Ausarbeitung Übung 03

Table of Contents

Pipeline

Lösungsidee

1. **Iterativ**: In der äußersten Schleife werden alle möglichen Vielfachen der ersten Rohrlänge l ausprobiert, „solange der Vorrat (v) reicht“. Für jede Teillänge
$$x \in \{ i \mid i \, MOD \, l = 0 \wedge 0 \leq \frac{i}{l} \leq v \}$$
werden alle Kombinationen des nächsten Rohres mithilfe einer zweiten – geschachtelten – Schleife probiert. Wenn eine Kombination die gewünschte Länge genau erreicht, werden alle Schleifen abgebrochen. Diese Herangehensweise ist nicht nur ungeeignet, sondern auch nicht vertretbar für große n, da man für n Rohre auch n Schleifen braucht.
2. **Rekursiv**: Den Anfang macht der Aufruf zur Funktion mit einer totalen Länge von 0. Dann wird die Länge des ersten verfügbaren Rohrs hinzugefügt. Nachdem ein Rohr „aus dem Sortiment entnommen wurde“, wird die Funktion rekursiv mit der neuen Länge aufgerufen. Es wird wieder das erste verfügbare Rohr gesucht und dessen Länge zur Zwischensumme addiert. Wenn die Funktion mit einer Summe gleich der gesuchten Gesamtlänge aufgerufen wird, ist der Test positiv, andernfalls negativ. Wird beim Rücksprung in die übergeordnete („rufende“) Funktion festgestellt, dass die soeben getestete Summe keine gesuchte Lösung war, wird das verwendete Rohr wieder zurück auf den Haufen geworfen, damit sie in Kombination mit anderen Rohren erneut getestet werden kann. Dieses Verfahren lehnt sich schon sehr stark an Backtracking an, weswegen die rekursive Variante der Backtracking-Version auch sehr stark ähnelt. Ich fand keinen Weg eine rekursive Funktion ohne Backtracking-Mechanismus zu entwickeln.
3. **Backtracking**: Funktioniert genau wie die Rekursive Funktion, bloß habe ich versucht die Backtracking Schablone (siehe Abbildung 1) zu verwenden, um die Verwendung von BT offensichtlich zu machen.

```
1  function LOESUNG (↓i)        — x_{0,i-1} bildet eine Teillösung
2      for alle möglichen Werte von x_i do
3          if x_i passt zur bisherigen Teillösung then
4              erzeuge mit x_i eine neue Teillösung
5
6              if i = n − 1 then
7                  WRITE (↓x)        — Lösung gefunden
8                  halt
9              end if
10
11             LOESUNG (↓(i+1))
12
13             nimm Teillösung mit x_i zurück      — Teillösung mit x_{0,i} nicht möglich
14         end if
15     end for      — Teillösung mit x_{0,i-1} nicht möglich
16 end function
```

Abbildung 1 Backtracking Schablone von Peter Kulczycki

Implementierung

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

typedef enum
{
    false, true
} bool;
typedef int *const CountsArray;
typedef const int *const LengthsArray;
typedef bool (*SolutionFunction)(int, LengthsArray, CountsArray, int, int);

/**
```

```
 * @see possible
 */
bool possible_it(int x, const int *const lengths, const int *const counts, int n)
{
    // I am eagerly awaiting the first time I see code like this at work...

    if (n > 3) {
        printf("The iterative version will only consider the first 3 pipe lengths.\n");
    }
    bool possible = false;
    int sum = 0;
    int first = 0;
    while (first < counts[0] + 1 && !possible) {
        sum = lengths[0] * first;
        possible = sum == x;

        if (n > 1) {
            int second = 0;
            while (second < counts[1] + 1 && !possible) {
                sum = lengths[0] * first + lengths[1] * second;
                possible = sum == x;

                if (n > 2) {
                    int third = 0;
                    while (third < counts[2] + 1 && !possible) {
                        sum = lengths[0] * first + lengths[1] * second + lengths[2] * third; // look
ma! I'm on top of the pyramid!!
                        possible = sum == x;

                        ++third;
                    }
                }

                ++second;
            }
        }

        ++first;
    }
    return possible;
}

/**
 * @see possible
 * @param current_length The sum of all pipe lengths tested in this branch
 *                       of the recursion tree.
 */
bool possible_rec(int x, LengthsArray lengths, CountsArray counts, int n, int current_length)
{
    // For a note on the recursive version, have
    // a look at the documentation.
    if (current_length == x) {
        return true;
    } else if (current_length > x) {
        return false;
    }

    int i = 0;
    bool possible = false;
    while (i < n && !possible) {
        // if there are more pipes with
        // the "current" length (lengths[i])
        if (counts[i] > 0) {
            // use it
            --counts[i];
            possible = possible_rec(x, lengths, counts, n, current_length + lengths[i]);
            // if that pipe didn't work,
            if (!possible) {
                // "un-use" (// backtrack) it
                ++counts[i];
            }
        }
        ++i;
    }
```

