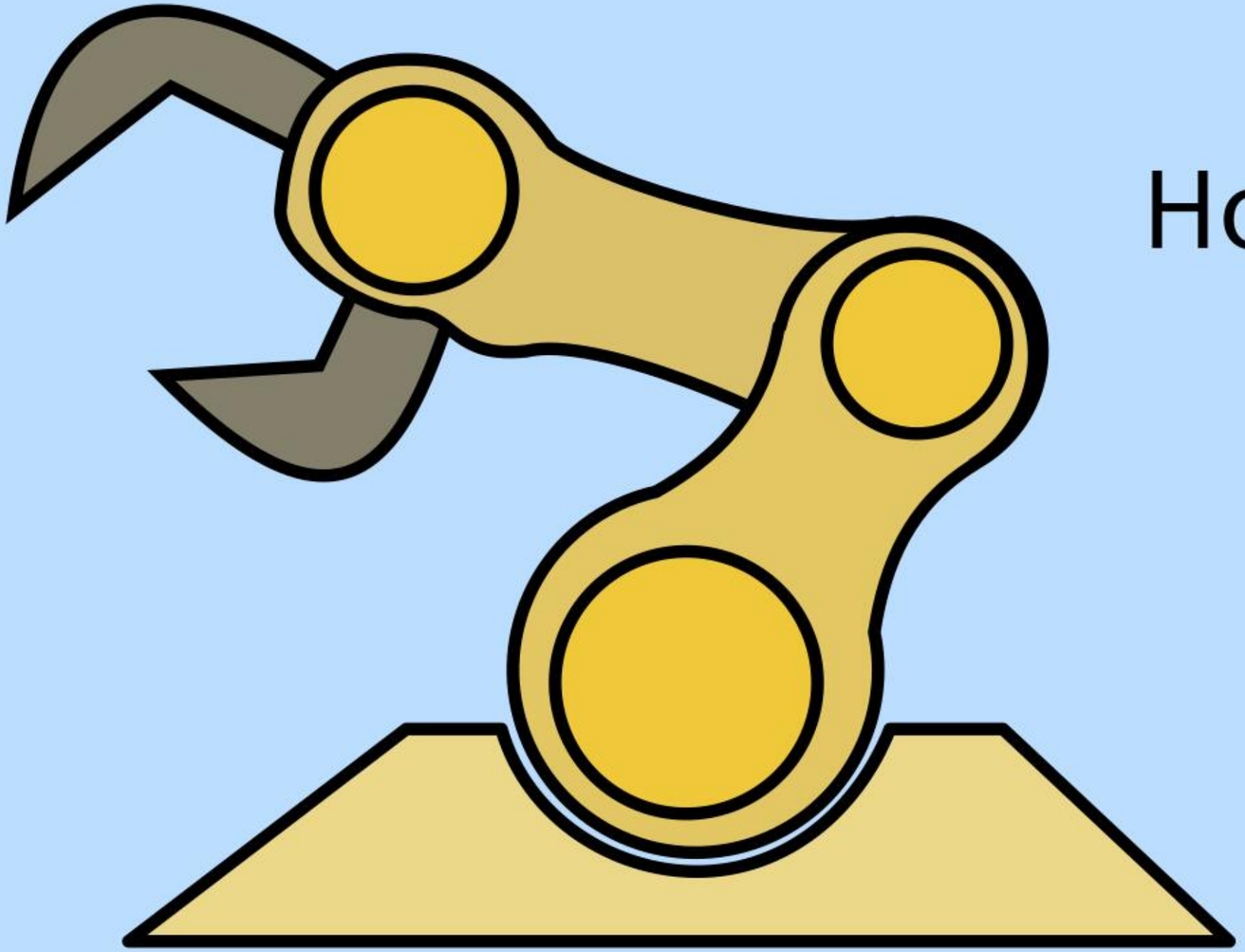
Arved Friedemann

BCTCS 2020

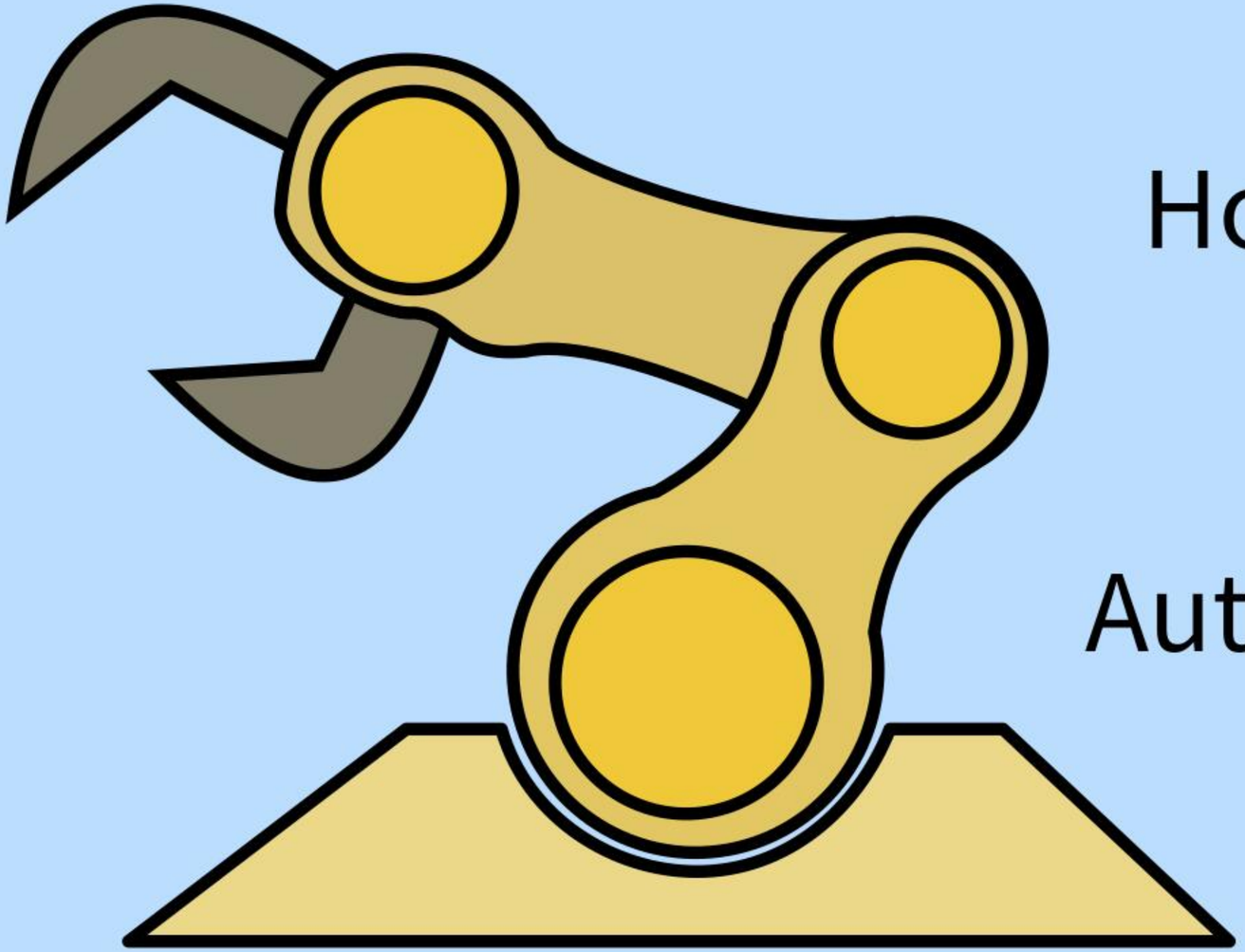
Supervised by
Dr Monika Seisenberger
Dr Oliver Kullmann

Swansea University
Prifysgol Abertawe



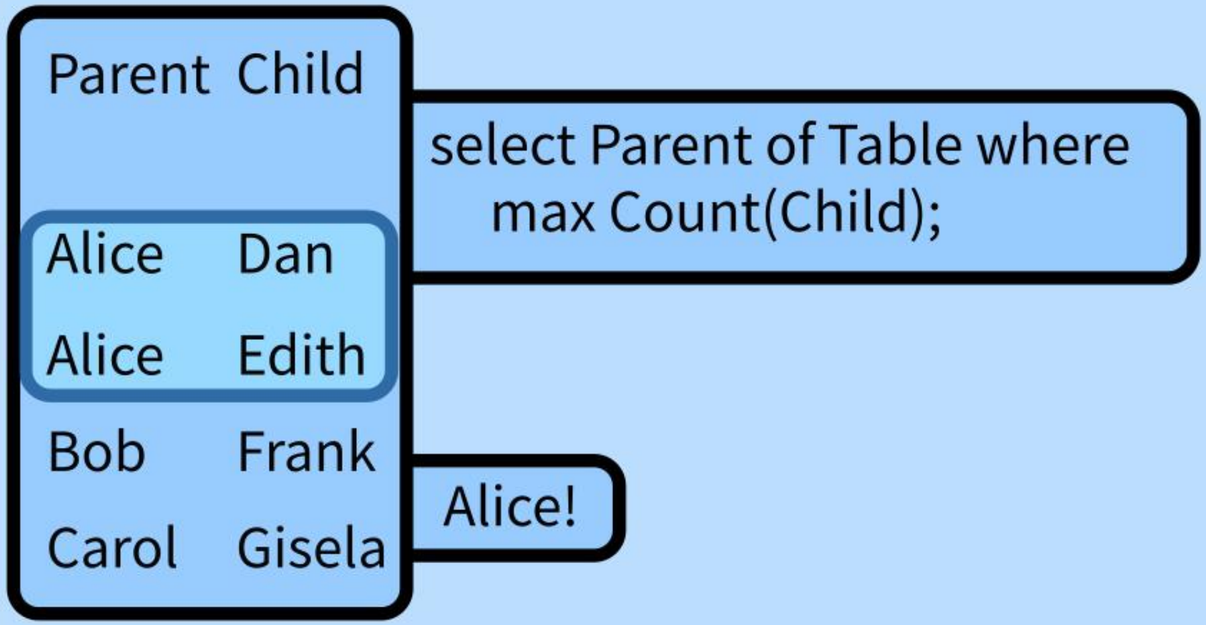


How do we automate **everything?**



How do we automate **everything?**

Automate **search!**


Parent Child
Alice Dan
Alice Edith
Bob Frank
Carol Gisela
select Parent of Table where
max Count(Child);
Alice!

Parent	Child
Alice	Dan
Alice	Edith
Bob	Frank
Carol	Gisela

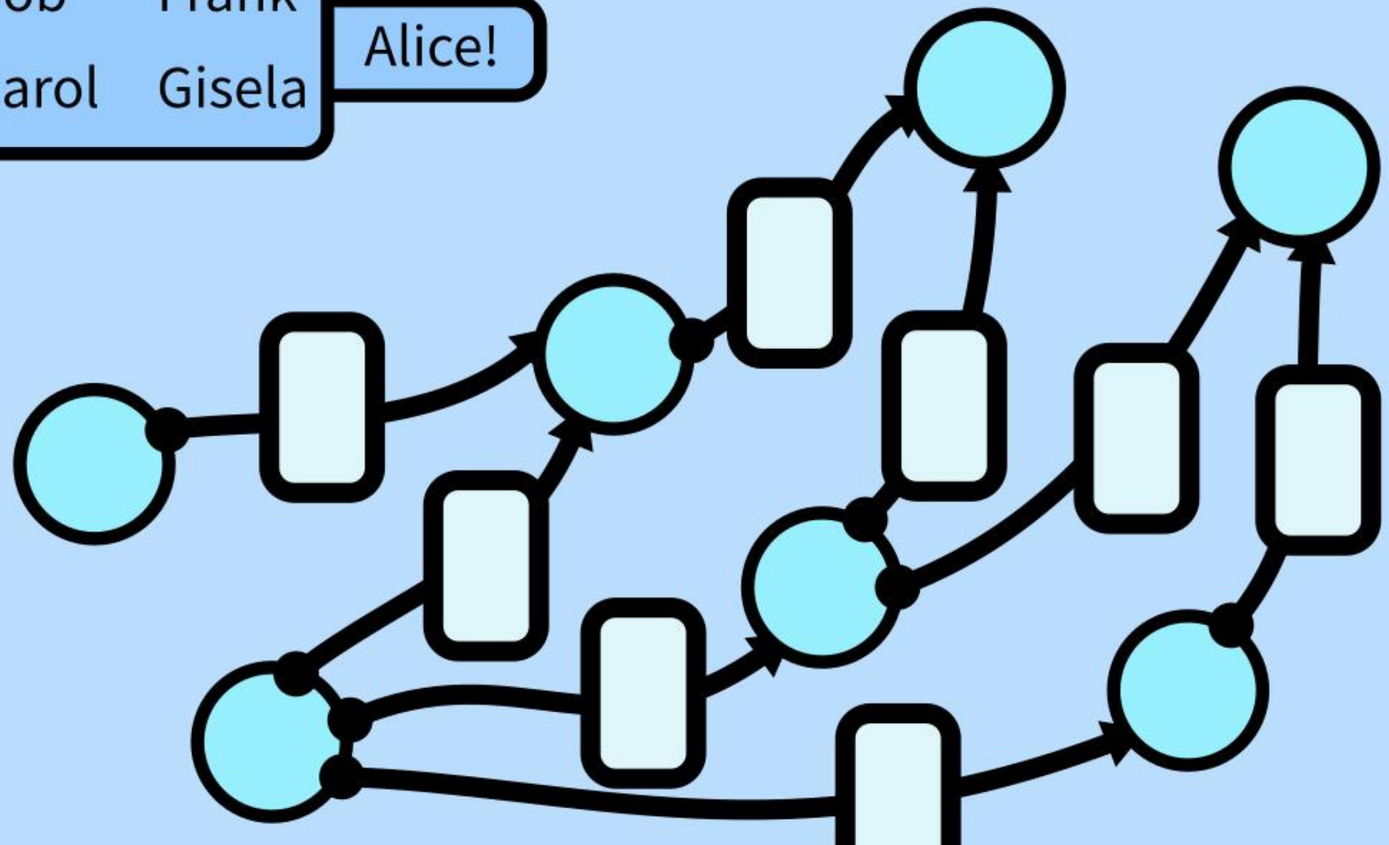
```
select Parent of Table where
    max Count(Child);
```

Alice!

Parent	Child
Alice	Dan
Alice	Edith
Bob	Frank
Carol	Gisela

select Parent of Table where
max Count(Child);

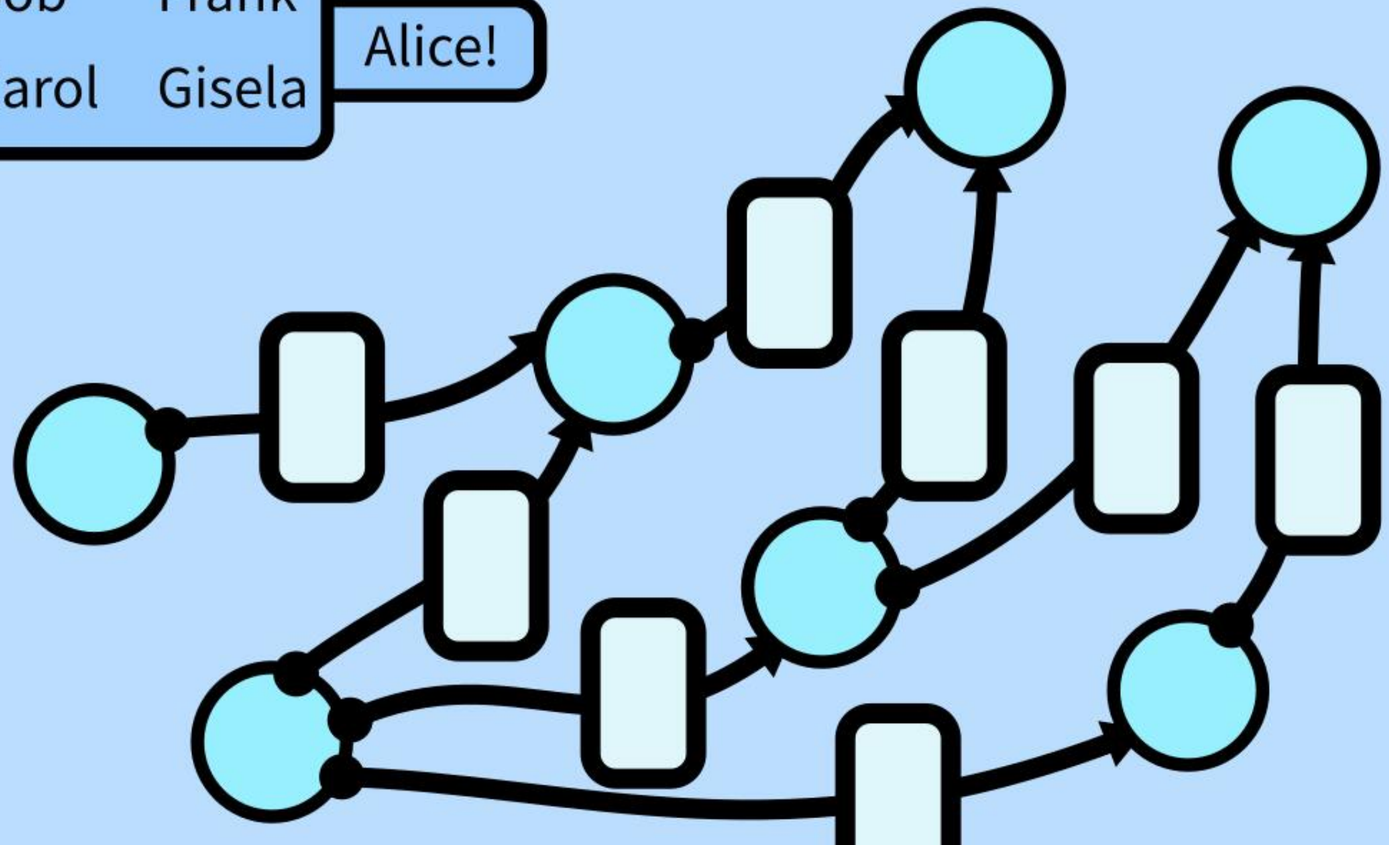
Alice!



Parent	Child
Alice	Dan
Alice	Edith
Bob	Frank
Carol	Gisela

select Parent of Table where
max Count(Child);

Alice!



4	8	
3	6	5
1	2	7

```
sort :: [Int] -> [Int]
sort []     = []
sort (x:xs) = ?!?

max :: [Int] -> Int
max xs = ?!?
```

SAT : Solvers for Boolean Logic

SAT : Solvers for Boolean Logic

SMT : Merging of Solvers for other Theories

(Unevaluated Functions, Integer Arithmetic, Bitvectors...)

SAT : Solvers for Boolean Logic

SMT : Merging of Solvers for other Theories

(Unevaluated Functions, Integer Arithmetic, Bitvectors...)

FOL : Solvers for First Order Logic

SAT : Solvers for Boolean Logic

SMT : Merging of Solvers for other Theories

(Unevaluated Functions, Integer Arithmetic, Bitvectors...)

FOL : Solvers for First Order Logic

random-SAT : SAT Solvers for uniformly sampling assignments

...

Problems:

No proper embedding into programming languages

Curry

Haskell CP

PROLOG

Problems:

No proper embedding into programming languages

Barely known why they work (heuristics etc.)

