

## Inhaltsverzeichnis

<b>1. Pipeline .....</b>	<b>2</b>
1.1. Lösungsidee .....	2
1.1.1. Iterativ .....	2
1.1.2. Rekursiv .....	2
1.1.3. Backtracking .....	2
1.2. Code .....	3
1.3. Testfälle .....	6
<b>2. Sudoku .....</b>	<b>7</b>
2.1. Lösungsidee .....	7
2.2. Code .....	9
2.3. Testfälle .....	15

## 1. Pipeline

### 1.1. Lösungsidee

#### 1.1.1. Iterativ

Um dieses Problem iterativ für die Anzahl  $n$  lösen zu können muss ein kleiner Trick mit einem Hilfsfeld angewandt werden. In diesem Hilfsfeld wird immer die aktuelle count – Anzahl abgelegt. Sollte die Anzahl in dem Hilfsfeld gegenüber der Anzahl im count Feld überschritten werden, so wird das Hilfsfeld an jener Stelle auf 0 gesetzt und der Index erhöht. Auf diese Weise können alle Möglichkeiten abgedeckt werden. Mit einer Abfrage wird nach jedem Schritt eine Funktion `is_valid` aufgerufen, die TRUE liefert, wenn die Länge  $x$  mit dem aktuellen Stand des Hilfsfelds und den Längen erreicht werden kann. Sollte die Funktion FALSE liefern, dann wird dieser Vorgang solange wiederholt, bis `is_valid` TRUE liefert oder  $i \geq n$  ist.

#### 1.1.2. Rekursiv

In der Rekursiven Funktion wird die Variable  $X$  verwendet um die Zwischensumme zu behandeln. Wenn  $X$  den Wert 0 erreicht, so kann die gefragte Länge mit den Pipes erreicht werden. Im Rekursionsabstieg wird  $n$  immer um 1 vermindert, solange  $n > 0$ . In der rekursiven Funktion wird das count Array mit einer Schleife behandelt. In dieser Schleife findet dann der rekursive Abstieg mit dem aktuellen „count Status“ ( $i$  der Schleife) statt.

#### 1.1.3. Backtracking

Grundsätzlich ident zum rekursiven Algorithmus jedoch wird im Fall, dass  $x$  bereits kleiner als 0 ist, kein rekursiver Schritt mehr durchgeführt, da man zu diesem Zeitpunkt bereits weiß, dass man zu keiner Lösung in diesem „Zweig“ kommen kann.

## 1.2. Code

```

#include <stdio.h>
#include <stdlib.h>
#include "./pipelib.h"

int main(int argc, char *argv[]){
    int lengths[] = {4 , 2 , 8 };
    int counts[] = {5 , 1 , 1 };
    int n = sizeof(lengths)/sizeof(int);

    int sol[MAX] = { 0 };
    int sum = 0;

    if(argc == 2) {

        printf("possibleIter: %d\n", possibleIter(atoi(argv[1]), lengths, counts,
n));
        printf("possibleRec: %d\n", possibleRec(atoi(argv[1]), lengths, counts, n)
);
        printf("possibleBack: %d\n", possibleBack(atoi(argv[1]), lengths, counts,
n));
    } else {
        printf("wrong number of arguments\n");
    }
    return EXIT_SUCCESS;
}

#ifdef pipelib_h
#define pipelib_h
#include <stdbool.h>

#define MAX 30

bool possibleIter (int const x, int const lengths [], int const counts [], int
const n);
bool possibleRec (int const x, int const lengths [], int const counts [], int
const n);
bool possibleBack (int const x, int const lengths [], int const counts [], int
const n);
bool is_valid(int x, int const lengths[], int sol[], int n);
bool inc(int sol[], int const counts[], int n);

#endif

```

```

#include "./pipelib.h"

/** recursive */
bool possibleRec (int const x, int const lengths [], int const counts [], int
const n){
    if(x == 0) return true;
    else if (n > 0) {
        bool is_possible = false;
        for(int i = 0; i <= counts[n-1]; i++) {
            is_possible = is_possible || possibleRec(x - (lengths[n-
1]*i), lengths, counts, n-1);
        }
        return is_possible;
    } else return false;
}

/** backtracking */
bool possibleBack (int const x, int const lengths [], int const counts [], int
const n){
    if(x == 0) return true;
    else if (n > 0) {
        bool is_possible = false;
        for(int i = 0; i <= counts[n-1]; i++) {
            if(x >= 0){
                is_possible = is_possible || possibleRec(x - (lengths[n-
1]*i), lengths, counts, n-1);
            }
        }
        return is_possible;
    } else return false;
}

/** iterativ */

bool is_valid(int x, int const lengths[], int sol[], int n) {
    int sum = 0;
    for(int i = 0; i < n; i++) {
        sum += lengths[i] * sol[i];
    }
    return sum == x;
}