```
    return possible;
}

/**
 * @see possible
 * @param current_length The sum of all pipe lengths tested in this branch
 *                       of the recursion tree.
 */
bool possible_bt(int x, LengthsArray lengths, CountsArray counts, int n, int current_length)
{
    int i = 0;
    bool possible = false;
    while (i < n && !possible) {
        // if there are more pipes with
        // the "current" length (lengths[i])

        if (current_length == x) {
            return true;
        }

        if (counts[i] > 0) {
            // use it
            --counts[i];
            possible = possible_bt(x, lengths, counts, n, current_length + lengths[i]);
            // if that pipe didn't work,
            if (!possible) {
                // "un-use" (// backtrack) it
                ++counts[i];
            }
        }
        ++i;
    }

    return possible;
}

/**
 * @param x The total length the pipes should add up to.
 * @param lengths The array of available pipe lengths.
 * @param counts The amount of pipes available for a specific length.
 * @param n The number of pipe-lengths. (Also used to index counts to
 *          retrieve the number of available pipes per length.)
 * @param f The function that should be used for checking the possibilities.
 * @return True if the function f found a way to combine the available (counts)
 *          pipelengths (lenghts) so that the total length is x.
 */
bool possible(int x, LengthsArray lengths, int *const counts, int n, SolutionFunction f)
{
    // Implementation note: I am not using the original counts array
    // since the solution function f mutates it. To avoid a potential
    // source of errors, I decided to use a copy. That way users will
    // not have to take the mutation into account when using this function.

    // Also using possible_it is potentially dangerous since its interface differs
    // from the one typedef'd as SolutionFunction. I would not do this (and also not
    // recommend doing so) in real world applications. Because, however, the only
    // goal of this exercise is to demonstrate different implementations, I'd say
    // it's fine.
    int *counts_cpy = (int *) malloc(sizeof(int) * (size_t) n);
    for (int i = 0; i < n; ++i) {
        counts_cpy[i] = counts[i];
    }
    bool possible = f(x, lengths, counts, n, 0);
    free(counts_cpy);
    return possible;
}

/*
 *
 * Tests
 *
 */

void assertBool(bool expected, bool got, const char *name)
```

```
{
    bool success = expected == got;
    printf("@\tTest \"%s\" %s.", name, success ? "succeeded" : "failed");
    if (!success) {
        printf(" Expected %d, got %d.", expected, got);
    }
    printf("\n");
}

#define INT_ARR(x) (int[x])

static time_t timer;

void peak()
{
    time_t diff = clock() - timer;
    printf("Time passed: %.2fms\n", ((float) diff) / CLOCKS_PER_SEC * 1000);
    timer = clock();
}

void test_it()
{
    timer = clock();
    /* Possible */
    // ugly (and unsafe!) typecasts, see implementation note in function "possible"!
    assertBool(true, possible(4,
                              INT_ARR(1) {1},
                              INT_ARR(1) {5}, 1, (SolutionFunction) possible_it),
               "Possible for 1 using iteration");
    peak();
    assertBool(true, possible(3,
                              INT_ARR(2) {1, 2},
                              INT_ARR(2) {2, 1}, 2, (SolutionFunction) possible_it),
               "Possible for 2 using iteration");
    peak();
    assertBool(true, possible(6,
                              INT_ARR(3) {1, 2, 3},
                              INT_ARR(3) {1, 1, 1}, 3, (SolutionFunction) possible_it),
               "Possible for 3 using iteration");
    peak();

    /* n > 3 */
    assertBool(true, possible(10,
                              INT_ARR(4) {1, 2, 3, 99},
                              INT_ARR(4) {1, 1, 3, 99}, 4, (SolutionFunction) possible_it),
               "Possible for 4 using iteration");
    peak();

    /* Impossible */
    assertBool(false, possible(10,
                               INT_ARR(1) {1},
                               INT_ARR(1) {5}, 1, (SolutionFunction) possible_it),
               "Impossible for 1 using iteration");
    peak();
    assertBool(false, possible(10,
                               INT_ARR(2) {1, 2},
                               INT_ARR(2) {2, 1}, 2, (SolutionFunction) possible_it),
               "Impossible for 2 using iteration");
    peak();
    assertBool(false, possible(10,
                               INT_ARR(3) {1, 2, 3},
                               INT_ARR(3) {1, 1, 1}, 3, (SolutionFunction) possible_it),
               "Impossible for 3 using iteration");
    peak();