Automatic construction currently impossible

Aims:

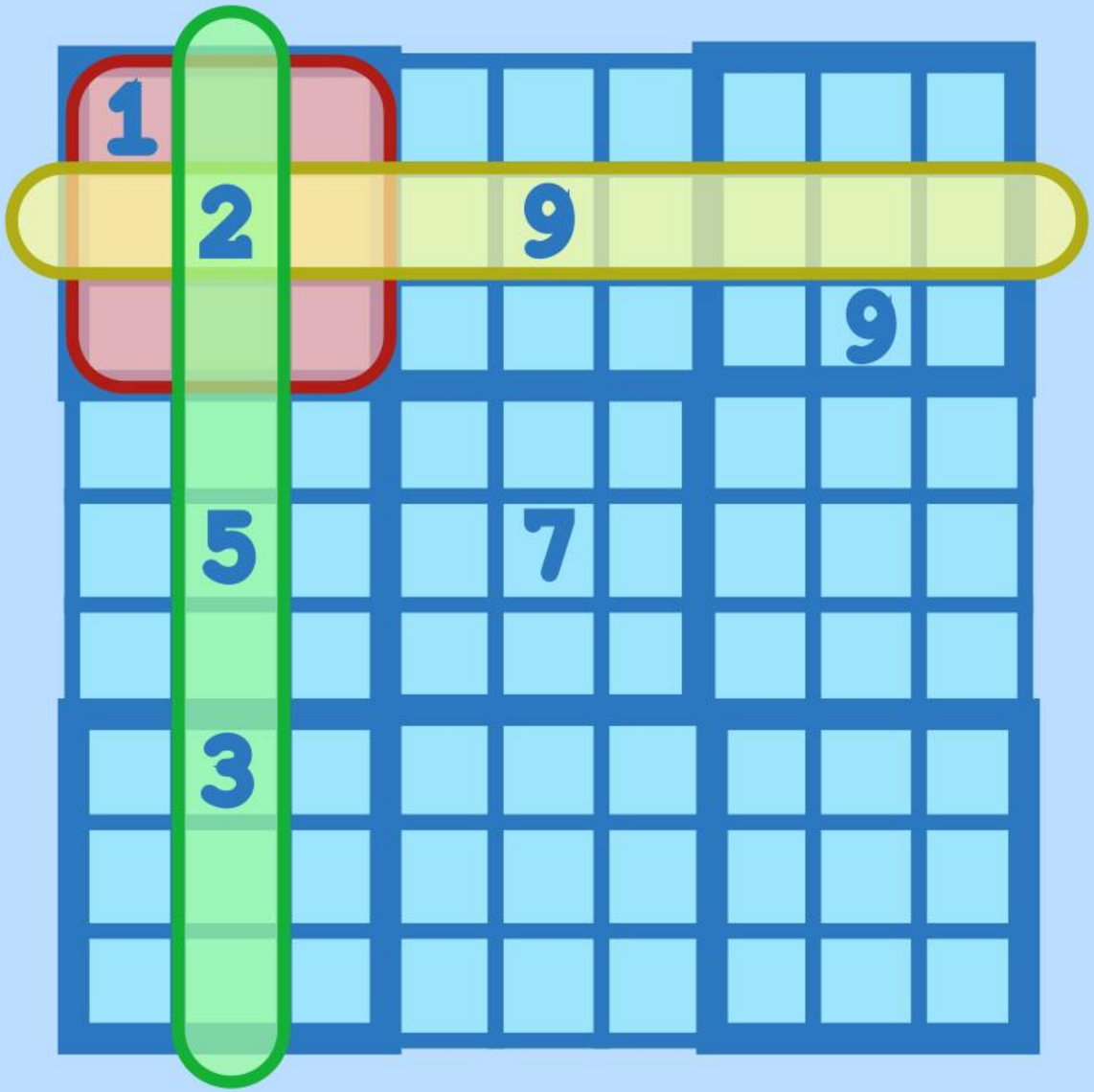
Create **general recursive language** that
-> has **inherent solving capabilities**
-> **can do proofs** and complexity analysis

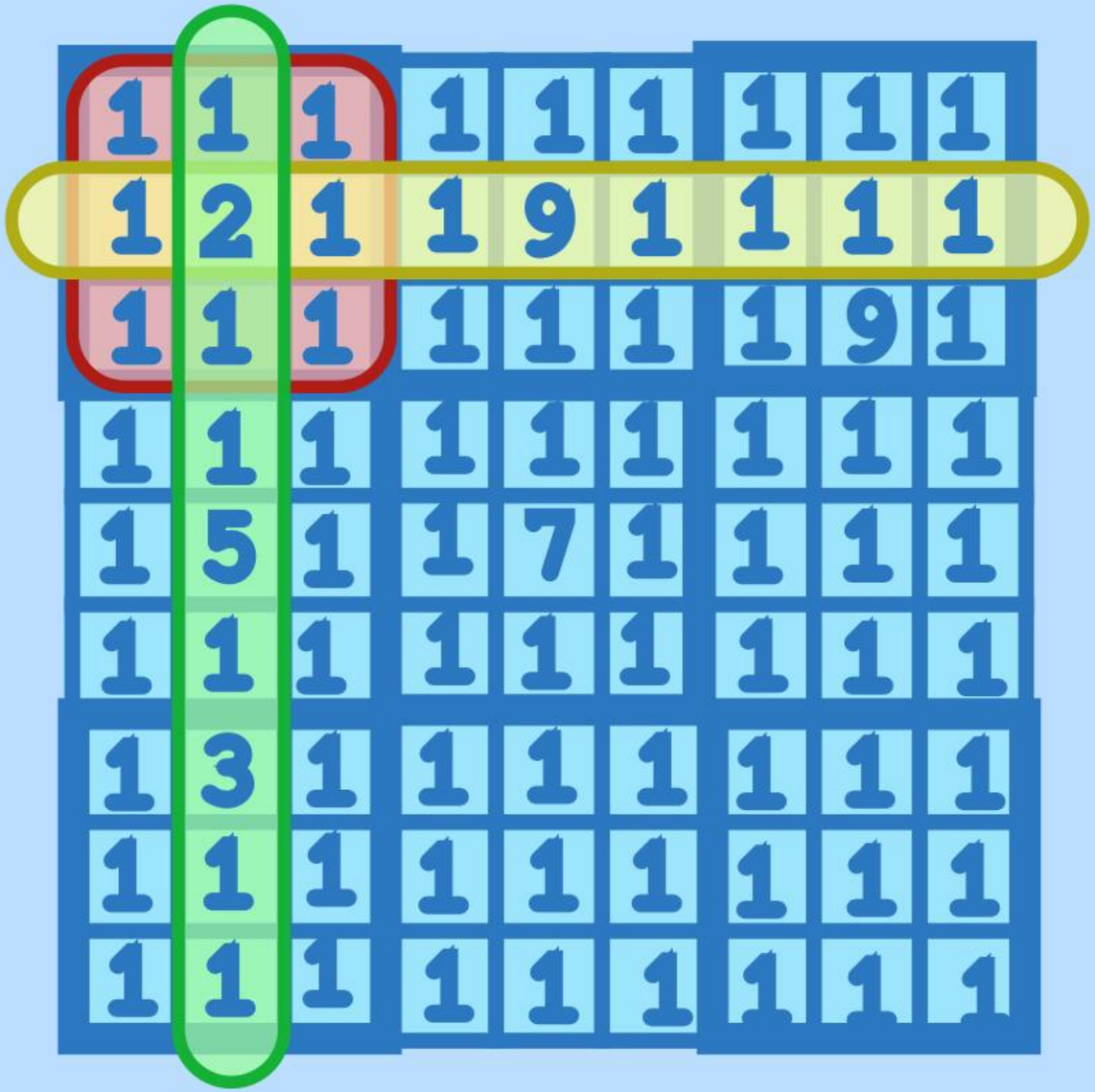
This talk: The concept
(why is this possible / useful)

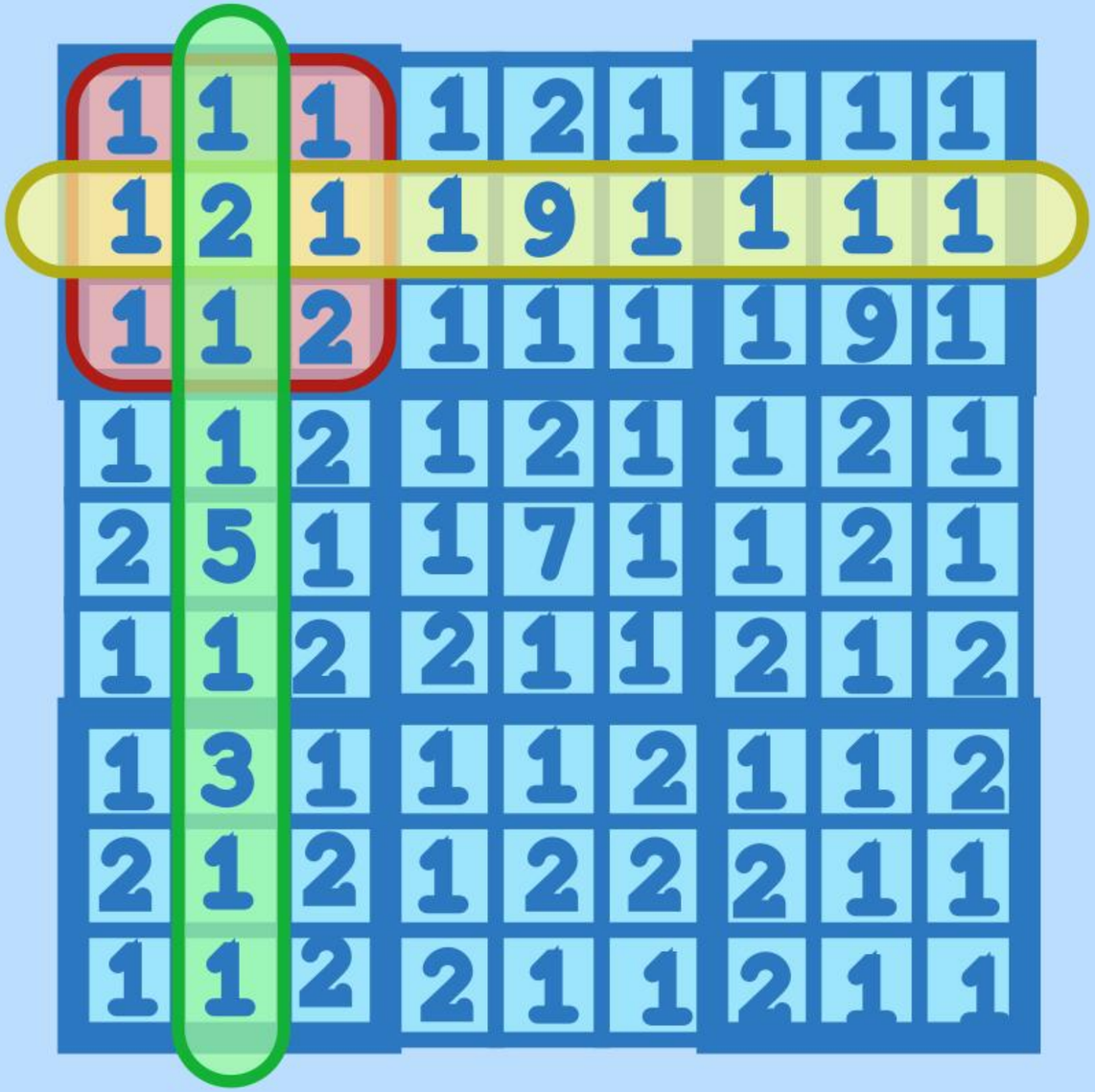
Implementation ongoing...

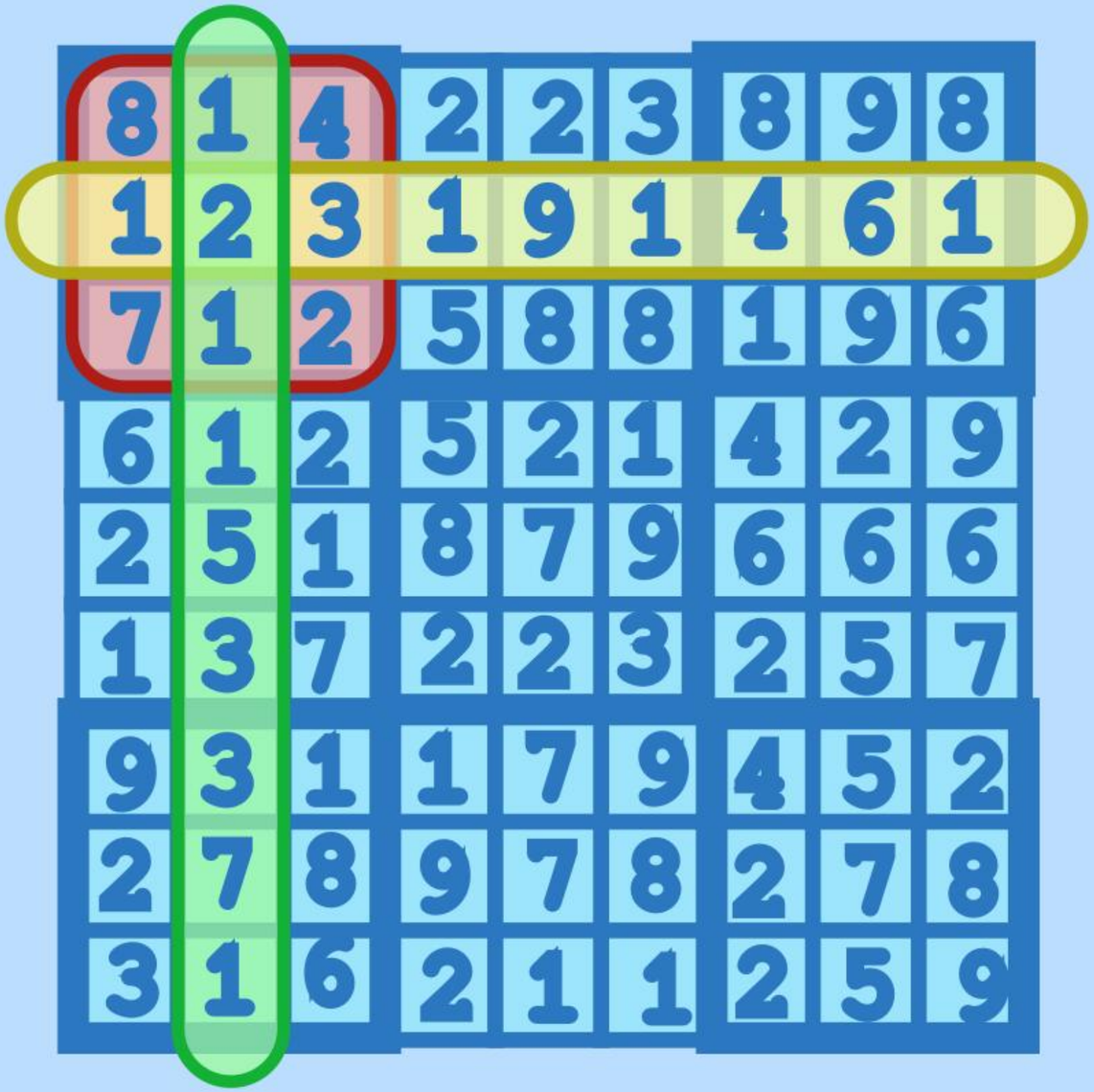
search ≈ solving

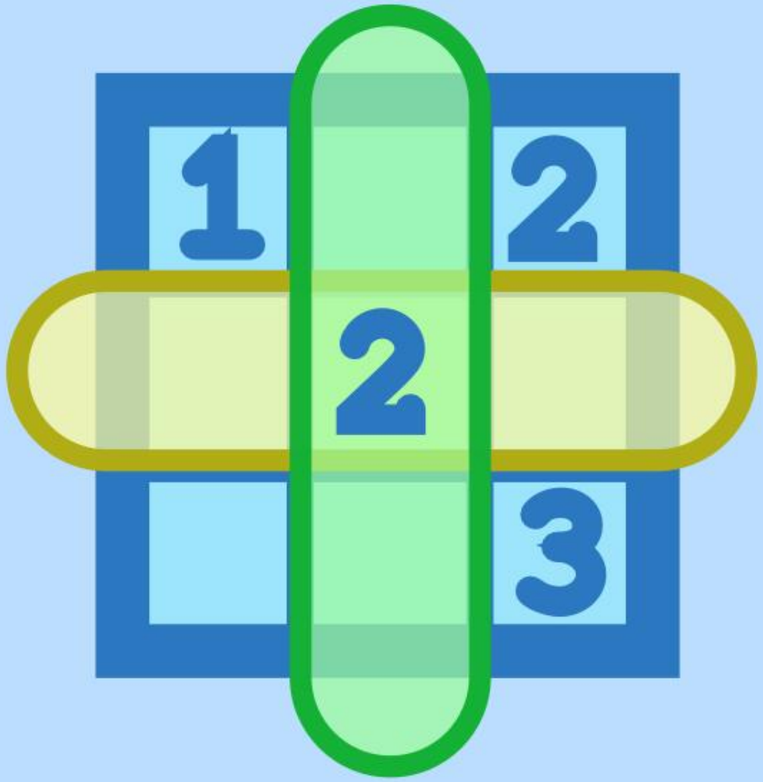
How does **solving** work?

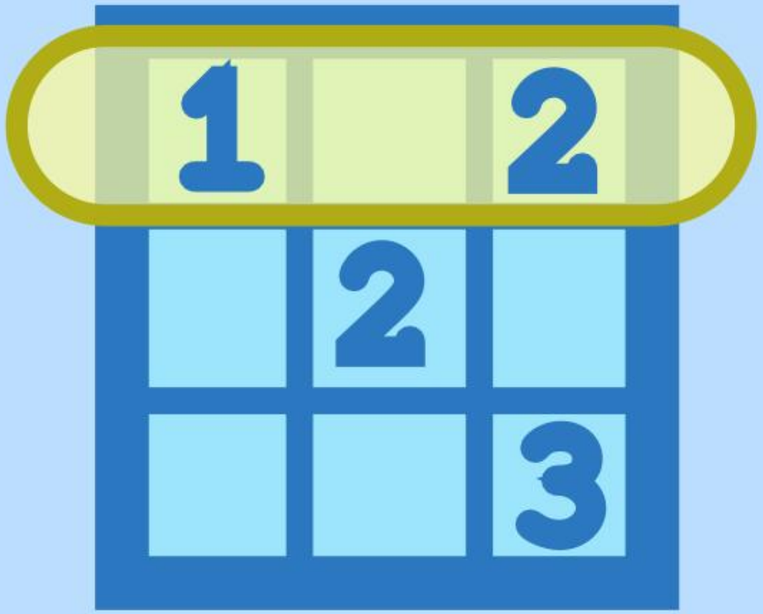

1
2
9
9
5
7
3

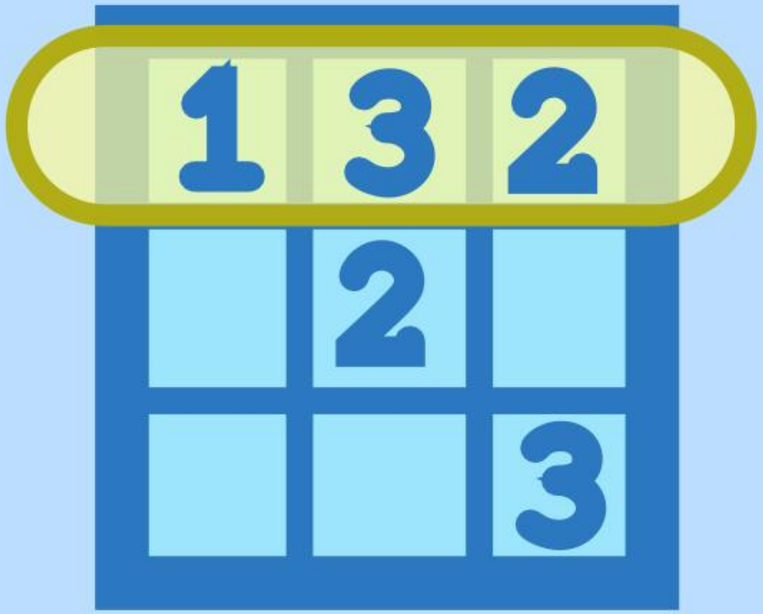

1 1 1 1 1 1 1 1 1
1 2 1 1 9 1 1 1 1
1 1 1 1 1 1 1 9 1
1 1 1 1 1 1 1 1 1
1 5 1 1 7 1 1 1 1
1 1 1 1 1 1 1 1 1
1 3 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1


1 1 1 1 2 1 1 1 1
1 2 1 1 9 1 1 1 1
1 1 2 1 1 1 1 9 1
1 1 2 1 2 1 1 2 1
2 5 1 1 7 1 1 2 1
1 1 2 2 1 1 2 1 2
1 3 1 1 1 2 1 1 2
2 1 2 1 2 2 2 1 1
1 1 2 2 1 1 2 1 1