```

```

bool inc(int sol[], int const counts[], int n) {
    bool run;
    int i = 0;

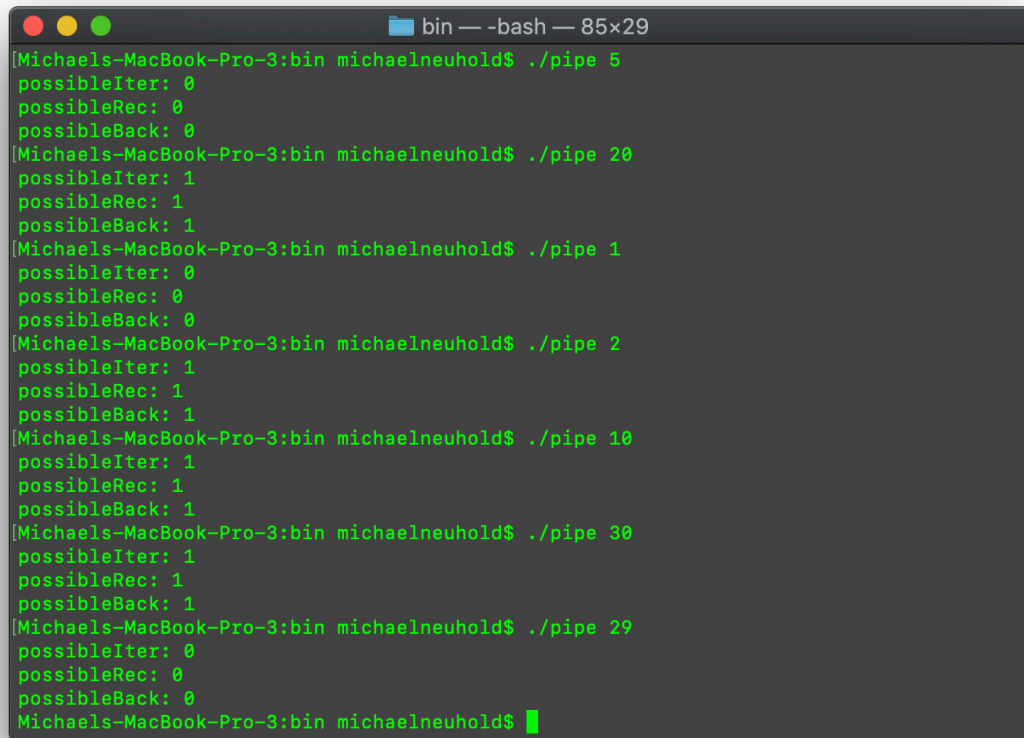
    do {
        sol[i] += 1;
        run = false;
        if(sol[i] > counts[i]) {
            sol[i] = 0;
            i++;
            if(i < n){
                run = true;
            } else {
                return false;
            }
        }
    } while(run);
    return true;
}

// returns true if it is possible and false if not
bool possibleIter(int const x, int const lengths [], int const counts [], int
const n) {
    // sol = solutions --> array to save "virtual indices of FOR loops"
    int sol[MAX] = { 0 };
    do {
        if(is_valid(x, lengths,sol,n)){
            return true;
        }
    } while(inc(sol, counts, n));
    return false;
}

```

### 1.3. Testfälle

Der folgende Screenshot zeigt 7 Testfälle. Wie in C üblich steht 0 für FALSE und 1 für TRUE.



```
bin — -bash — 85x29
[Michaels-MacBook-Pro-3:bin michaelneuhold$ ./pipe 5 ]
possibleIter: 0
possibleRec: 0
possibleBack: 0
[Michaels-MacBook-Pro-3:bin michaelneuhold$ ./pipe 20 ]
possibleIter: 1
possibleRec: 1
possibleBack: 1
[Michaels-MacBook-Pro-3:bin michaelneuhold$ ./pipe 1 ]
possibleIter: 0
possibleRec: 0
possibleBack: 0
[Michaels-MacBook-Pro-3:bin michaelneuhold$ ./pipe 2 ]
possibleIter: 1
possibleRec: 1
possibleBack: 1
[Michaels-MacBook-Pro-3:bin michaelneuhold$ ./pipe 10 ]
possibleIter: 1
possibleRec: 1
possibleBack: 1
[Michaels-MacBook-Pro-3:bin michaelneuhold$ ./pipe 30 ]
possibleIter: 1
possibleRec: 1
possibleBack: 1
[Michaels-MacBook-Pro-3:bin michaelneuhold$ ./pipe 29 ]
possibleIter: 0
possibleRec: 0
possibleBack: 0
[Michaels-MacBook-Pro-3:bin michaelneuhold$ ]
```