    /* n > 4 */
    assertBool(false, possible(7,
                               INT_ARR(5) {1, 2, 3, 4, 5},
                               INT_ARR(5) {2, 0, 1, 1, 0}, 5, (SolutionFunction) possible_it),
               "Impossible for 4 using iteration");
    peak();
}

void test_rec()
```

```
{
    timer = clock();
    /* Possible */
    assertBool(true, possible(7,
                              INT_ARR(5) {1, 2, 3, 4, 5},
                              INT_ARR(5) {2, 0, 1, 1, 0}, 5, possible_rec),
               "Possible for 5 using recursion");
    peak();
    assertBool(true, possible(35,
                              INT_ARR(7) {1, 2, 3, 4, 5, 10, 15},
                              INT_ARR(7) {2, 0, 1, 1, 0, 1, 2}, 7, possible_rec),
               "Possible for 7 using recursion");
    peak();
    assertBool(true, possible(55,
                              INT_ARR(10) {1, 2, 3, 4, 5, 6, 7, 8, 9, 10},
                              INT_ARR(10) {1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 10, possible_rec),
               "Possible for 10 using recursion");
    peak();

    /* Impossible */
    assertBool(false, possible(10,
                               INT_ARR(5) {1, 2, 3, 4, 5},
                               INT_ARR(5) {2, 0, 1, 1, 0}, 5, possible_rec),
               "Impossible for 5 using recursion");
    peak();
    assertBool(false, possible(100,
                               INT_ARR(7) {1, 2, 3, 4, 5, 10, 15},
                               INT_ARR(7) {2, 0, 1, 1, 0, 1, 2}, 7, possible_rec),
               "Impossible for 7 using recursion");
    peak();
    assertBool(false, possible(56,
                               INT_ARR(10) {1, 2, 3, 4, 5, 6, 7, 8, 9, 10},
                               INT_ARR(10) {1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 10, possible_rec),
               "Impossible for 10 using recursion");
    peak();
}

void test_bt()
{
    timer = clock();
    /* Possible */
    assertBool(true, possible(7,
                              INT_ARR(5) {1, 2, 3, 4, 5},
                              INT_ARR(5) {2, 0, 1, 1, 0}, 5, possible_bt),
               "Possible for 5 using backtracking");
    peak();
    assertBool(true, possible(35,
                              INT_ARR(7) {1, 2, 3, 4, 5, 10, 15},
                              INT_ARR(7) {2, 0, 1, 1, 0, 1, 2}, 7, possible_bt),
               "Possible for 7 using backtracking");
    peak();
    assertBool(true, possible(55,
                              INT_ARR(10) {1, 2, 3, 4, 5, 6, 7, 8, 9, 10},
                              INT_ARR(10) {1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 10, possible_bt),
               "Possible for 10 using backtracking");
    peak();

    /* Impossible */
    assertBool(false, possible(10,
                               INT_ARR(5) {1, 2, 3, 4, 5},
                               INT_ARR(5) {2, 0, 1, 1, 0}, 5, possible_bt),
               "Impossible for 5 using backtracking");
    peak();
    assertBool(false, possible(100,
                               INT_ARR(7) {1, 2, 3, 4, 5, 10, 15},
                               INT_ARR(7) {2, 0, 1, 1, 0, 1, 2}, 7, possible_bt),
               "Impossible for 7 using backtracking");
    peak();
    assertBool(false, possible(56,
                               INT_ARR(10) {1, 2, 3, 4, 5, 6, 7, 8, 9, 10},
                               INT_ARR(10) {1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 10, possible_bt),
               "Impossible for 10 using backtracking");
    peak();
}
```

```
int main()
{
    test_it();
    test_rec();
    test_bt();
    return EXIT_SUCCESS;
}
```

Metriken

Methodik	Möglichkeit	Problemgröße	Zeit in ms
Iterativ	Möglich	1	0.0
		2	0.0
		3	0.0
		4	0.0
	Unmöglich	1	0.0
		2	0.0
		3	**2.0**
		4	0.0
Rekursiv	Möglich	5	0.0
		7	0.0
		10	0.0
	Unmöglich	5	0.0
		7	**1.0**
		10	**422.0**
Backtracking	Möglich	5	0.0
		7	0.0
		10	**1.0**
	Unmöglich	5	**1.0**
		7	0.0
		10	**420.0**

Testergebnisse