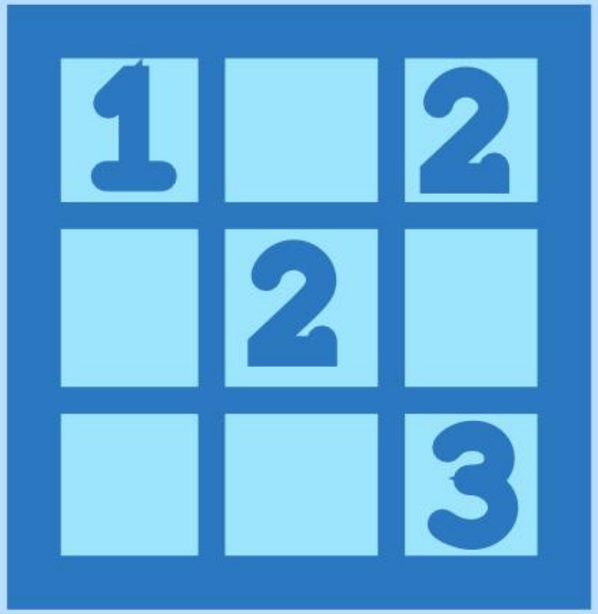

8 1 4 2 2 3 8 9 8
1 2 3 1 9 1 4 6 1
7 1 2 5 8 8 1 9 6
6 1 2 5 2 1 4 2 9
2 5 1 8 7 9 6 6 6
1 3 7 2 2 3 2 5 7
9 3 1 1 7 9 4 5 2
2 7 8 9 7 8 2 7 8
3 1 6 2 1 1 2 5 9