## 2. Sudoku

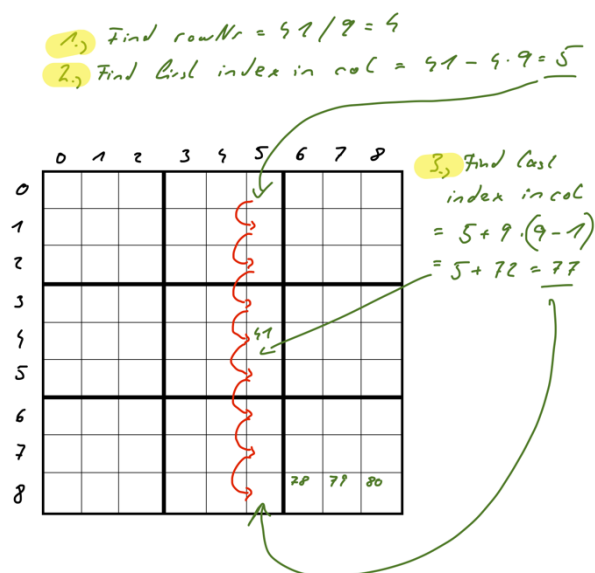
### 2.1. Lösungsidee

Das Programm wurde nach dem Backtracking-Schema implementiert. Um es auszuführen sind keine Parametereingaben beim Programmstart notwendig. Es wurden einige Testfälle direkt im Code implementiert.

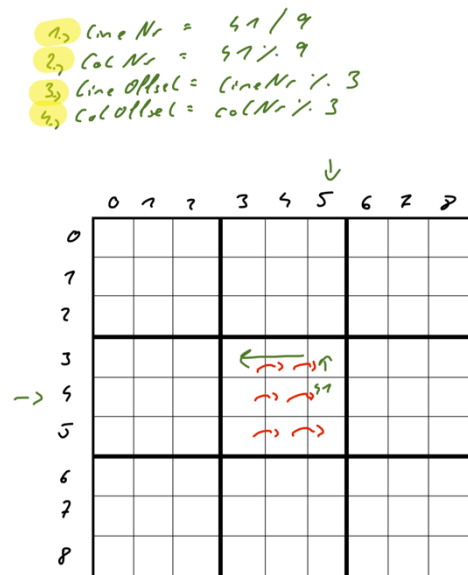
In der Funktion `sudoku_solve` wird das gewählte Sudoku zunächst ausgegeben und im Anschluss wird eine rekursive Funktion aufgerufen, die TRUE liefert wenn das Sudoku gelöst werden konnte und FALSE liefert wenn es nicht gelöst werden konnte. In dieser Rekursiven Funktion wird zunächst nach dem nächsten nicht ausgefüllten Feld im Sudoku gesucht. Im Anschluss können nun alle möglichen Zahlen (1 - 9) in einer Schleife durchwandert werden. Es wird geprüft ob die derzeitige Zahl an der betrachteten Stelle eine valide Eingabe ist. Grundsätzlich setzt sich diese Prüfung aus drei kleineren Funktionen zusammen. Es wird jeweils ein Zeilen-, ein Spalten- und ein Sub-Sudoku- Check durchgeführt.

Die nachstehenden Bilder beschreiben die Vorgehensweise bei den einzelnen Check Funktionen.