```
"C:\Users\Niklas Vest\Documents\FH\fh-hgb-ws1819\swo\ue03\cmake-build-debug\pipe.exe"
@Test "Possible for 1 using iteration" succeeded.
Time passed: 0.00ms
@Test "Possible for 2 using iteration" succeeded.
Time passed: 0.00ms
@Test "Possible for 3 using iteration" succeeded.
Time passed: 0.00ms
The iterative version will only consider the first 3 pipe lengths.
@Test "Possible for 4 using iteration" succeeded.
Time passed: 0.00ms
@Test "Impossible for 1 using iteration" succeeded.
Time passed: 1.00ms
@Test "Impossible for 2 using iteration" succeeded.
Time passed: 0.00ms
@Test "Impossible for 3 using iteration" succeeded.
Time passed: 0.00ms
The iterative version will only consider the first 3 pipe lengths.
@Test "Impossible for 4 using iteration" succeeded.
Time passed: 0.00ms
@Test "Possible for 5 using recursion" succeeded.
Time passed: 0.00ms
@Test "Possible for 7 using recursion" succeeded.
Time passed: 1.00ms
@Test "Possible for 10 using recursion" succeeded.
Time passed: 0.00ms
@Test "Impossible for 5 using recursion" succeeded.
Time passed: 0.00ms
@Test "Impossible for 7 using recursion" succeeded.
Time passed: 0.00ms
@Test "Impossible for 10 using recursion" succeeded.
Time passed: 427.00ms
@Test "Possible for 5 using backtracking" succeeded.
Time passed: 0.00ms
@Test "Possible for 7 using backtracking" succeeded.
Time passed: 0.00ms
@Test "Possible for 10 using backtracking" succeeded.
Time passed: 0.00ms
@Test "Impossible for 5 using backtracking" succeeded.
Time passed: 0.00ms
@Test "Impossible for 7 using backtracking" succeeded.
Time passed: 0.00ms
@Test "Impossible for 10 using backtracking" succeeded.
Time passed: 434.00ms

Process finished with exit code 0
```

Sudoku

Lösungsidee

Man erstellt ein dreidimensionales Feld, welches auf der x- und y-Achse die Sudoku Felder beschreibt. Die z-Achse entspricht einem extra Feld der Größe 10, welches mit den Indices i von 0 bis 8 die Möglichkeit beschreibt, dass der Wert $i + 1$ für diese Feld eine Lösung darstellt. Der Index 10 ist eine Flagge um für ein beliebiges Feld zu speichern, ob dieses schon vorab ausgefüllt wurde oder ob es ein „zu lösendes Feld“ ist.
Der Lösungsalgorithmus versucht für jedes nicht vorab ausgefüllte Feld die Werte von 1 bis 9 in das Feld einzutragen. Ist dieser Wert bereits in derselben Reihe, Spalte oder Box *als Lösung eingetragen*, wird der nächste Wert getestet. Ein Wert ist genau dann *als Lösung eingetragen*, wenn in dem Feld p von möglichen Lösungswerten für genau einen Index i gilt, dass $p[i] = w$. Um die Indices für den Reihen-, Zeilen- und Box-Check zu berechnen verwende ich verschiedene Funktionen, die einen Index zwischen 0 und 8 auf jeweils eine ganze Reihe, Zeile oder Box abbilden. Beispielsweise kann ein Index i auf eine Spalte x mit folgender Funktion abgebildet werden:

$$I(x, i) \coloneqq i \times 9 + x$$

Die Funktion projiziert den Index auf ein eindimensionales Feld der Größe 81, was ein Überbleibsel meiner ursprünglichen Version ist. (Für Details siehe *each_row_in_column*, *each_column_in_row* und *each_field_in_box*).

Ist für ein Feld im Sudoku keine Lösung mehr möglich, werden durch Backtracking alte Lösungen invalidiert. Die Invalidierung geschieht durch das Setzen aller p_i für $0 \leq i \leq 8$ auf den Wert w. So wird sichergestellt, dass weitere Ausprägungen des Rekursionspfades diese Felder als ungelöst erkennen (i. e. bei der Suche nach einer Lösung für andere Felder nicht berücksichtigen). Sobald eine vollständige Lösung gefunden ist, wird diese in das originale (eindimensionale) Feld eingetragen, welches das ursprünglich ungelöste Sudoku beinhaltet.
Anmerkung: Die Implementierung mag etwas übertrieben erscheinen, ist aber gerade so abstrakt, als dass sie (zumindest für mich) sehr gut nachvollziehbar ist.

Implementierung

main.c