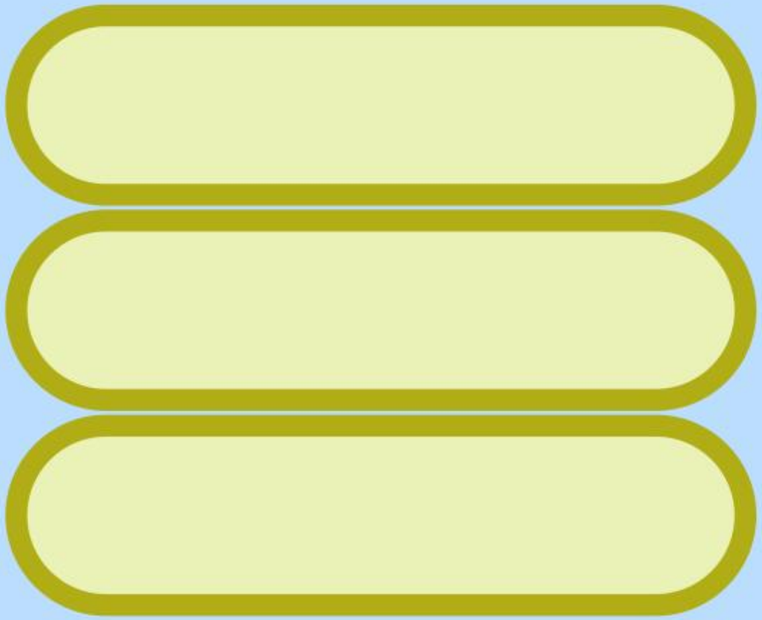

1
2
2
3

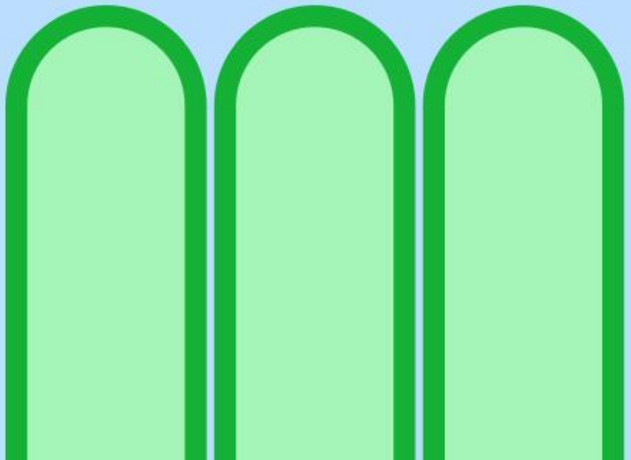

1
2
2
3


1 3 2
2
3

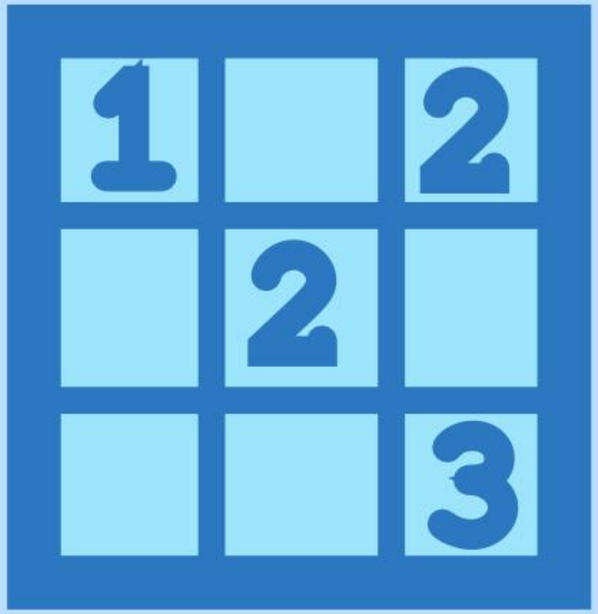
Unit Propagation

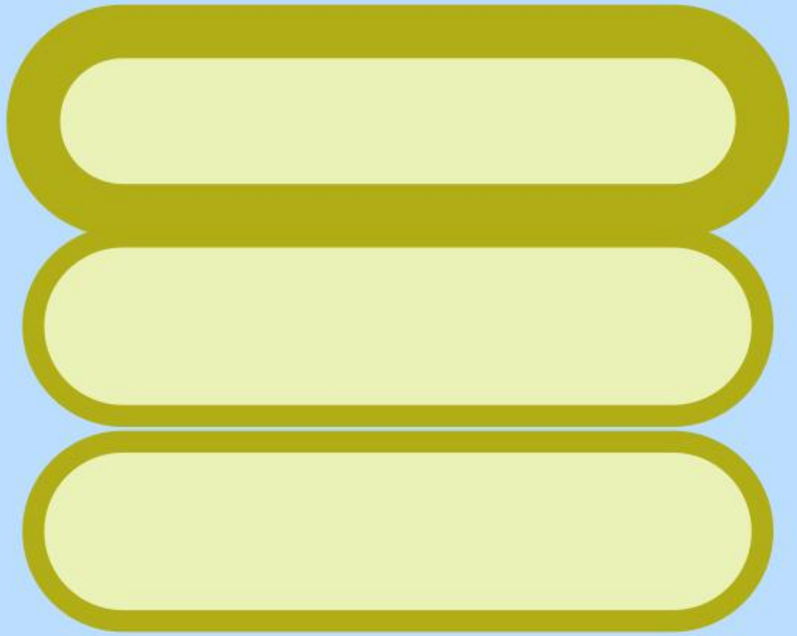


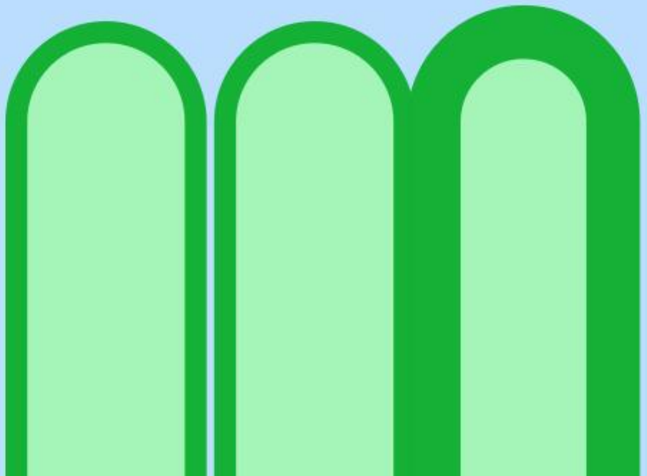




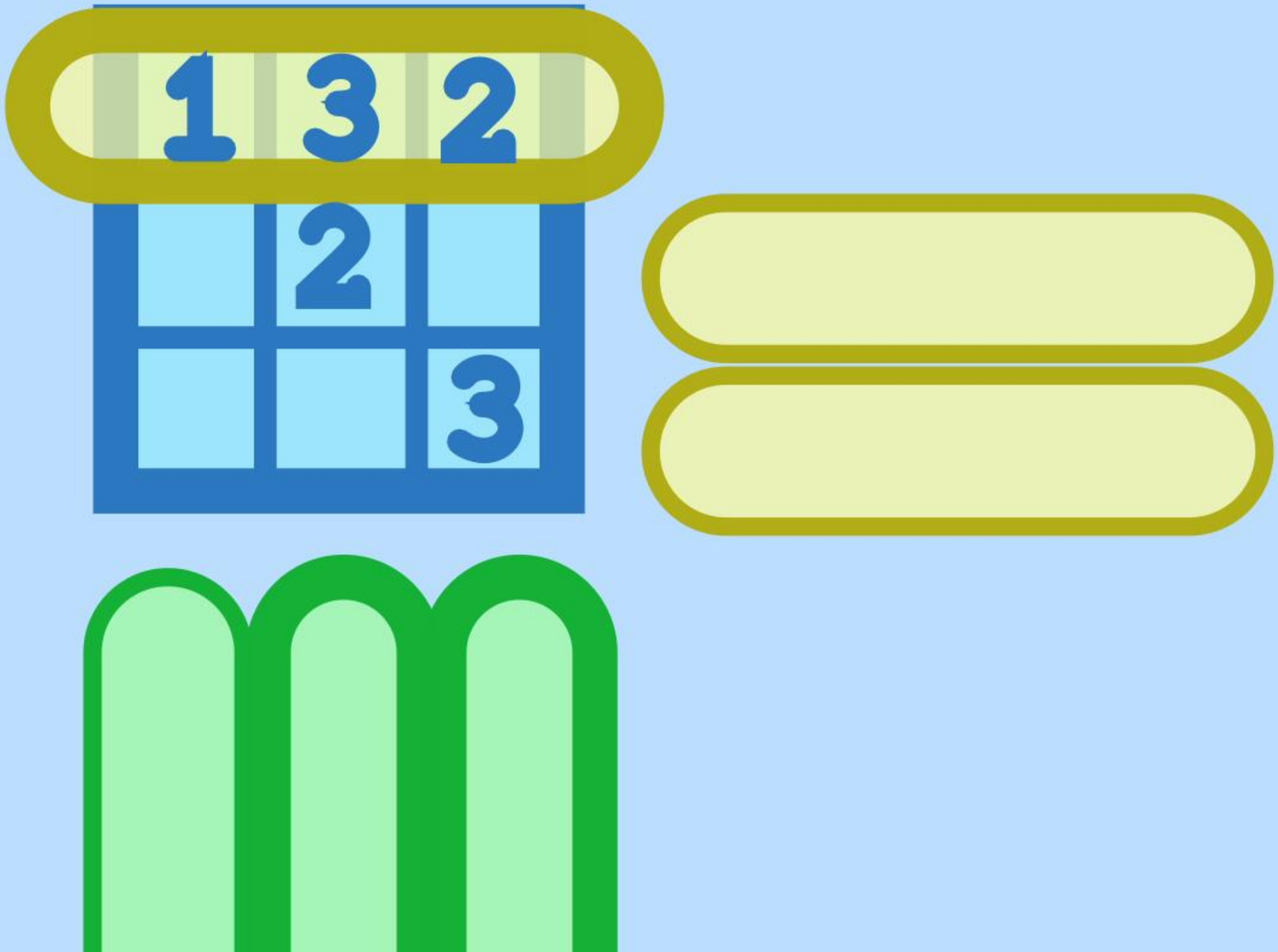
Unit Propagation



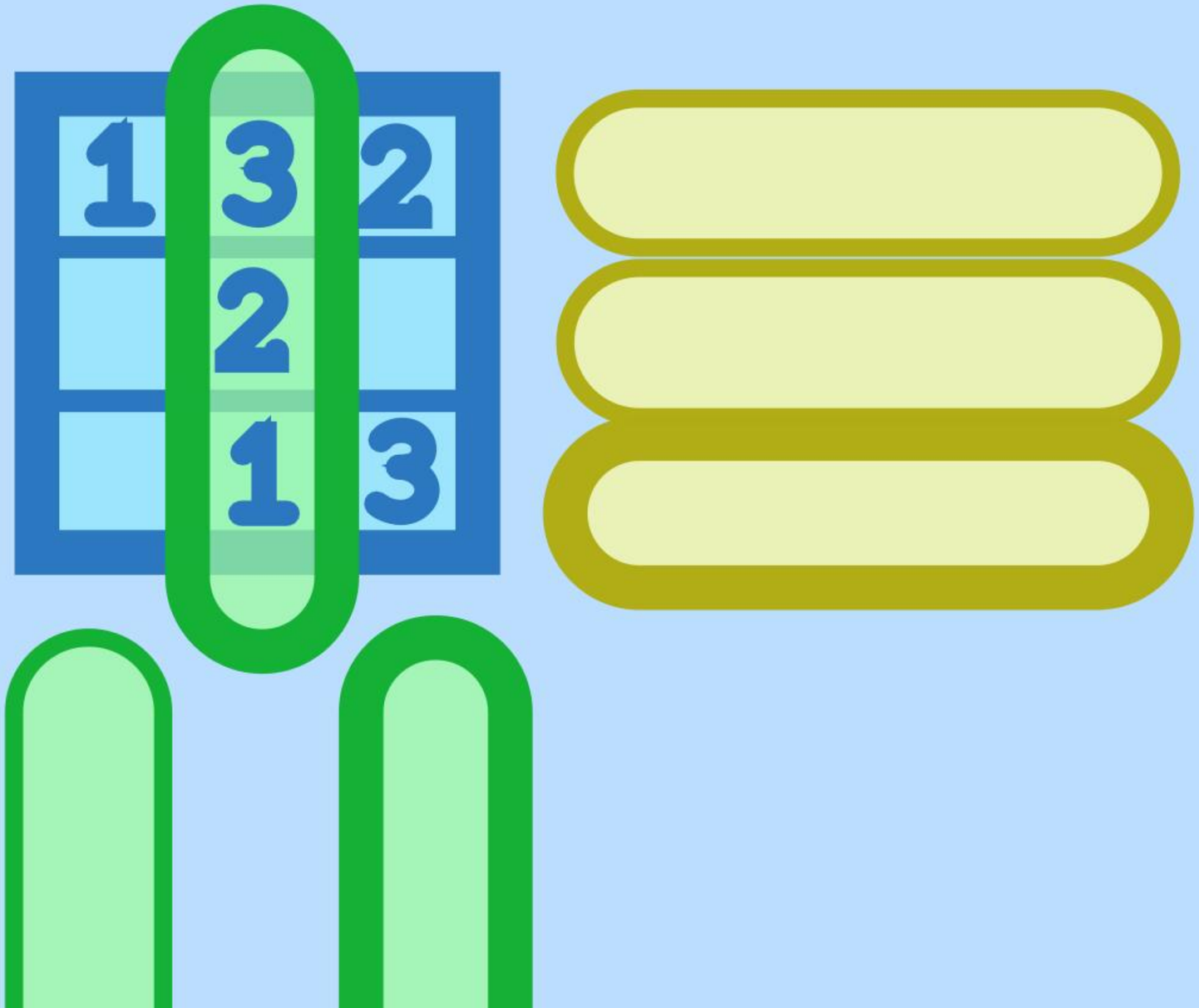




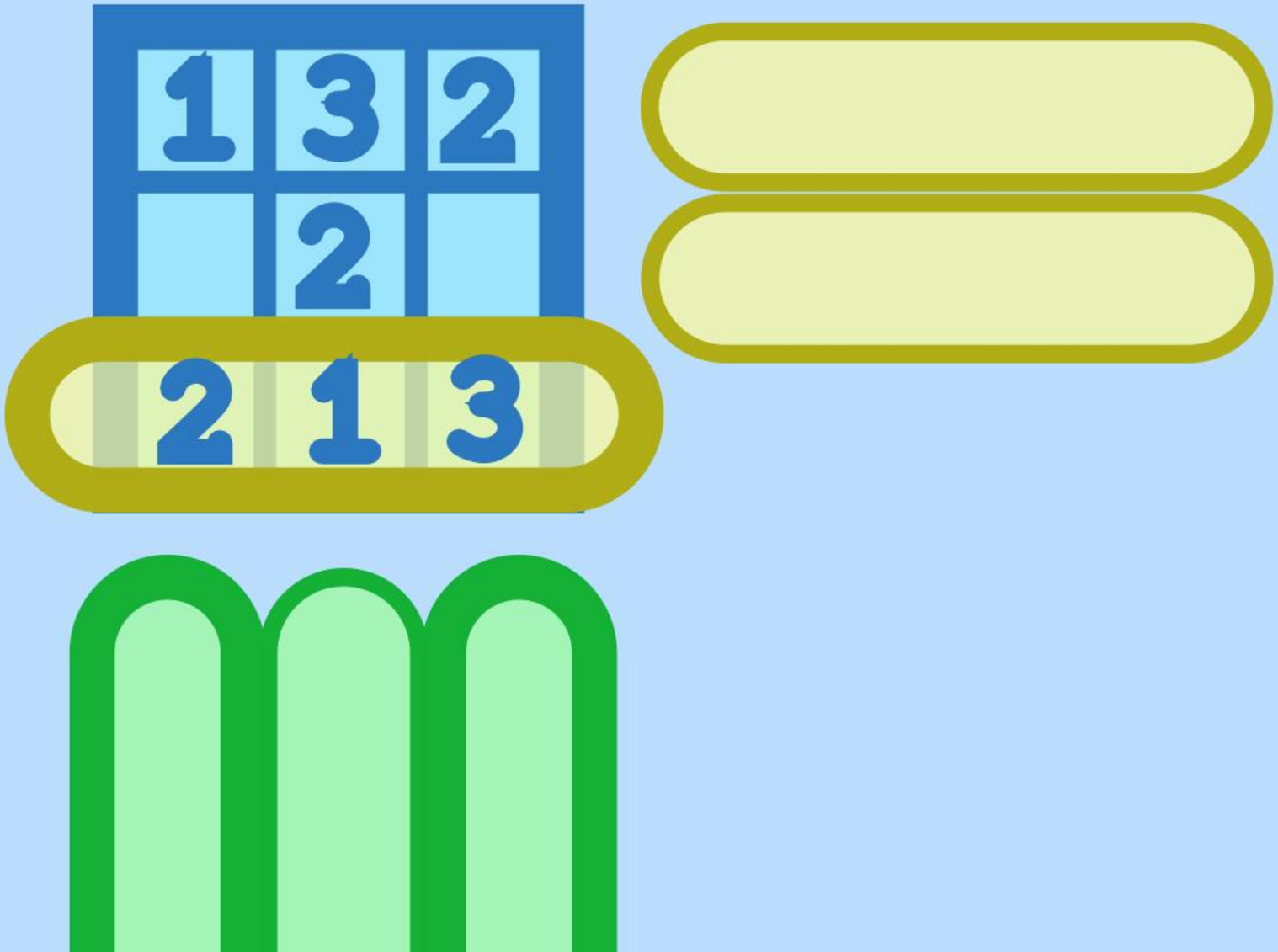
Unit Propagation



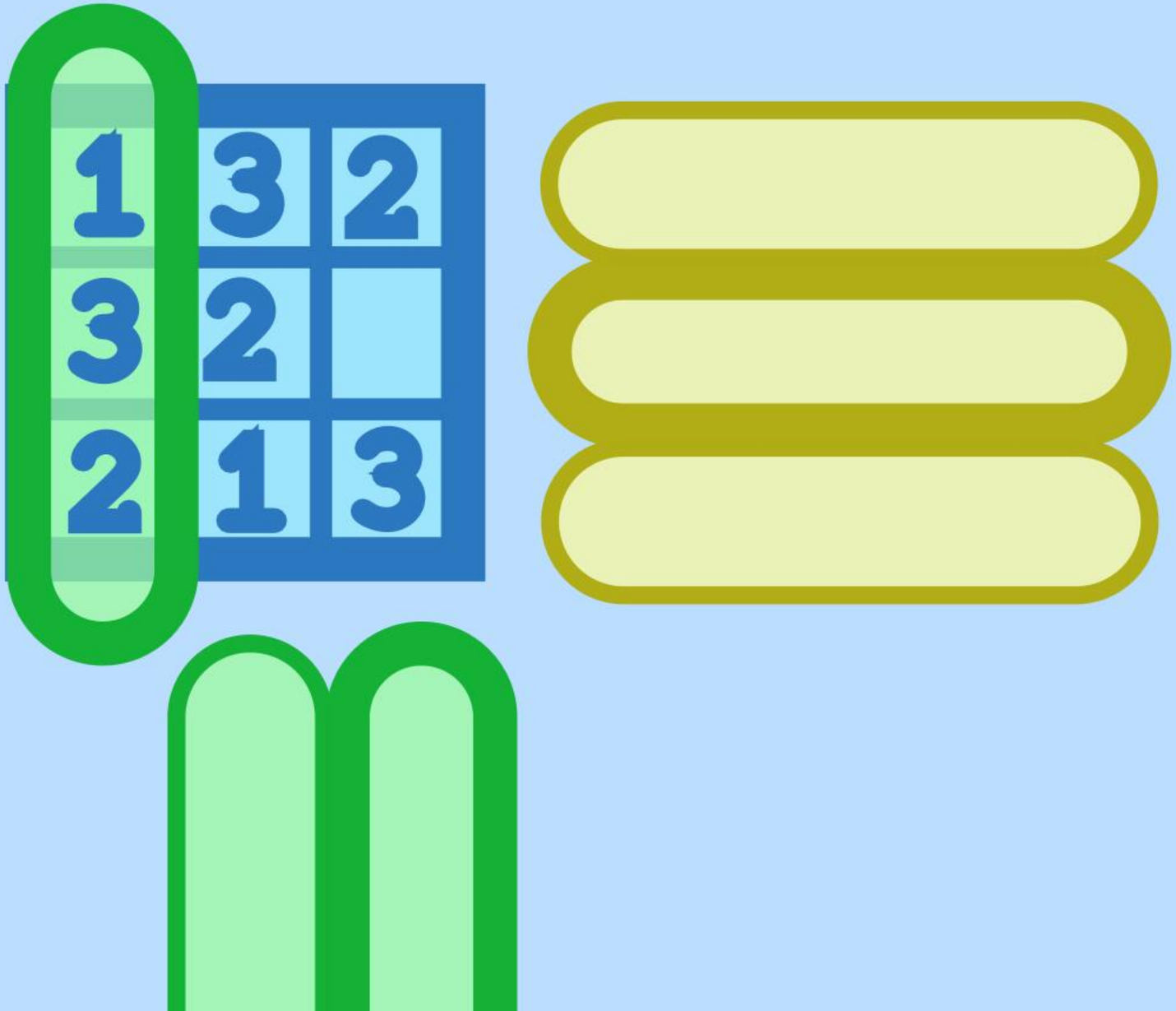
Unit Propagation