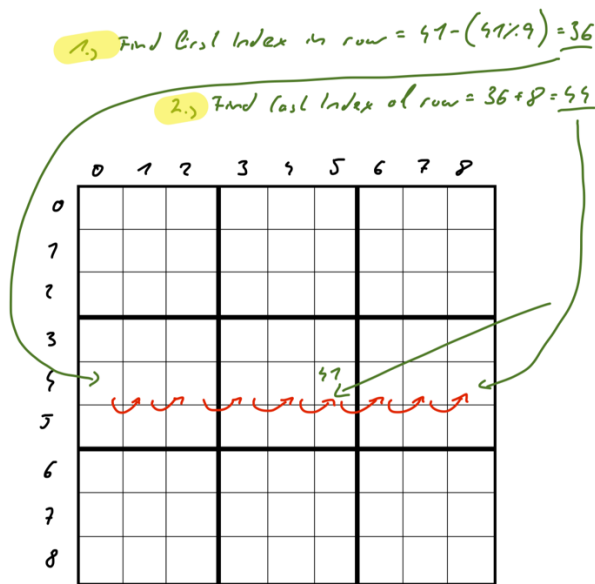
Spalten-Check:



Sub-Square-Check:



## Zeilen-Check



Wenn alle Check-Funktionen True zurückliefern, so ist die Zahl an der betrachteten Stelle eine gültige Eingabe und kann hineingeschrieben werden. Im Anschluss wird mit einem Rekursionsschritt geprüft, ob mit dieser Eingabe das Sudoku vollständig gelöst werden kann. Ist dies nicht der Fall, so wird der Schritt rückgängig gemacht (sprich wieder auf 0 gesetzt) und mit der nächsten Zahl probiert.

Der Algorithmus ist fertig, wenn die Funktion `find_empty_element` -1 zurückliefert. Dies bedeutet, dass keine freie Felder im Sudoku mehr existieren und somit das Sudoku gelöst ist.



## 2.2. Code

```

#include "./sudokulib.h"

int main() {

    // testcases
    int squares1[MAX] = {
        5,0,9 , 0,0,0 , 0,0,7 ,
        0,8,0 , 0,1,0 , 5,2,0 ,
        0,0,3 , 0,8,4 , 0,0,1 ,

        0,9,0 , 7,0,0 , 0,0,2 ,
        4,0,0 , 0,5,0 , 3,9,0 ,
        8,0,2 , 1,0,0 , 0,0,4 ,

        0,0,0 , 3,0,2 , 0,0,5 ,
        0,4,0 , 0,0,0 , 7,0,0 ,
        1,0,7 , 0,9,0 , 0,8,0
    };

    int squares2[MAX] = {
        0,1,0 , 0,0,8 , 4,9,6 ,
        0,4,0 , 0,6,1 , 0,0,0 ,
        0,9,0 , 0,0,0 , 5,7,0 ,

        6,0,7 , 0,0,0 , 0,1,0 ,
        0,0,0 , 4,1,7 , 0,0,3 ,
        0,0,0 , 6,0,0 , 0,0,2 ,

        1,0,9 , 8,0,0 , 0,4,0 ,
        0,0,4 , 0,9,3 , 6,0,0 ,
        0,0,0 , 7,4,5 , 0,2,0
    };

    int squares3[MAX] = {
        3,0,0 , 0,0,0 , 0,0,1 ,
        0,7,6 , 0,0,0 , 5,3,0 ,
        0,1,9 , 0,4,0 , 7,2,0 ,

        0,0,0 , 8,0,6 , 0,0,0 ,
        0,0,1 , 0,7,0 , 9,0,0 ,
        0,0,0 , 5,0,4 , 0,0,0 ,

        0,3,5 , 0,6,0 , 2,8,0 ,
        0,8,7 , 0,0,0 , 6,5,0 ,
        6,0,0 , 0,0,0 , 0,0,7
    };
}

```

```

int squares4[MAX] = {
    0,0,1,  2,0,7,  0,0,0,
    0,6,2,  0,0,0,  0,0,0,
    0,0,0,  0,0,0,  9,4,0,

    0,0,0,  9,8,0,  0,0,3,
    5,0,0,  0,0,0,  0,0,0,
    7,0,0,  0,3,0,  0,2,1,

    0,0,0,  1,0,2,  0,0,0,
    0,7,0,  8,0,0,  4,1,0,
    3,0,4,  0,0,0,  0,8,0
};

// tests
printf("1. Sudoku\n");
sudoku_solve(squares1);
print_big_line();
printf("2. Sudoku\n");
sudoku_solve(squares2);
print_big_line();
printf("3. Sudoku\n");
sudoku_solve(squares3);
print_big_line();
printf("4. Sudoku\n");
sudoku_solve(squares4);
print_big_line();

return EXIT_SUCCESS;
}

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>

#define MAX 81
#define SUDOKU_SIZE 9
#define BIG_LINE_SIZE 27
#define SUB_SUDOKU_SIZE 3
#define TO_MILLISEC 1000
#define EMPTY 0

void print_line();
void print_big_line();
void print_err(char msg[]);
void print_sudoku(int squares[]);
bool check_line(int squares[], int i, int number);
bool check_col(int squares[], int i, int number);