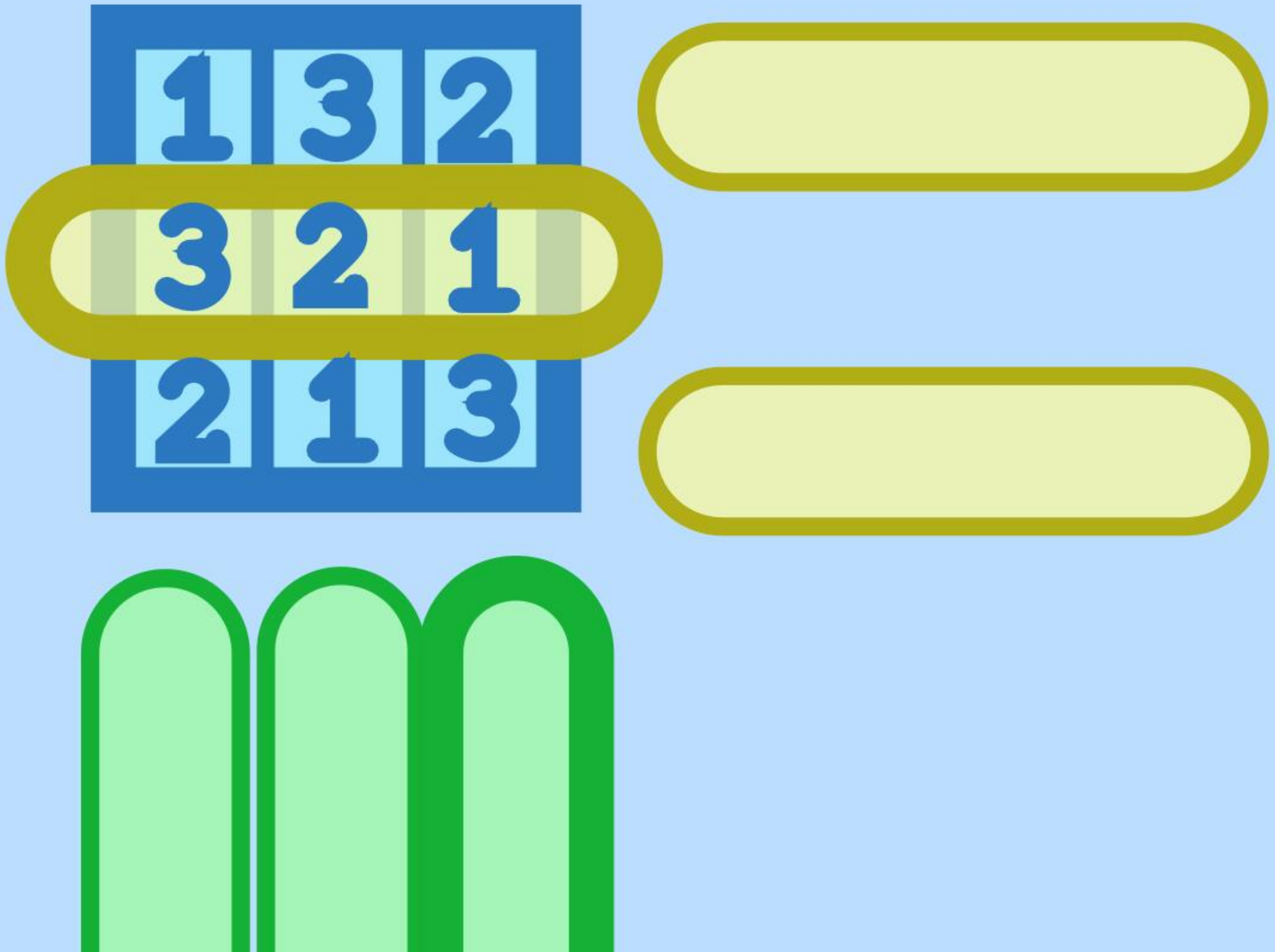
Unit Propagation



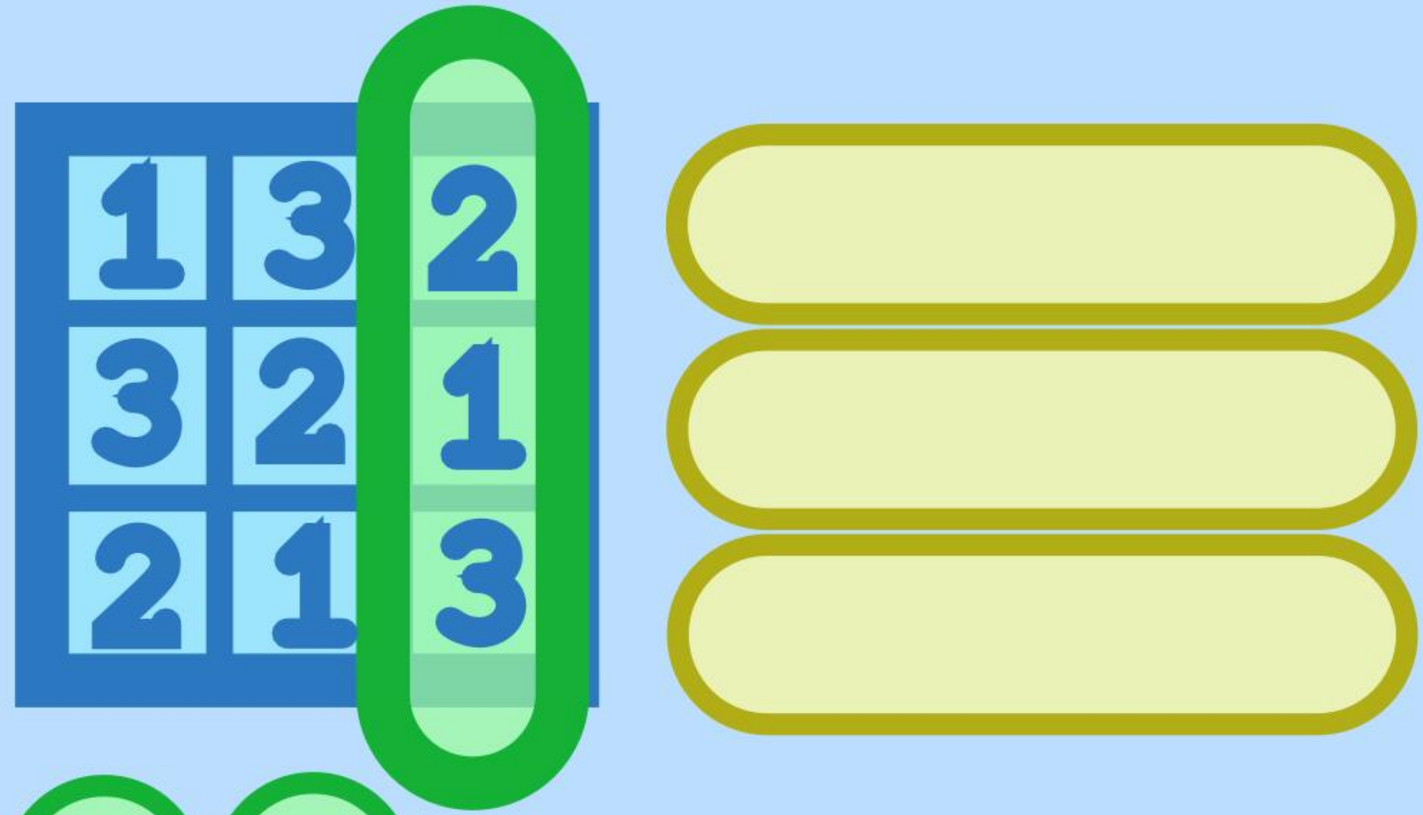
Unit Propagation



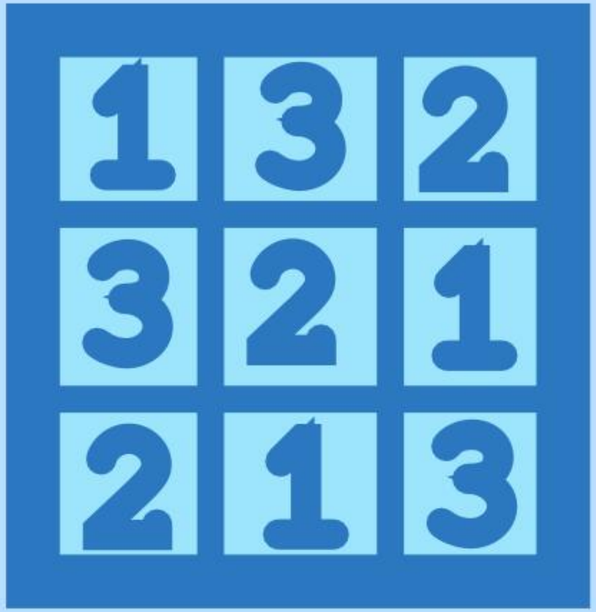
Unit Propagation

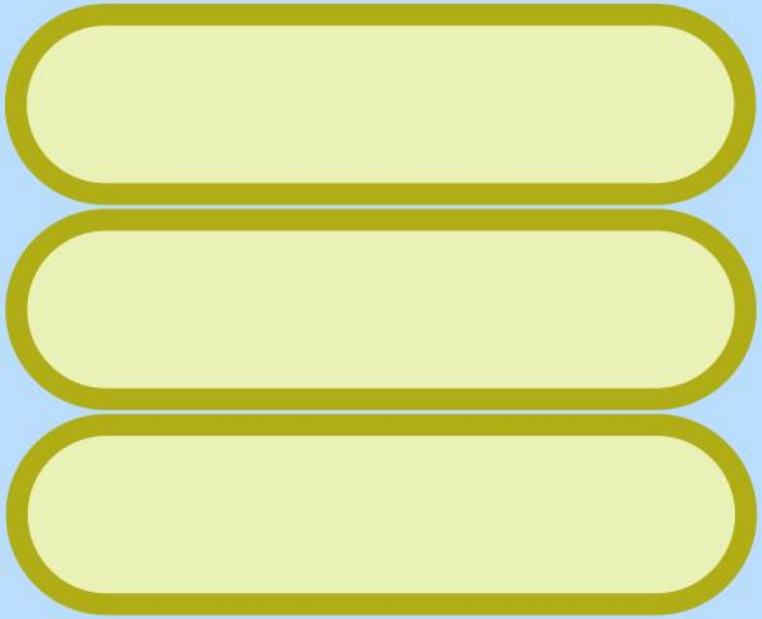


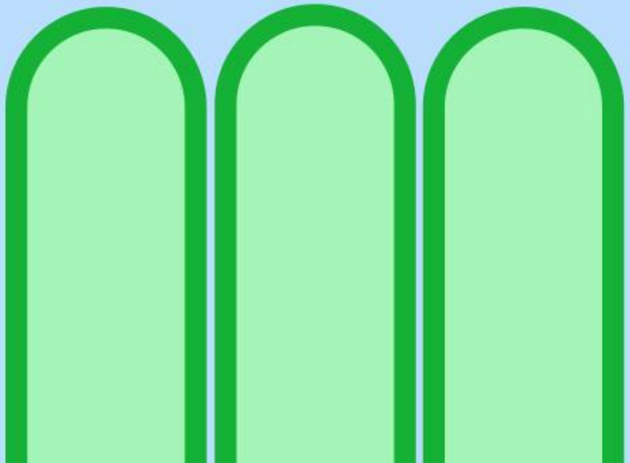
Unit Propagation



Unit Propagation





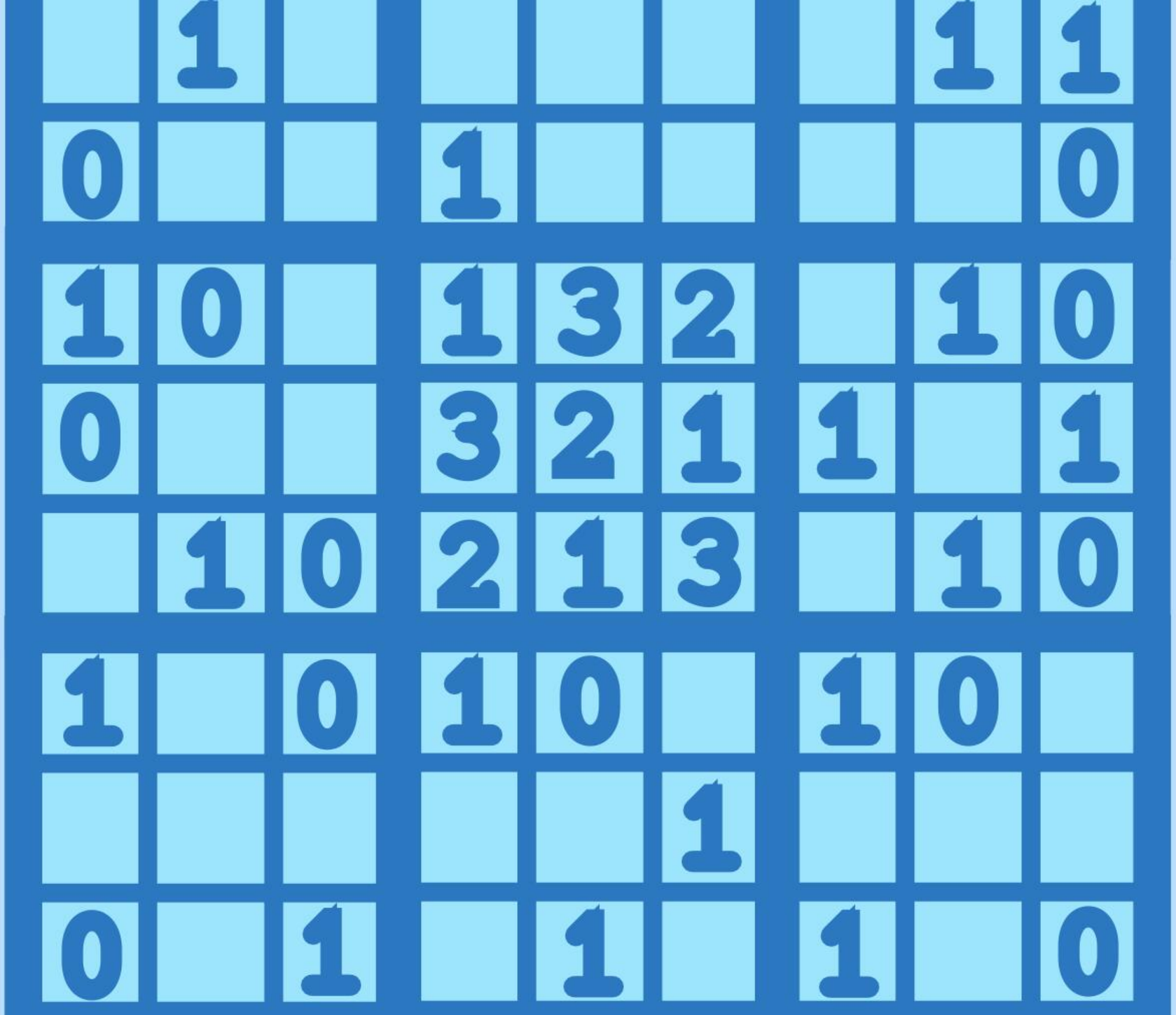


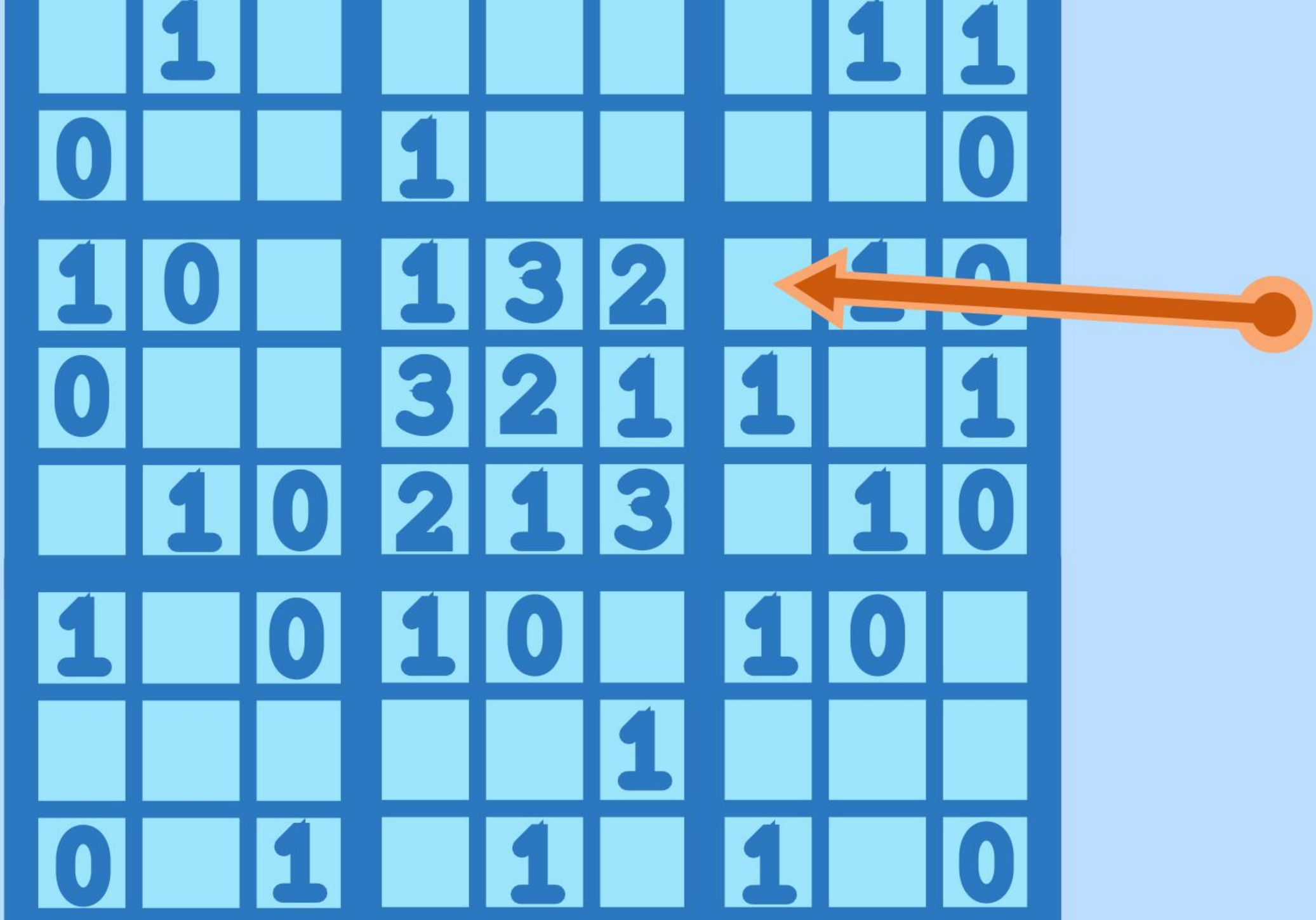
Unit propagation runs until either there are **no more units to propagate** or there has been a **conflict**.

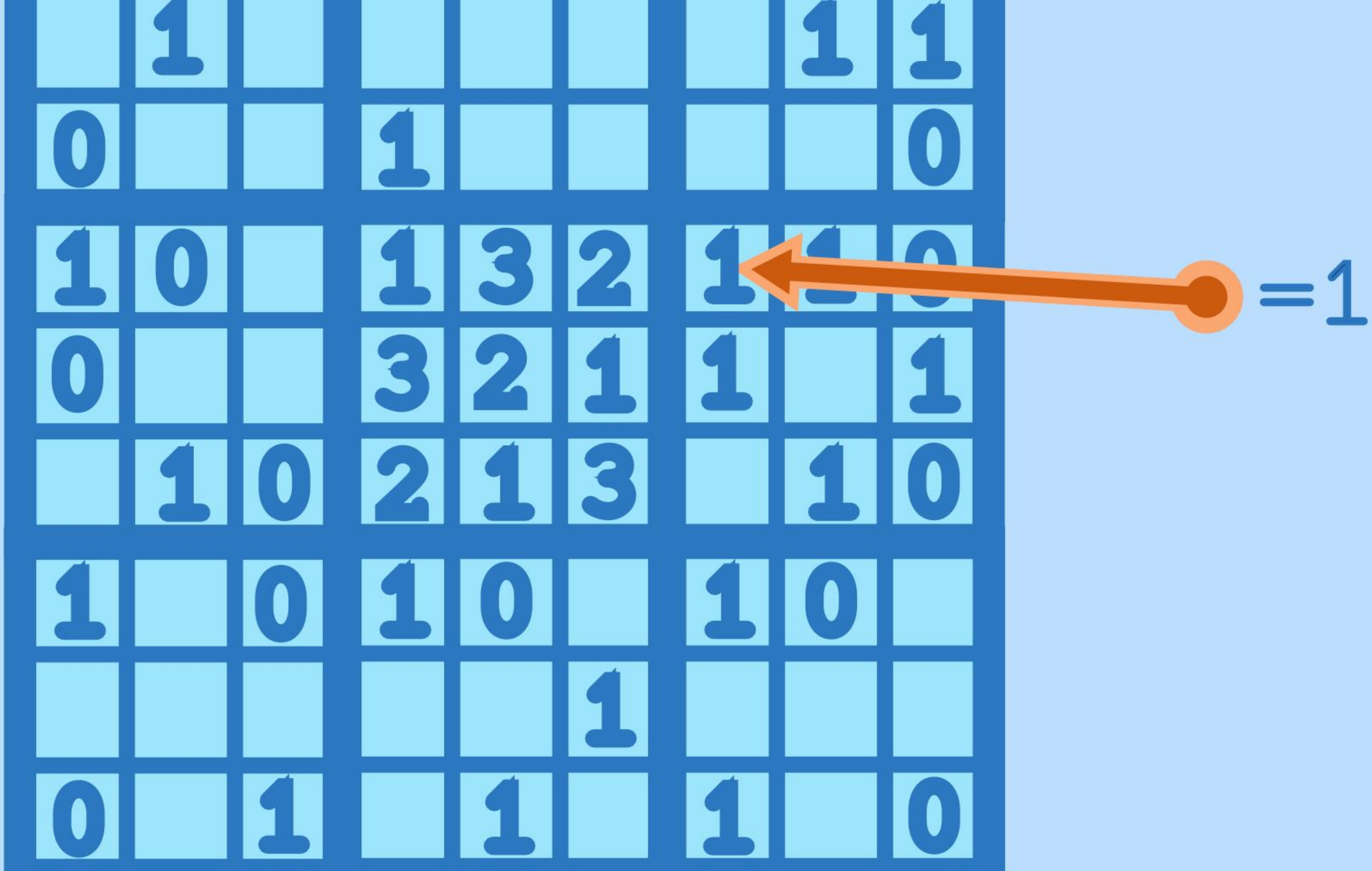
A **conflict** arises iff a variable is assigned **two conflicting values**.

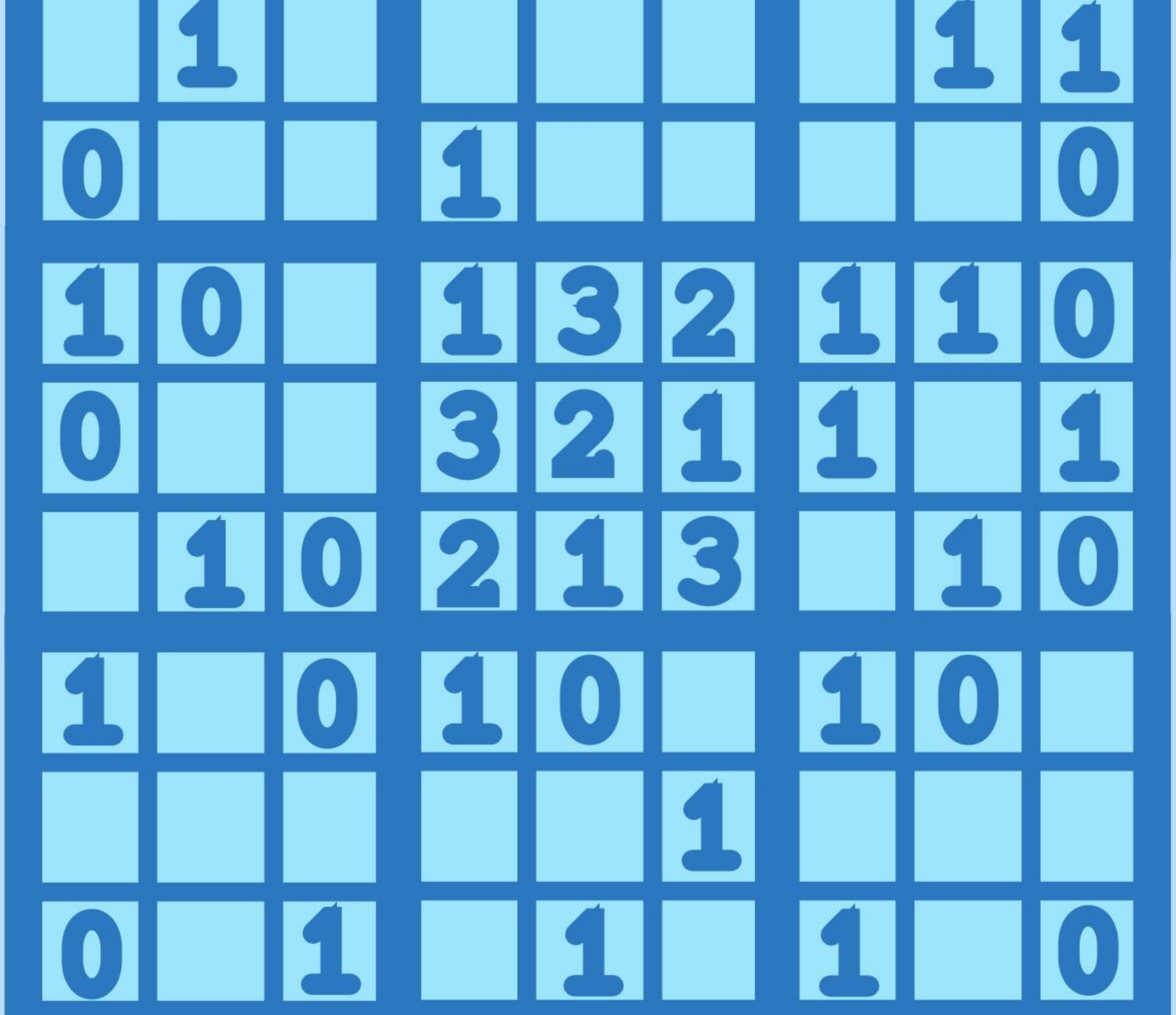
The bigger picture...

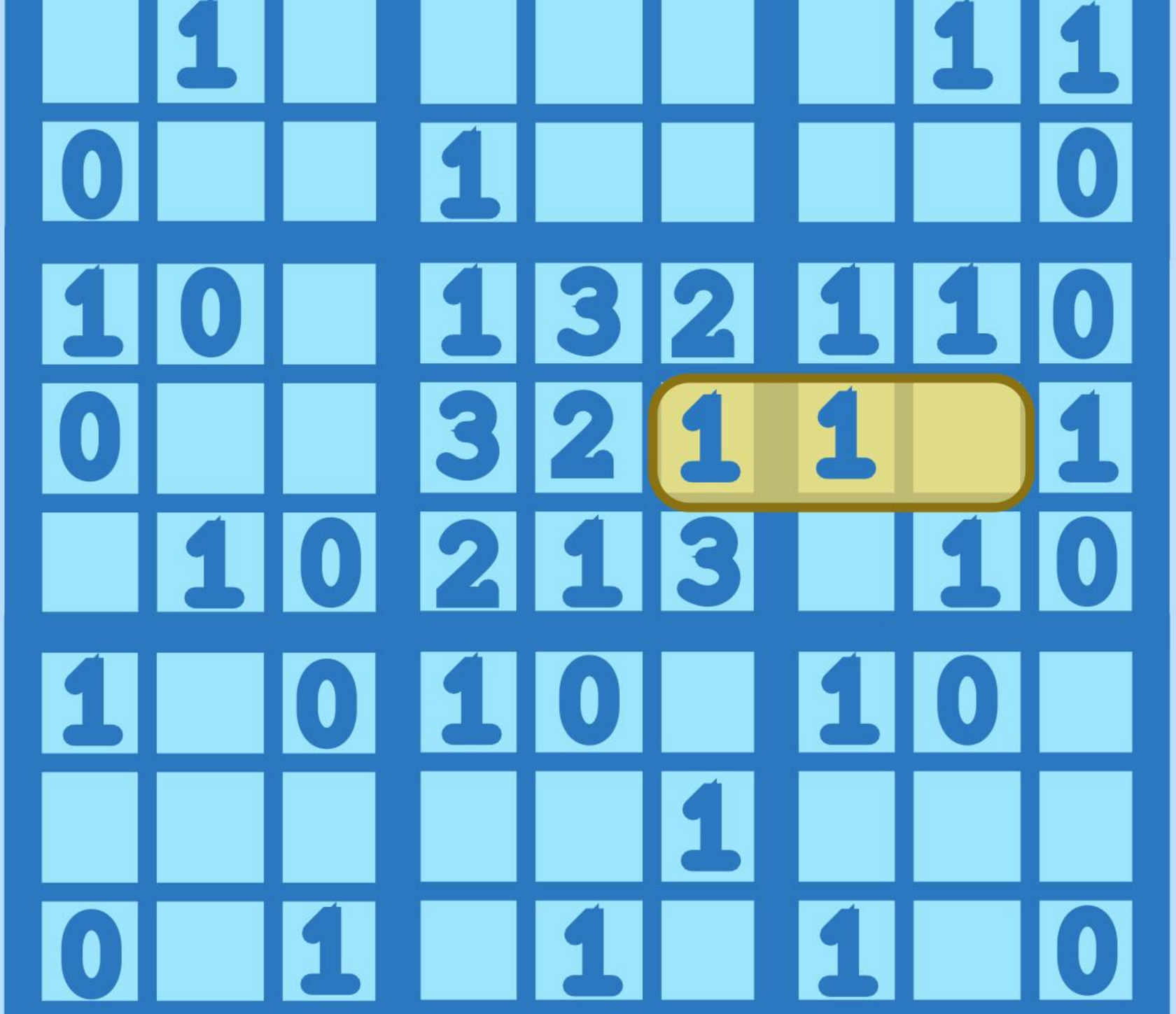
Why is **search** such a ubiquitous problem in computer science?