```

```

bool check_square(int squares[], int i, int number);
bool check_elements(int squares[], int i, int number);
int find_empty_element(int squares[]);
bool sudoku_solve_rec(int squares[]);
void sudoku_solve(int squares[]);

#include "../sudokulib.h"

// prints separator between BIG_LINE_SIZE
void print_line() {
    for(int i = 0; i < 21; i++)
        printf("-");
    printf("\n");
}

// prints separator between testcases
void print_big_line() {
    for(int i = 0; i < 50; i++)
        printf("=");
    printf("\n");
}

// prints error
void print_err(char msg[]) {
    printf("<< ERROR (%s) >>\n", msg);
}

// prints sudoku to console
void print_sudoku(int squares[]){
    print_line();
    for(int i = 0; i < MAX; i++){
        if(squares[i] == EMPTY){
            printf("- ");
        } else {
            printf("%d ", squares[i]);
        }
        if((i+1) % SUB_SUDOKU_SIZE == 0 && (i+1) % SUDOKU_SIZE != 0)
            printf("| ");
        if((i+1) % SUDOKU_SIZE == 0)
            printf("\n");
        if((i+1) % BIG_LINE_SIZE == 0)
            print_line();
    }
    return;
}

```

```

// returns true if number is valid in line/row
// else false
bool check_line(int squares[], int i, int number) {
    int start = i - (i % SUDOKU_SIZE);
    int end = start + SUDOKU_SIZE - 1;
    for(int j = start; j <= end; j++) {
        if(squares[j] == number) {
            return false;
        }
    }
    return true;
}

// returns true if number is valid in col
// else false
bool check_col(int squares[], int i, int number) {
    int start = i - SUDOKU_SIZE * (i / SUDOKU_SIZE);
    int end = start + SUDOKU_SIZE * SUDOKU_SIZE - 1;
    for(int j = start; j <= end; j += SUDOKU_SIZE) {
        if(squares[j] == number) {
            return false;
        }
    }
    return true;
}

// returns true if number is valid in sub_square
// else false
bool check_square(int squares[], int i, int number) {
    int line_nr = i / SUDOKU_SIZE;
    int col_nr = i % SUDOKU_SIZE;
    int line_offset = line_nr % SUB_SUDOKU_SIZE;
    int col_offset = col_nr % SUB_SUDOKU_SIZE;

    int sub_square_top = i - SUDOKU_SIZE * line_offset;
    int sub_square_start = sub_square_top - col_offset;
    int sub_square_end = sub_square_start + 2 * SUDOKU_SIZE + 2;

    int j = sub_square_start;

    while(j <= sub_square_end){
        if(squares[j] == number){
            return false;
        } else if((j + 1) % SUB_SUDOKU_SIZE == 0) {
            j += SUDOKU_SIZE - 2;
        } else {
            j++;
        }
    }
    return true;
}

```

```

// returns if number is valid at index i
bool check_elements(int squares[], int i, int number) {
    return check_square(squares,i,number) &&
           check_line(squares,i,number) &&
           check_col(squares,i,number);
}

// returns index of next empty square
// if there is no empty square return -1
int find_empty_element(int squares[]) {
    for(int i = 0; i <= MAX-1; i++) {
        if(squares[i] == EMPTY) {
            return i;
        }
    }
    return -1;
}

// recursive function with backtracking
bool sudoku_solve_rec(int squares[]) {

    int i = find_empty_element(squares);

    if(i == -1){
        return true;
    }

    for(int number = 1; number <= SUDOKU_SIZE; number++) {
        if(check_elements(squares,i,number)) {
            squares[i] = number;
            if(sudoku_solve_rec(squares)){
                return true;
            } else {
                squares[i] = EMPTY;
            }
        }
    }
    return false;
}

```

```
// solve sudoku calls recursive function
void sudoku_solve(int squares[]) {
    clock_t start, end = 0;
    print_sudoku(squares);
    start = clock();
    if(sudoku_solve_rec(squares)) {
        end = clock();
        print_sudoku(squares);
    } else {
        print_err("could not be solved");
    }
    printf("time: %f ms\n", ((double)((end - start)) / CLOCKS_PER_SEC) * TO_MILL
ISEC);
}
```