1 1 1
0 1 0
1 0 1 3 2 1 0
0 3 2 1 1 1
1 0 2 1 3 1 0
1 0 1 0 1 0
1
0 1 1 1 0

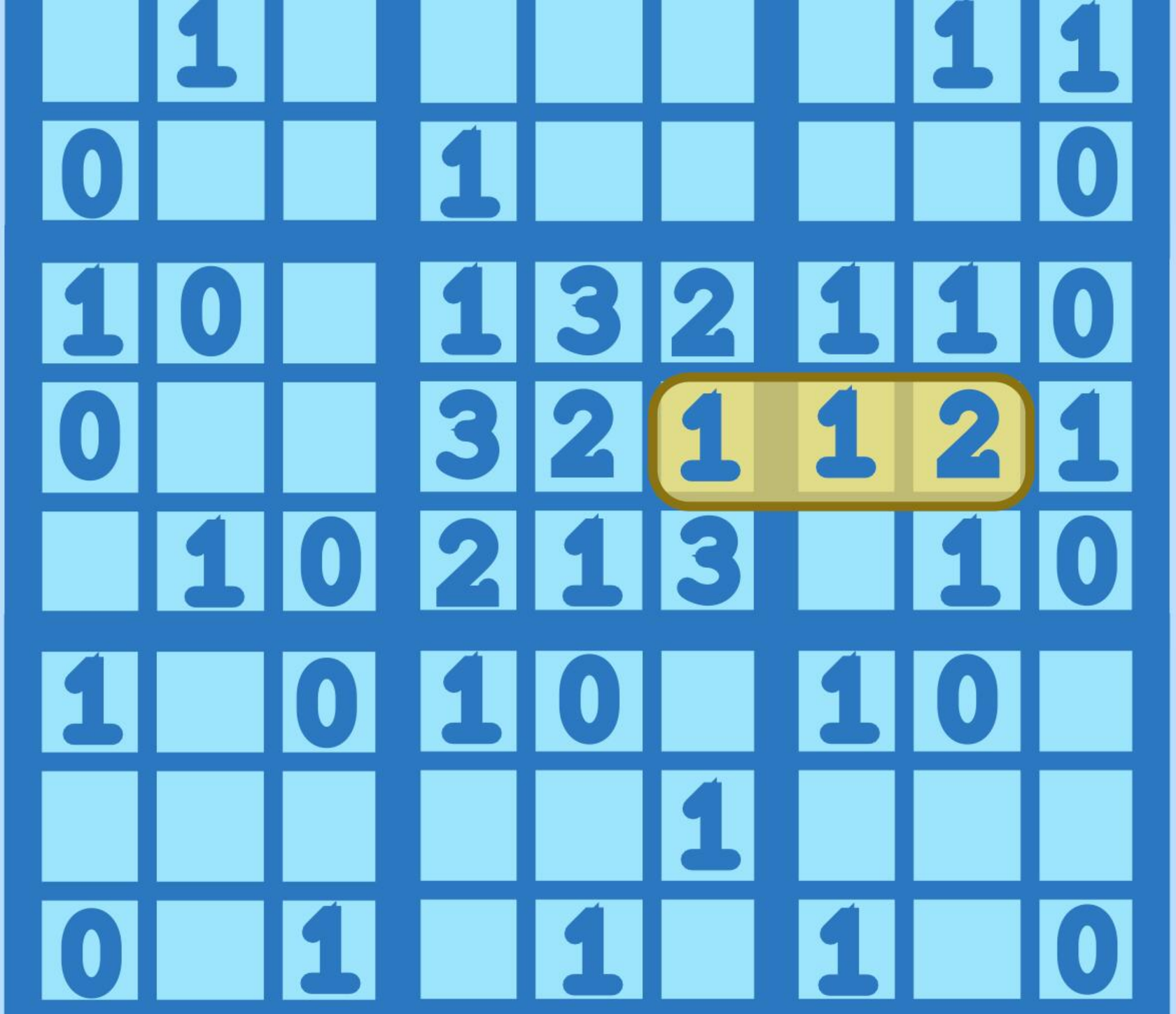



=1

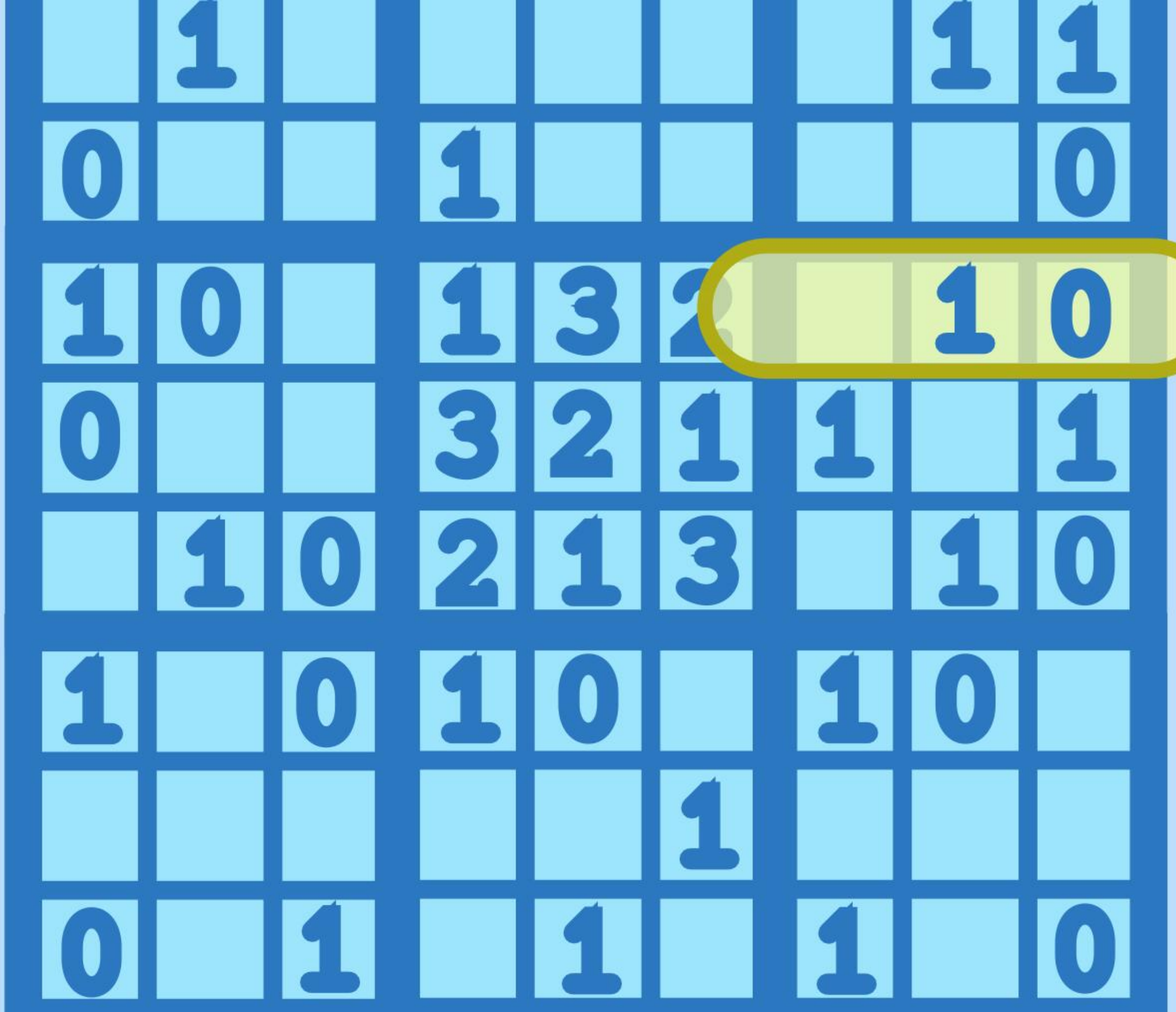

1 1 1
0 1 0
1 0 1 3 2 1 1 0
0 3 2 1 1 1
1 0 2 1 3 1 0
1 0 1 0 1 0
1
0 1 1 1 0

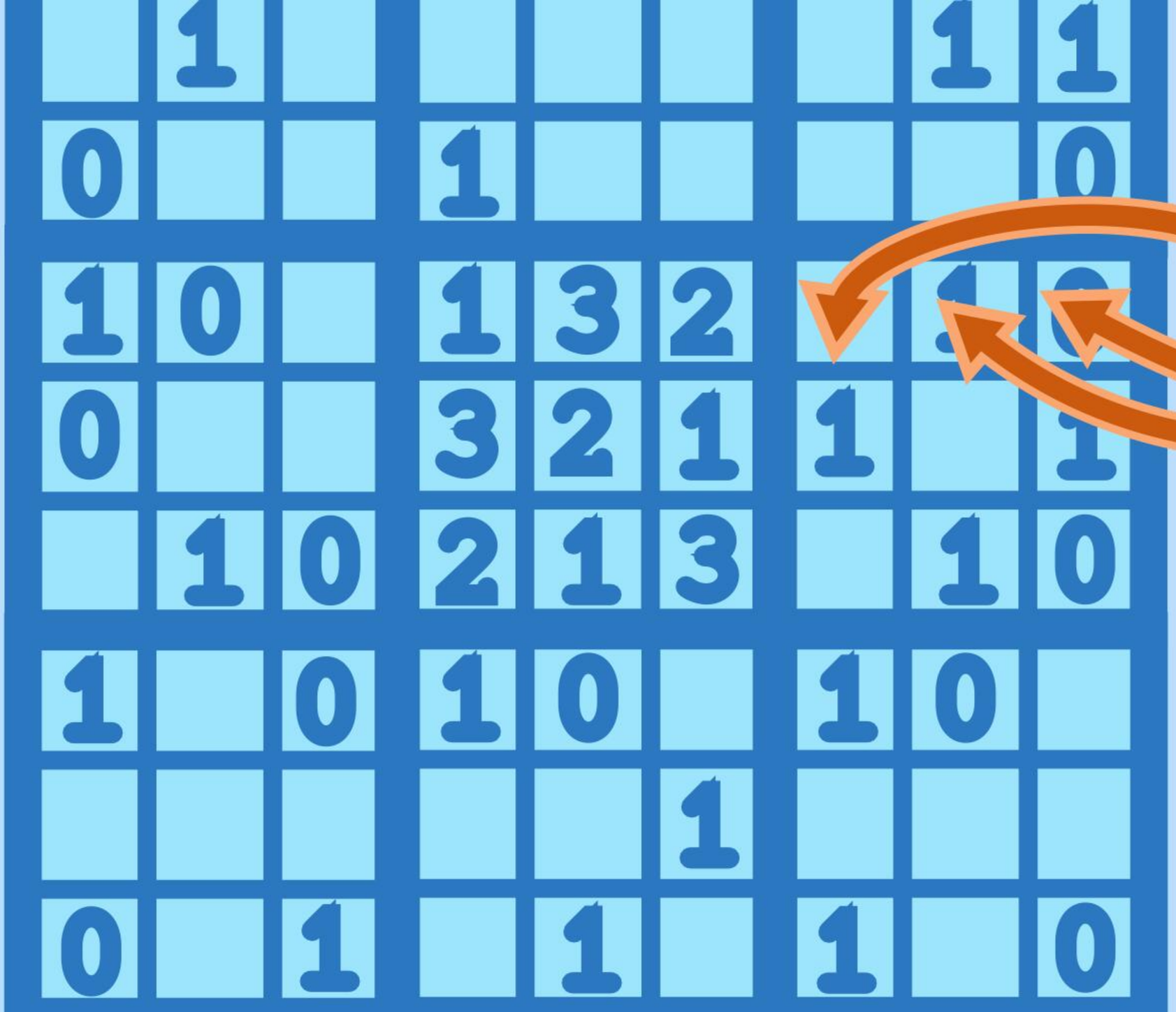


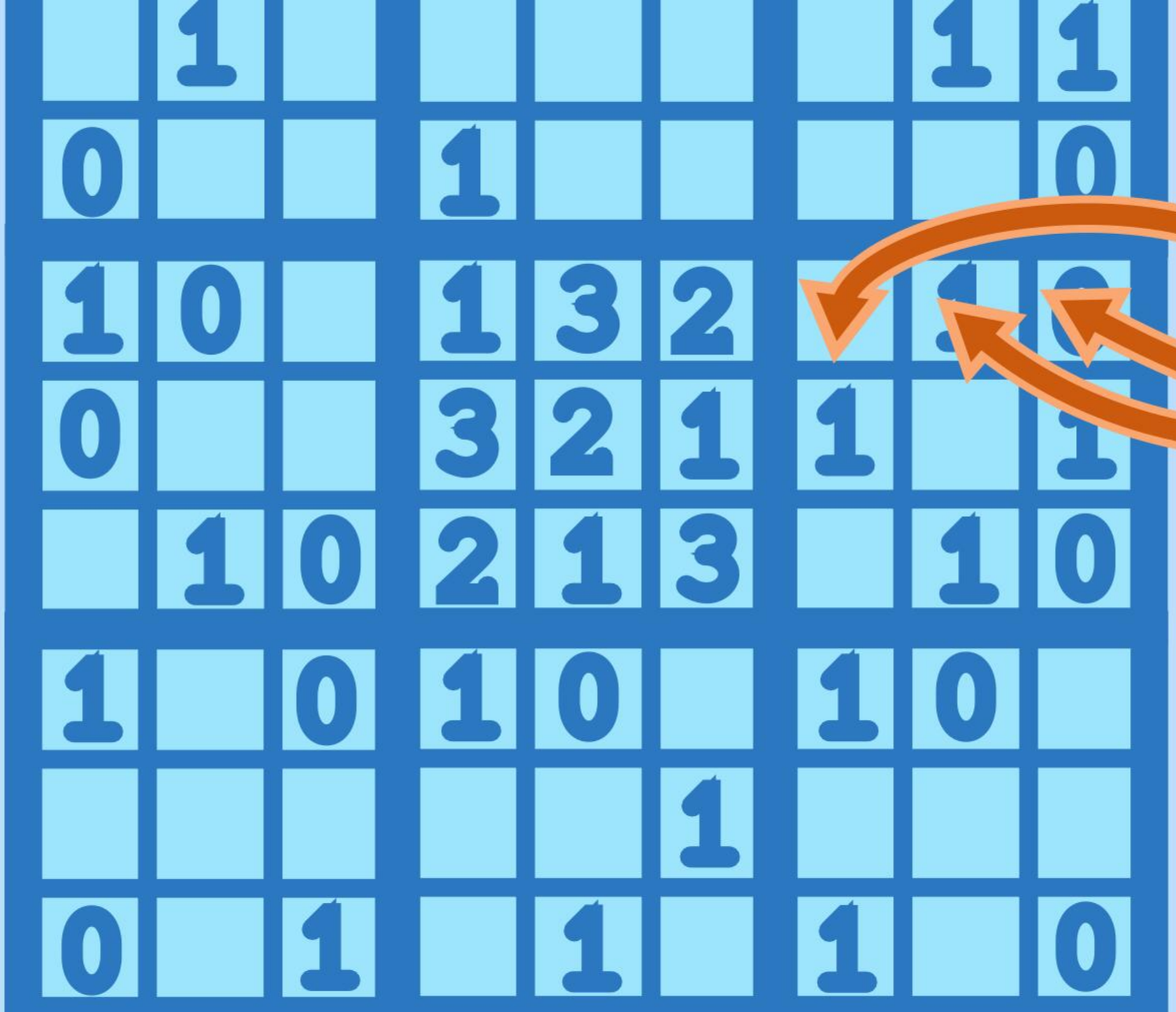

```
f a b = a+b
```

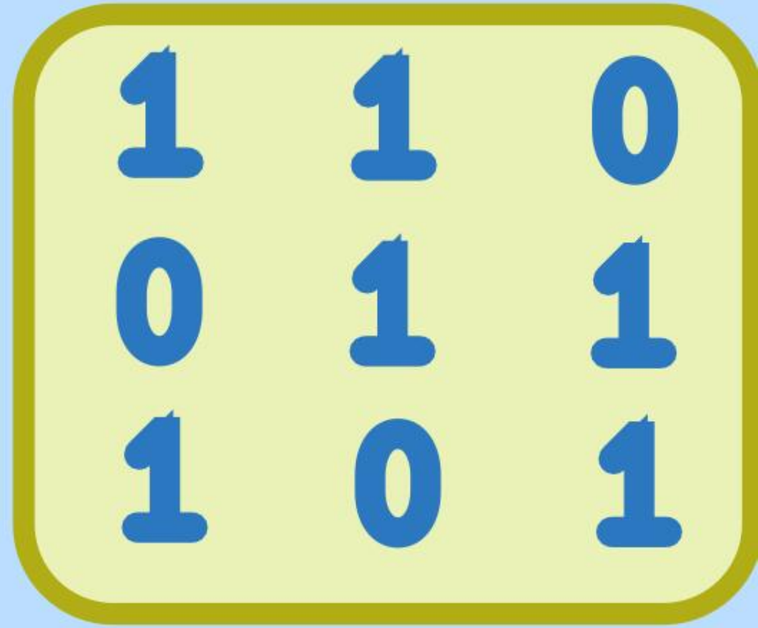


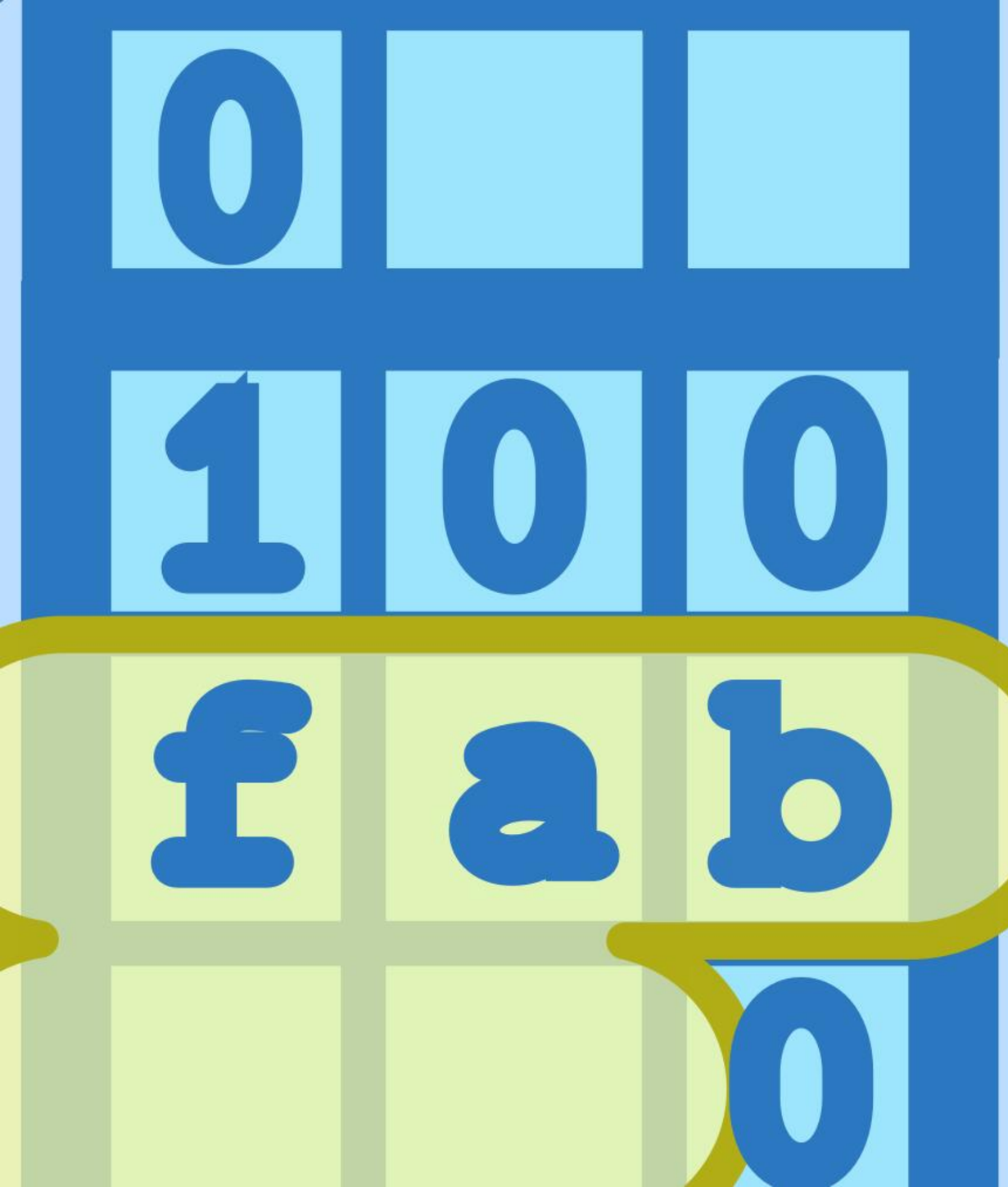

```
f a b = a+b
```


1 1 1
0 1 0
1 0 1 3 2 1 0
0 3 2 1 1 1
1 0 2 1 3 1 0
1 0 1 0 1 0
1
0 1 1 1 0


1 1 1
0 1 0
1 0 1 3 2 1 0
0 3 2 1 1 1
1 0 2 1 3 1 0
1 0 1 0 1 0
1
0 1 1 1 0


1 1 1
0 1 0
1 0 2 1 3 2 1 0
0 3 2 1 1 1
1 0 2 1 3 1 0
1 0 1 0 1 0
1
0 1 1 1 0


1 1 0
0 1 1
1 0 1

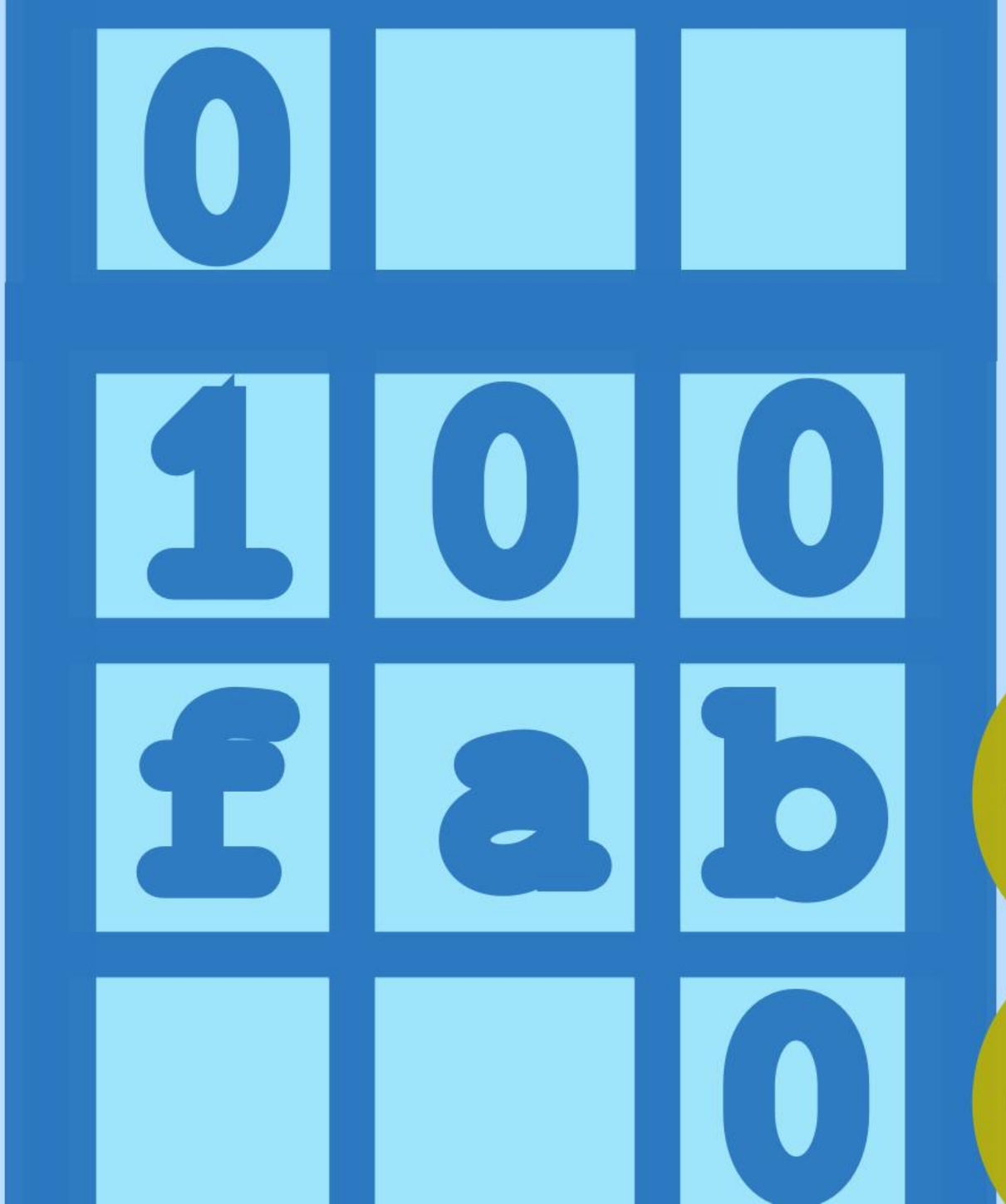


f c d =>

a a

f a b =>

b a

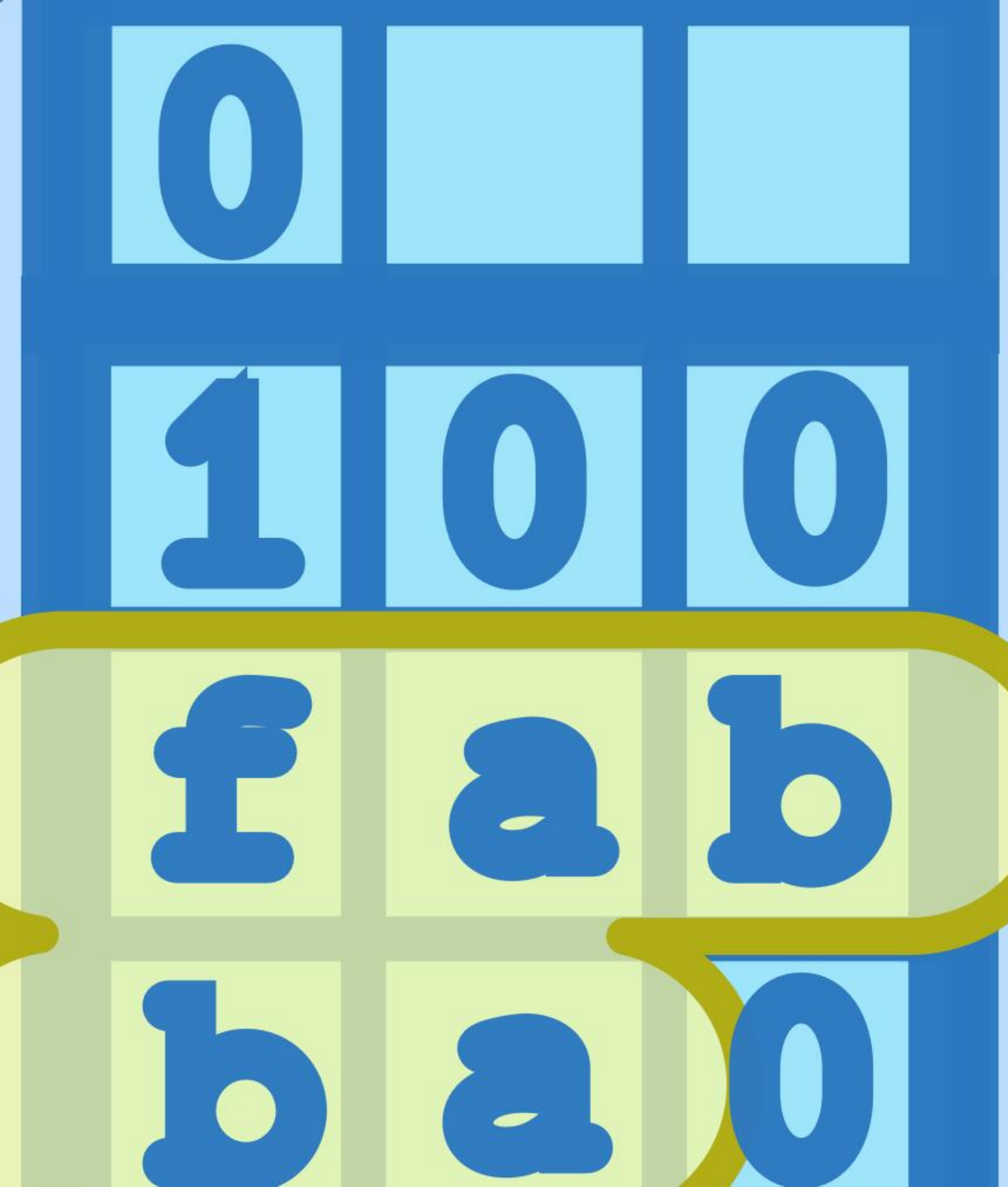


f c d =>

a a

f a b =>

b a



f c d =>

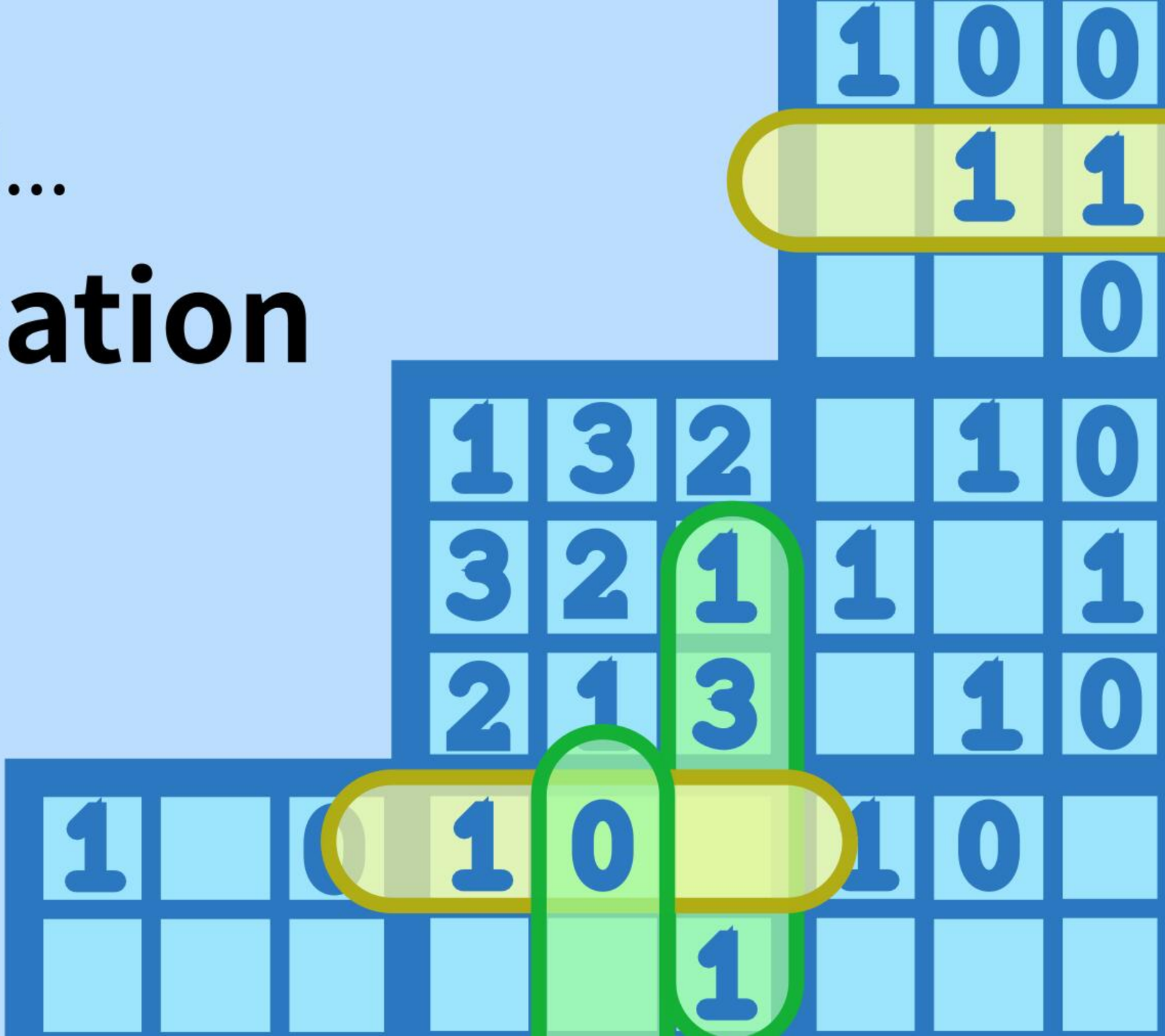
a a

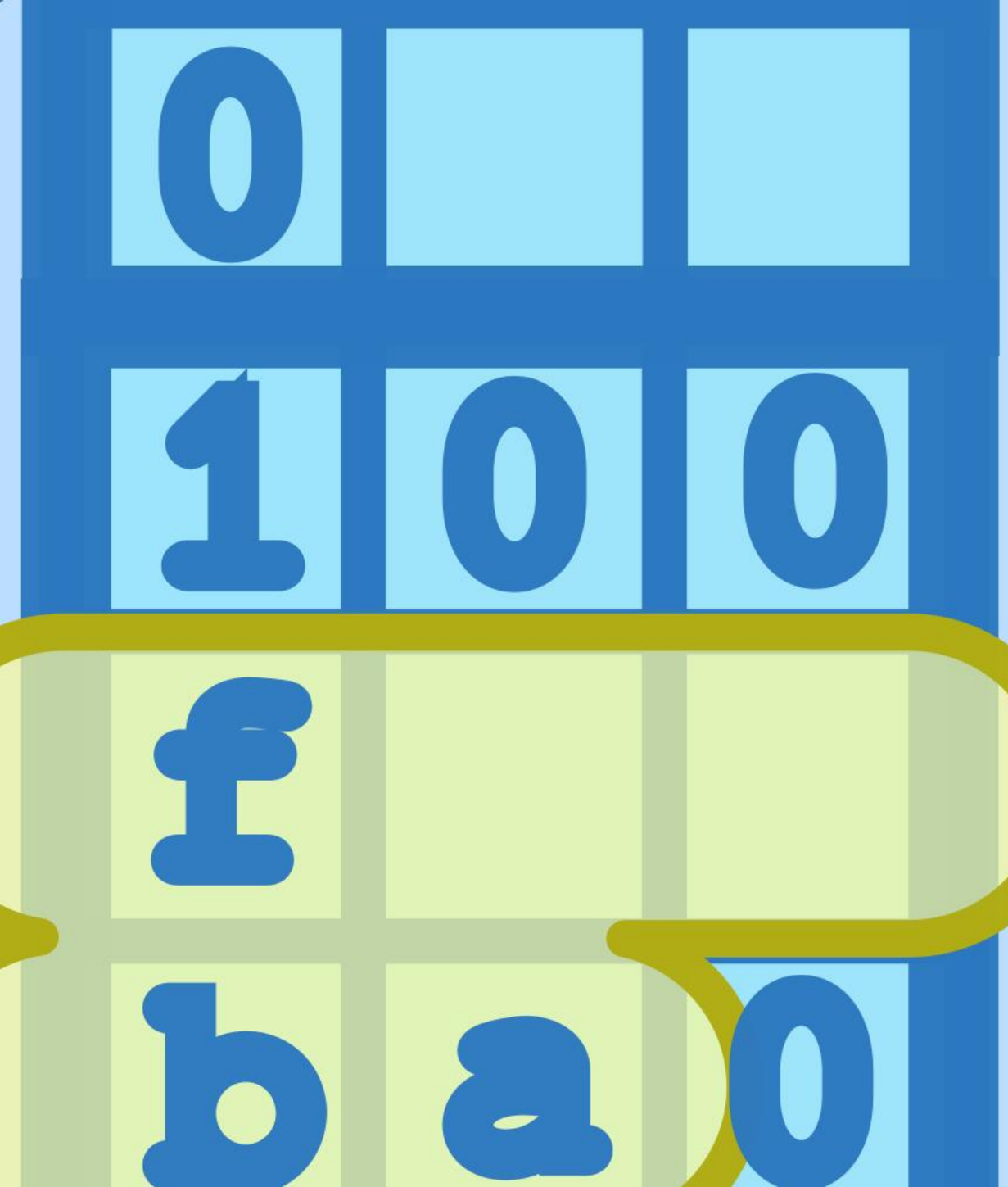
f a b =>

b a

order not fixed...