### 2.3. Testfälle

Es wurde 4 Testfälle im Code implementiert. Dabei wurde darauf geachtet, dass sie verschiedene Schwierigkeitsstufen haben. Unter der Lösung jedes Sudoku wird jeweils die Berechnungszeit ausgegeben. Wenn man die Zeiten analysiert, so ist das 3. Sudoku am schwersten zu lösen gewesen.

```

1. Sudoku
5 - 9 | - - - | - - 7
- 8 - | - 1 - | 5 2 -
- - 3 | - 8 4 | - - 1
-----
- 9 - | 7 - - | - - 2
4 - - | - 5 - | 3 9 -
8 - 2 | 1 - - | - - 4
-----
- - - | 3 - 2 | - - 5
- 4 - | - - - | 7 - -
1 - 7 | - 9 - | - 8 -
-----
5 1 9 | 6 2 3 | 8 4 7
6 8 4 | 9 1 7 | 5 2 3
7 2 3 | 5 8 4 | 9 6 1
-----
3 9 6 | 7 4 8 | 1 5 2
4 7 1 | 2 5 6 | 3 9 8
8 5 2 | 1 3 9 | 6 7 4
-----
9 6 8 | 3 7 2 | 4 1 5
2 4 5 | 8 6 1 | 7 3 9
1 3 7 | 4 9 5 | 2 8 6
-----
time: 0.249000 ms
printf("1. Sudoku\n");

```

```

2. Sudoku
- 1 - | - - 8 | 4 9 6
- 4 - | - 6 1 | - - -
- 9 - | - - - | 5 7 -
-----
6 - 7 | - - - | - 1 -
- - - | 4 1 7 | - - 3
- - - | 6 - - | - - 2
-----
1 - 9 | 8 - - | - 4 -
- - 4 | - 9 3 | 6 - -
- - - | 7 4 5 | - 2 -
-----
2 1 3 | 5 7 8 | 4 9 6
7 4 5 | 9 6 1 | 2 3 8
8 9 6 | 2 3 4 | 5 7 1
-----
6 8 7 | 3 5 2 | 9 1 4
9 5 2 | 4 1 7 | 8 6 3
4 3 1 | 6 8 9 | 7 5 2
-----
1 7 9 | 8 2 6 | 3 4 5
5 2 4 | 1 9 3 | 6 8 7
3 6 8 | 7 4 5 | 1 2 9
-----
time: 0.094000 ms
printf("1. Sudoku\n");
printf("2. Sudoku\n");

```

```

3. Sudoku
-----
3 - - | - - - | - 1
- 7 6 | - - - | 5 3 -
- 1 9 | - 4 - | 7 2 -
-----
- - - | 8 - 6 | - - -
- - 1 | - 7 - | 9 1 -
- - - | 5 - 4 | - - -
-----
- 3 5 | - 6 - | 2 8 -
- 8 7 | - - - | 6 5 -
6 - - | - - - | - - 7
-----
3 2 8 | 6 5 7 | 4 9 1
4 7 6 | 1 2 9 | 5 3 8
5 1 9 | 3 4 8 | 7 2 6
-----
2 5 4 | 8 9 6 | 1 7 3
8 6 1 | 2 7 3 | 9 4 5
7 9 3 | 5 1 4 | 8 6 2
-----
9 3 5 | 7 6 1 | 2 8 4
1 8 7 | 4 3 2 | 6 5 9
6 4 2 | 9 8 5 | 3 1 7
-----
time: 1.277000 ms
=====

```

```

4. Sudoku
-----
- - 1 | 2 - 7 | - - -
- 6 2 | - - - | - - -
- - - | - - - | 9 4 -
-----
- - - | 9 8 - | - - 3
5 - - | - - - | - - -
7 - - | - 3 - | - 2 1
-----
- - - | 1 - 2 | - - -
- 7 - | 8 - - | 4 1 -
3 - 4 | - - - | - 8 -
-----
4 3 1 | 2 9 7 | 5 6 8
9 6 2 | 4 5 8 | 1 3 7
8 5 7 | 3 1 6 | 9 4 2
-----
1 2 6 | 9 8 4 | 7 5 3
5 8 3 | 7 2 1 | 6 9 4
7 4 9 | 6 3 5 | 8 2 1
-----
6 9 8 | 1 4 2 | 3 7 5
2 7 5 | 8 6 3 | 4 1 9
3 1 4 | 5 7 9 | 2 8 6
-----
time: 0.221000 ms
=====

```