Parallelisation



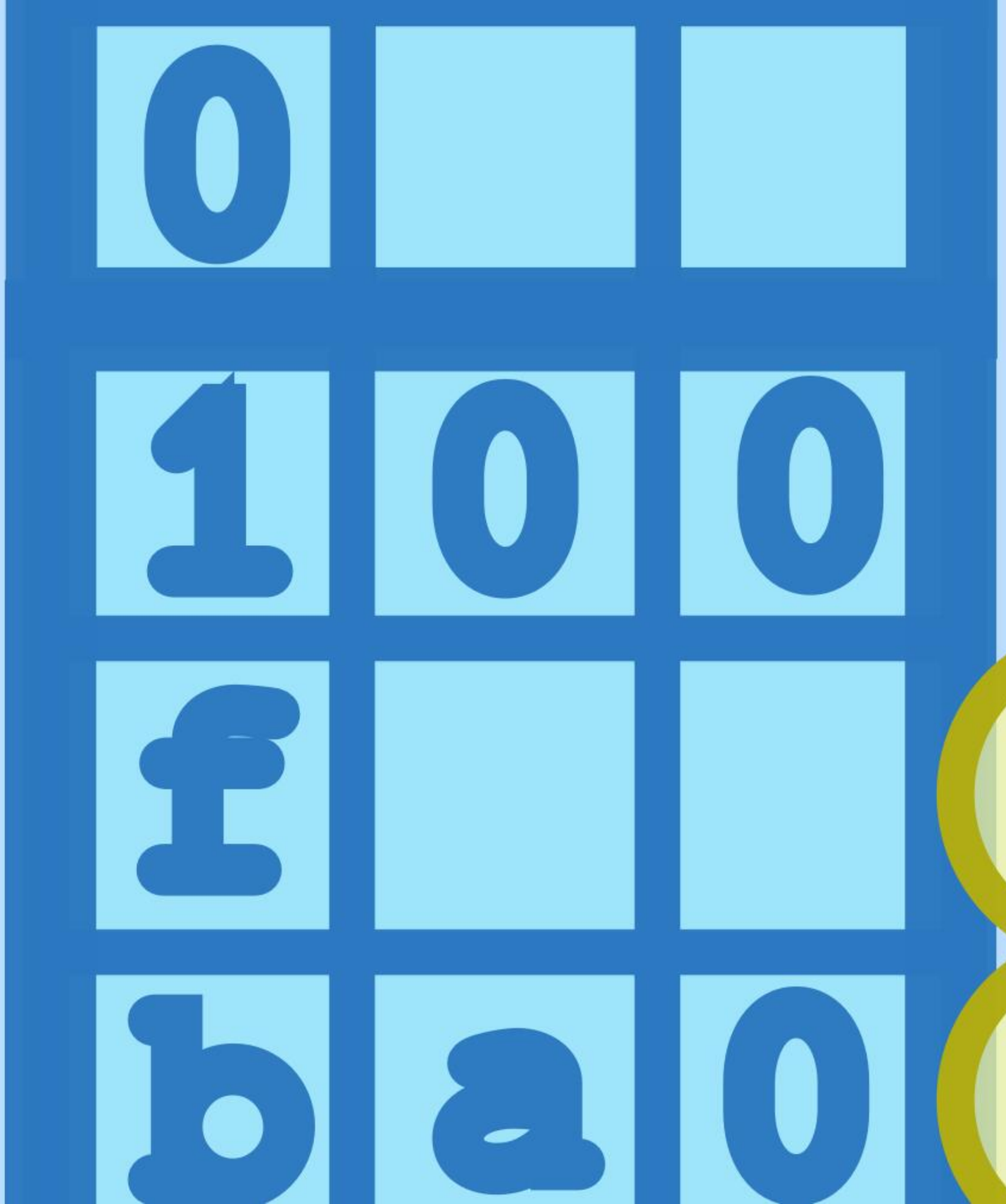


f c d =>

a a

f a b =>

b a

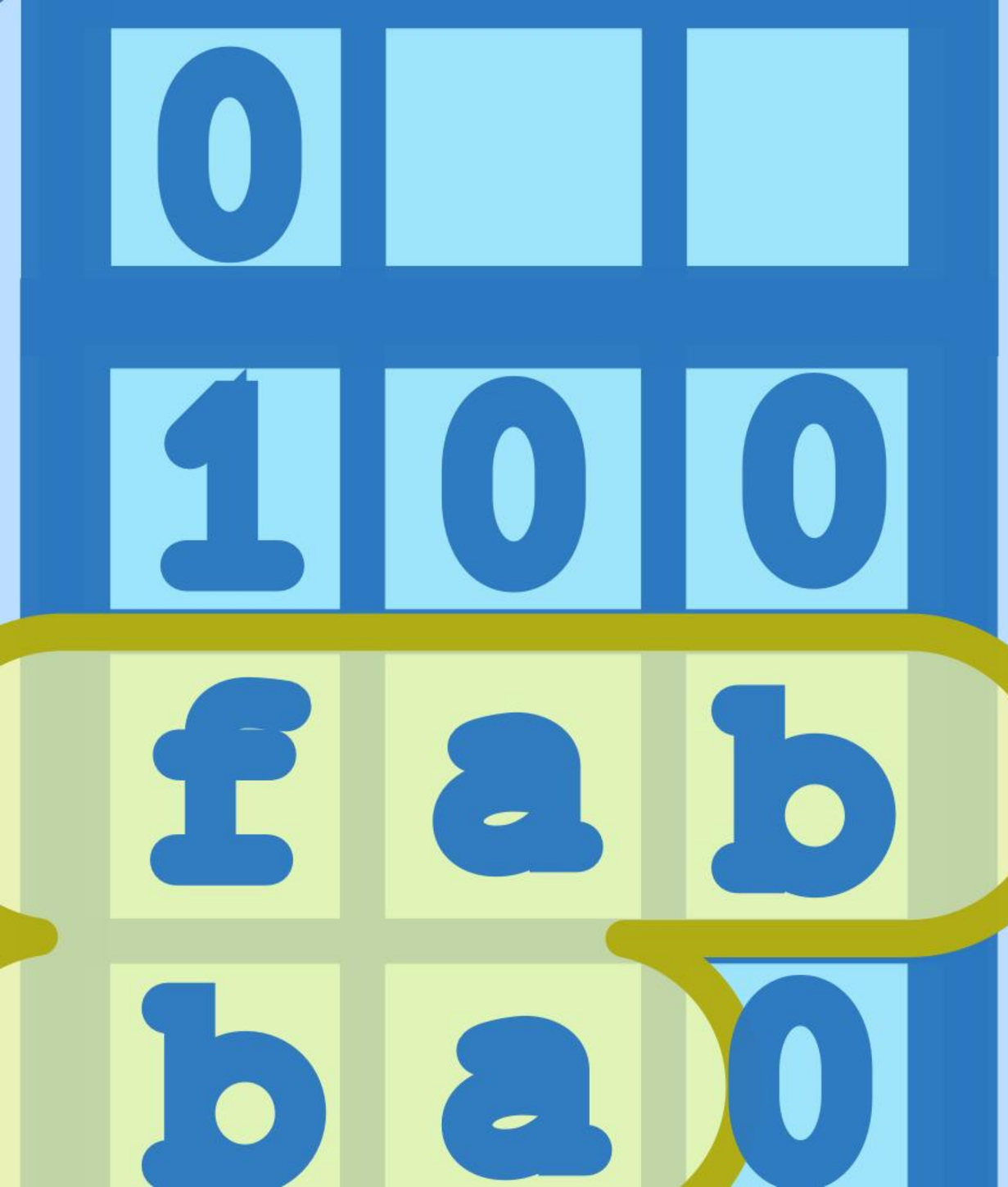


f c d =>

a a

f a b =>

b a



f c d =>

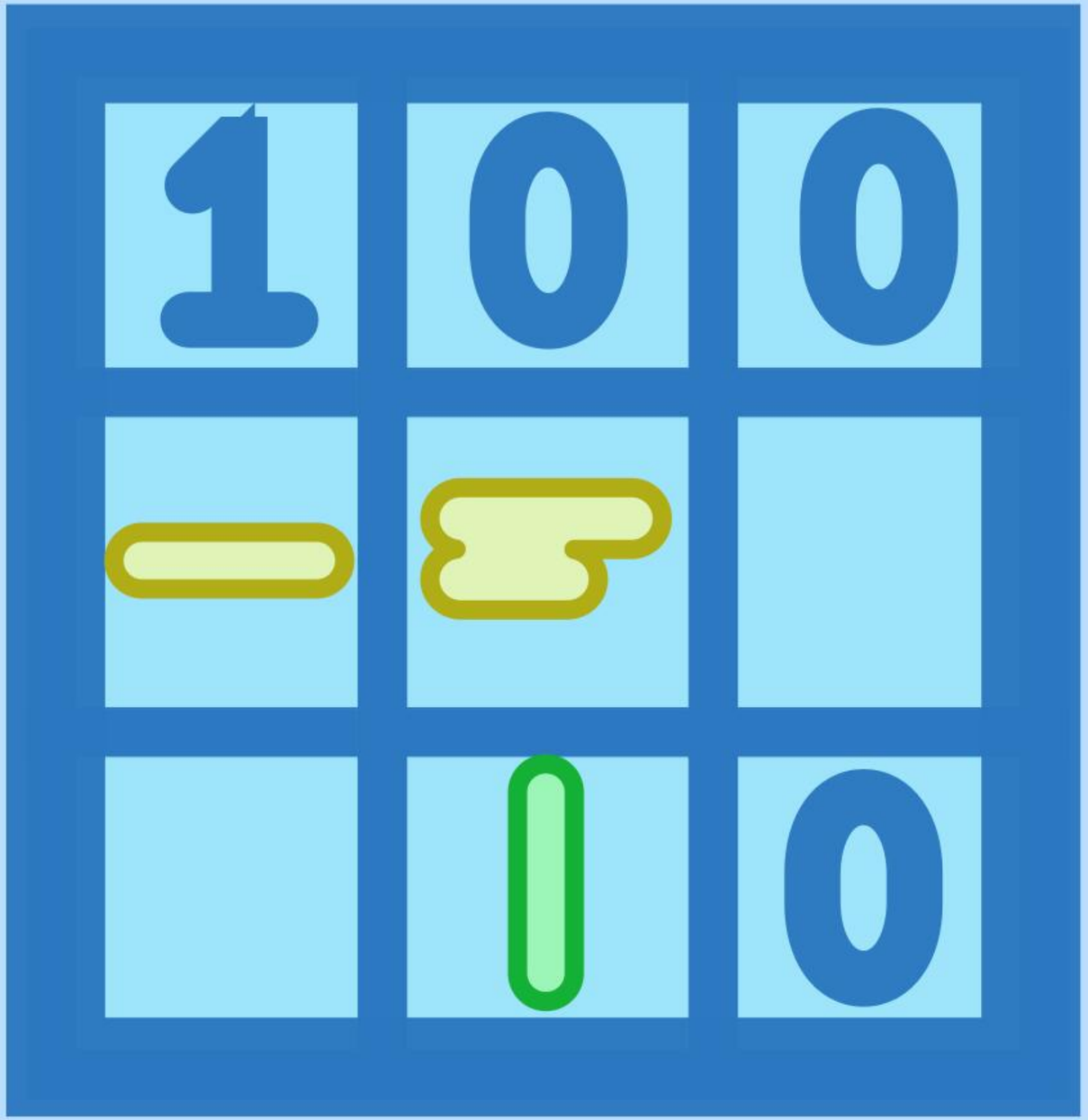
a a

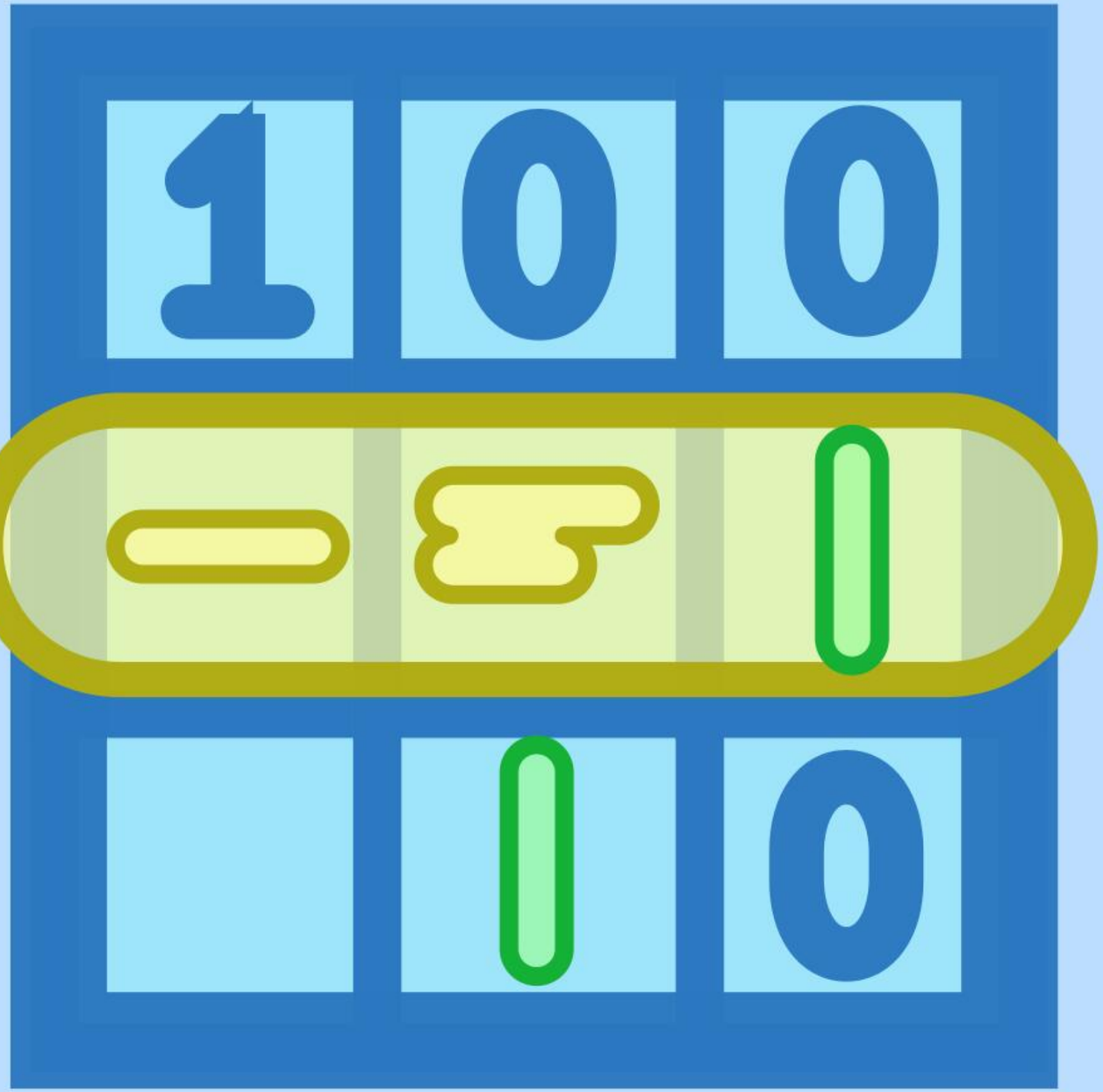
f a b =>

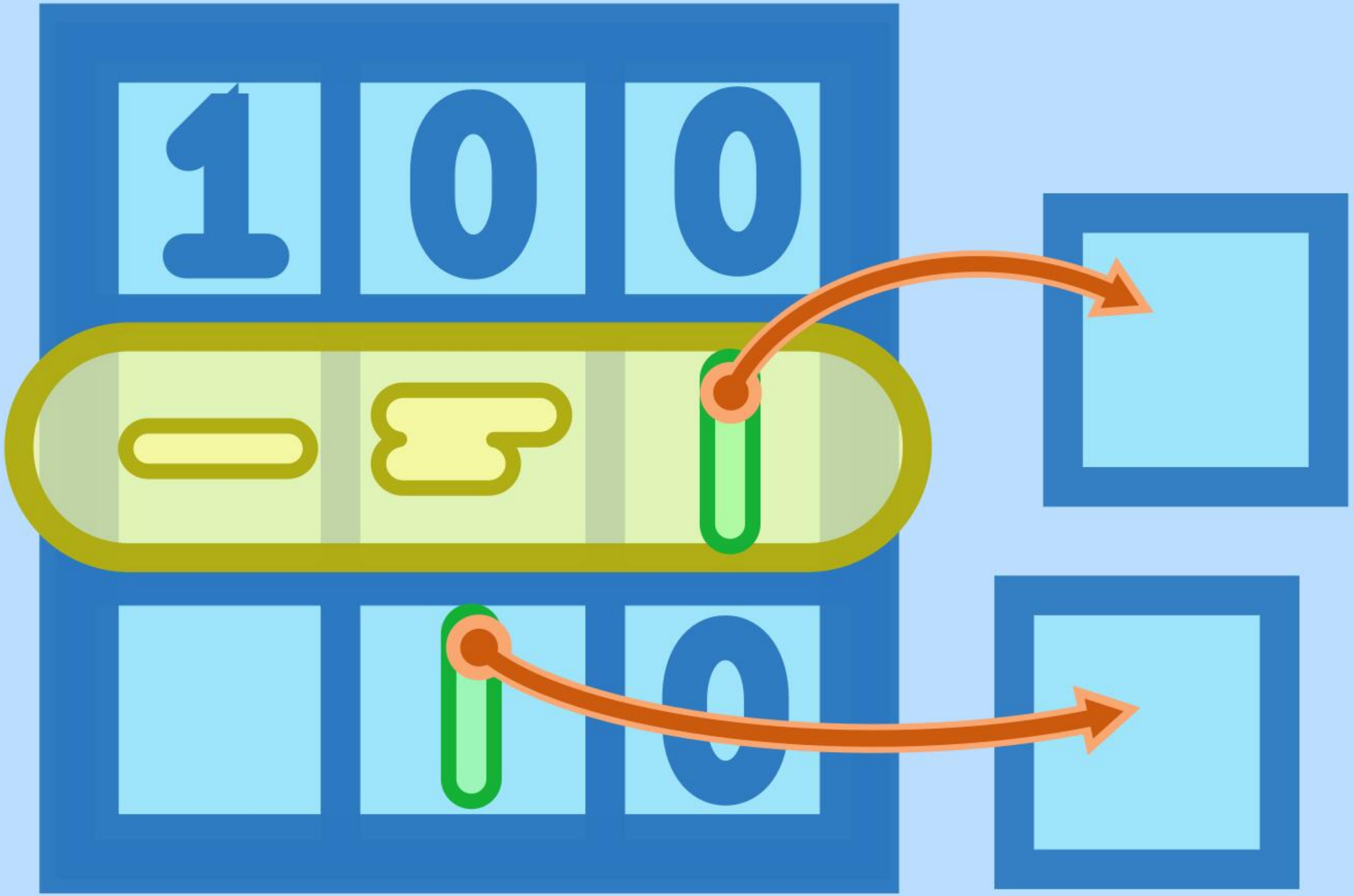
b a

In classical **functional languages** it is extremely **hard** to reason from the **output to the input**!

Generalisation


1
0
0
0


1
0
0
0


1
0
0
0

Now, take **any** programming language...

At this point it should be **noted:**

Unit propagation with **sufficiently many instantiated variables*** is just the **execution of a program**!

*sufficiently many for unit propagation to deduce all variables

```
data RAMIns (Ptr : Set)  : Set where

    halt : RAMIns Ptr
    increment : Ptr -> RAMIns Ptr
    copy : Ptr -> Ptr -> RAMIns Ptr
    jump-if_eq_to_: (eq1 : Ptr) -> (eq2 : Ptr) -> (target : Ptr) -> RAMIns Ptr
```

```
data RAMIns (Ptr : Set)  : Set where

    halt : RAMIns Ptr
    increment : Ptr -> RAMIns Ptr
    copy : Ptr -> Ptr -> RAMIns Ptr
    jump-if_eq_to_: (eq1 : Ptr) -> (eq2 : Ptr) -> (target : Ptr) -> RAMIns Ptr


step : (mem : Memory) -> (prog-counter : Ptr) -> RAMIns Ptr -> Tuple Memory Ptr
step mem ctr halt = (mem , ctr)
step mem ctr (increment ptr) = ( do {p <- get ptr; write ptr (succc ptr)} , succ ctr)
step mem ctr ...
```

```
data RAMIns (Ptr : Set)  : Set where

    halt : RAMIns Ptr
    increment : Ptr -> RAMIns Ptr
    copy : Ptr -> Ptr -> RAMIns Ptr
    jump-if_eq_to_: (eq1 : Ptr) -> (eq2 : Ptr) -> (target : Ptr) -> RAMIns Ptr


step : (mem : Memory) -> (prog-counter : Ptr) -> RAMIns Ptr -> Tuple Memory Ptr
step mem ctr halt = (mem , ctr)
step mem ctr (increment ptr) = ( do {p <- get ptr; write ptr (succc ptr)} , succ ctr)
step mem ctr ...


unit-prop : Memory -> List Ptr -> Tuple Memory (List Ptr)
```

```
data RAMIns (Ptr : Set)  : Set where

    halt : RAMIns Ptr
    increment : Ptr -> RAMIns Ptr
    copy : Ptr -> Ptr -> RAMIns Ptr
    jump-if_eq_to_: (eq1 : Ptr) -> (eq2 : Ptr) -> (target : Ptr) -> RAMIns Ptr
```

Note: This instruction does not work on immutable memory!
(Workaround: have pointers with timestamp or use functional language core)

```
step : (mem : Memory) -> (prog-counter : Ptr) -> RAMIns Ptr -> Tuple Memory Ptr
step mem ctr halt = (mem , ctr)
step mem ctr (increment ptr) = ( do {p <- get ptr; write ptr (succc ptr)} , succ ctr)
step mem ctr ...

unit-prop : Memory -> List Ptr -> Tuple Memory (List Ptr)
```

Sketch!

for alternative instruction set see
Cook and Reckhow (1973), RASP

```
data RAMIns : Set where

    halt : RAMIns
    assign-new : Ptr -> RAMIns -> RAMIns
    copy : Ptr -> Ptr -> RAMIns
    jump-if_eq_to_: (eq1 : Ptr) -> (eq2 : Ptr) -> (target : Ptr) -> RAMIns

    pointer : Ptr -> RAMIns
    tuple : Ptr -> Ptr -> RAMIns

    fst : Ptr -> Ptr -> RAMIns
    snd : Ptr -> Ptr -> RAMIns
    trd : Ptr -> Ptr -> RAMIns
```

This creates a new constraint at a given location with the given constructor

data containters

accessors for constraits subcontents

Already stronger search than **PROLOG**!

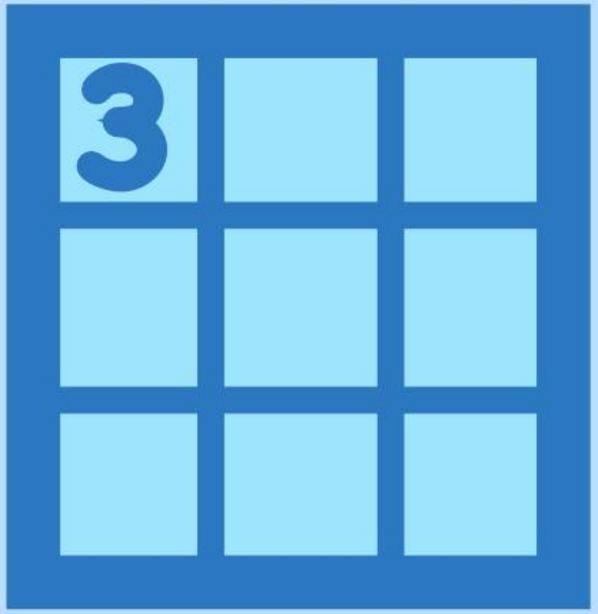
Theorem: Every **general recursive** language can be transformed into an equaliy **general recursive unit propagating search engine**.

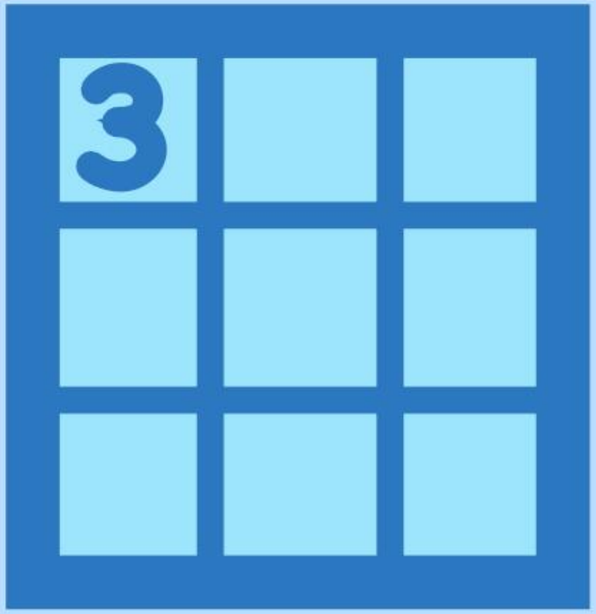
Theorem: Every deterministic general recursive language can be transformed into an equaliy general recursive unit propagating search engine. *

*with maybe just a few syntactic changes
(ability to talk about free variables)

What to do when unit propagation is **insufficient**?

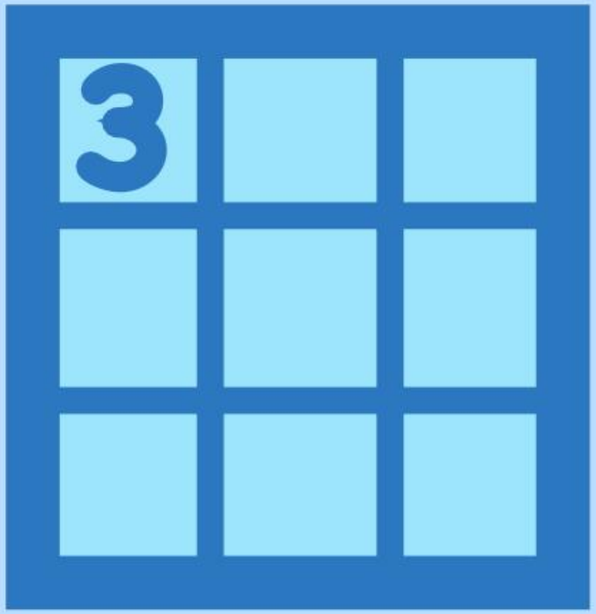
Unit propagation might not deduce all values.





Unit propagation might not deduce all values.

Can be useful: Half evaluated term is representation of **solution set**.

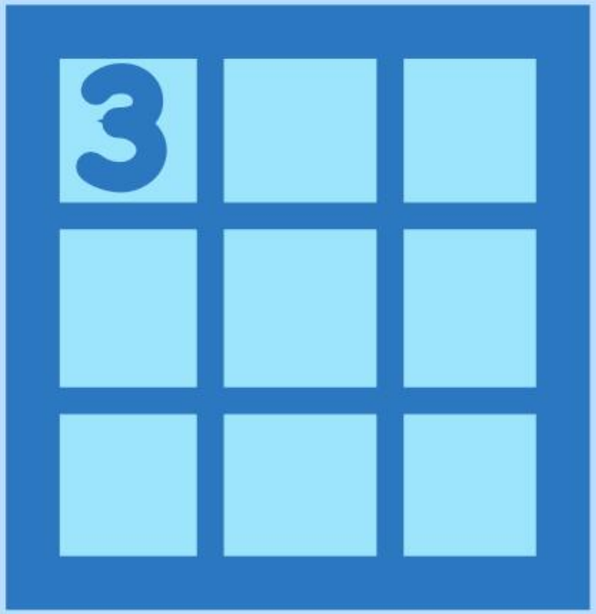


Unit propagation might not deduce all values.

Can be useful: Half evaluated term is representation of **solution set**.

Aim 1: Evaluate term further to be more **efficient to query for solutions**

Needs **formal** handle for **complexity of querying**.
Good thing we have our language simulated.



Unit propagation might not deduce all values.

Can be useful: Half evaluated term is representation of **solution set**.

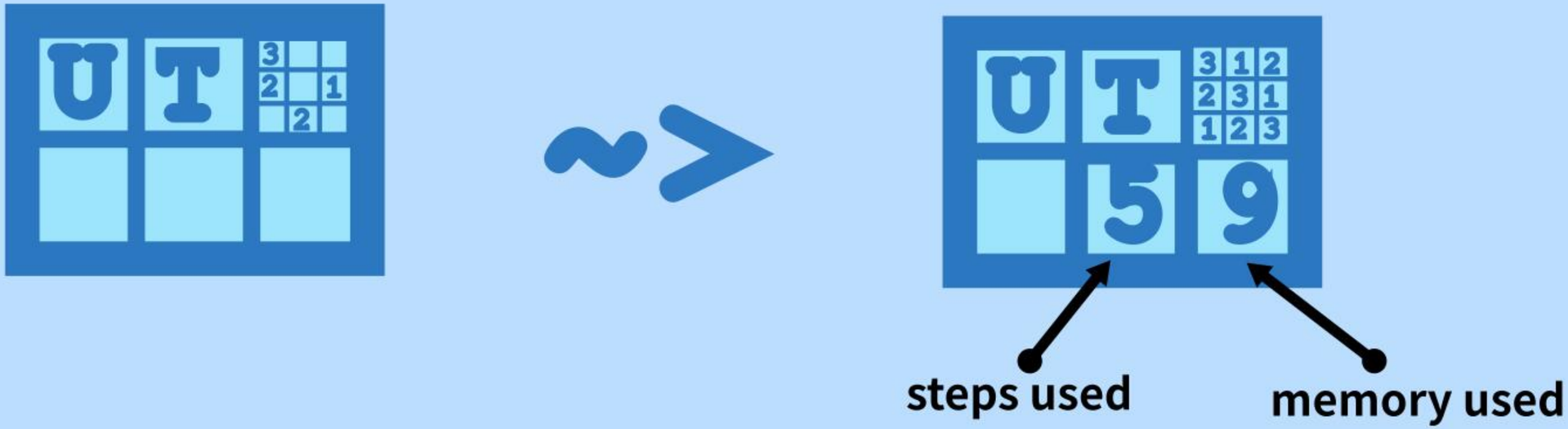
Aim 2: Solve for any assignment using **only unit propagation**

Can unit propagation be complete?
Yes, very much so.

A **partial memory state** *S* is a representation of **partially assigned memory** together with a list of all assigned or unassigned **relevant pointers** (so it can be known where to search for to be evaluated clauses and when all needed variables have been assigned a value)

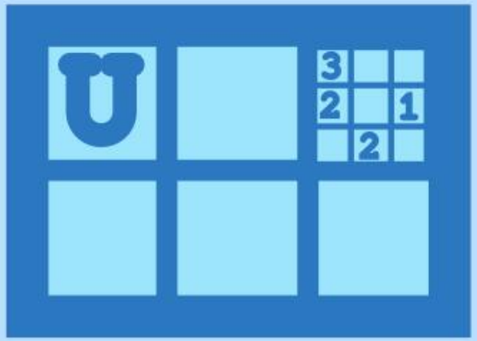
There is a term *T* in memory mem and an input pointer p, such that for all partial memory states R at p, **unit propagation** reduces R until **all unassigned variables have an assignmen**t , or propagates that there cannot be an assignment. Such a term T is called a **solver**.

A solver can be quickly adjusted to also **deduce complexity information**.
Due to the concurrent nature of unit propagation, this might not even slow the solver!



This can be achieved by also **simulating unit propagation** with unit propagation, defined in a similar matter as we did with the solvers.

What happens if we leave the **solver unassigned?**



?

What happens if we leave the **solver unassigned?**

It is **being deduced**. This can be used to **learn or verify*** solvers.
Let's acknowledge for a second how amazing that is.

*might still need interaction

Also, **this is allowed**.

*I know, bracketing is wrong

The deduced solver can **directly feedback** into its **own deduction**.
This means: The more the solver is being developed, the quicker its further development becomes!

Summary

Every **programming language** can be turned into a **search engine**.

Unit propagation as evaluation mechanism has many upsides like (simulated) **concurrency** or **flexible evaluation orders**.

Using this technique, **search engines** could be **automatically created**.

State of Affairs

Aim to create a verified implementation in **AGDA**

So far: **Memory interface** has been formalised

Long term aim: Extract a **C-like implementation** from the **proof of correctness**.

Aim to proof the existence of a **unit propagation and solving term**.

The concept is **ready**.

We just need to **build** it.

Thank you for your **attention!**

Please ask