```c
#include <stdio.h>
#include <time.h>
#include "Sudoku.h"

static time_t timer;

void peak()
{
    time_t diff = clock() - timer;
    printf("Time passed: %.2fms\n", ((float) diff) / CLOCKS_PER_SEC * 1000);
    timer = clock();
}

int main()
{
    int wikipedia_sudoku[81] = {
            5, 3, 0,    0, 7, 0,    0, 0, 0,
            6, 0, 0,    1, 9, 5,    0, 0, 0,
            0, 9, 8,    0, 0, 0,    0, 6, 0,

            8, 0, 0,    0, 6, 0,    0, 0, 3,
            4, 0, 0,    8, 0, 3,    0, 0, 1,
            7, 0, 0,    0, 2, 0,    0, 0, 6,

            0, 6, 0,    0, 0, 0,    2, 8, 0,
```

```
            0, 0, 0,    4, 1, 9,    0, 0, 5,
            0, 0, 0,    0, 8, 0,    0, 7, 9
    };
    int easy[81] = {
            0, 0, 1,    5, 0, 6,    2, 0, 0,
            0, 5, 0,    0, 4, 2,    0, 9, 0,
            0, 0, 3,    7, 1, 0,    8, 0, 0,

            7, 0, 6,    0, 2, 0,    5, 0, 0,
            0, 2, 0,    0, 0, 0,    0, 3, 0,
            0, 0, 5,    0, 8, 0,    7, 0, 4,

            0, 0, 9,    0, 5, 7,    4, 0, 0,
            0, 1, 0,    3, 6, 0,    0, 7, 0,
            0, 0, 2,    1, 0, 8,    6, 0, 0
    };
    int medium[81] = {
            0, 0, 7,    0, 5, 0,    3, 9, 0,
            0, 0, 0,    0, 4, 2,    0, 7, 0,
            5, 2, 0,    9, 0, 0,    0, 0, 1,

            9, 0, 2,    0, 0, 0,    0, 6, 0,
            0, 1, 5,    6, 2, 4,    9, 3, 0,
            0, 6, 0,    0, 0, 0,    1, 0, 7,

            1, 0, 0,    0, 0, 5,    0, 8, 3,
            0, 3, 0,    4, 1, 0,    0, 0, 0,
            0, 5, 9,    0, 3, 0,    2, 0, 0
    };
    int hard[81] = {
            8, 0, 0,    0, 4, 0,    3, 0, 0,
            0, 4, 0,    7, 0, 0,    0, 8, 0,
            0, 0, 9,    0, 0, 8,    4, 7, 0,

            0, 0, 5,    3, 0, 0,    0, 2, 0,
            9, 0, 2,    5, 0, 4,    6, 0, 3,
            0, 8, 0,    0, 0, 9,    5, 0, 0,

            0, 9, 1,    8, 0, 0,    2, 0, 0,
            0, 2, 0,    0, 0, 1,    0, 3, 0,
            0, 0, 8,    0, 2, 0,    0, 0, 5
    };
    int very_hard[81] = {
            0, 0, 1,    8, 0, 5,    0, 0, 0,
            0, 0, 9,    0, 1, 0,    0, 5, 3,
            2, 5, 0,    0, 0, 0,    0, 6, 0,

            3, 2, 0,    0, 0, 0,    9, 0, 1,
            0, 1, 0,    7, 0, 2,    0, 3, 0,
            5, 0, 4,    0, 0, 0,    0, 2, 6,

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

    timer = clock();
    sudoku(wikipedia_sudoku);
    peak();
    sudoku(easy);
    peak();
    sudoku(medium);
    peak();
    sudoku(hard);
    peak();
    sudoku(very_hard);
    peak();
    print_sudoku(wikipedia_sudoku);
    print_sudoku(easy);
    print_sudoku(medium);
    print_sudoku(hard);
    print_sudoku(very_hard);
    return EXIT_SUCCESS;
}
```

Sudoku.h

```c
/*
 * A quick note to this head file:
 * Usually you would want to hide functions you don't
 * want to expose to the user. However I am pretty sure
 * whoever wants to dig into the code will be thankful to
 * find the entire documentation in the header file.
 */
#ifndef UE03_SUDOKU_H
#define UE03_SUDOKU_H

#include <stdlib.h>
#include "SudokuField.h"

/**
 * A 3D Array holding all possibilities.
 */
typedef bool * *const *const Possibilities;

/**
 * A function used to calculate an index given a fixed
 * item and an item changing every iteration.
 * @see check
 */
typedef size_t (*IndexCalculationFunction)(size_t, size_t);

/**
 * Applies the supplied solution to a specified field in the sudoku.
 * @param possibles The array of possibilities for the sudoku.
 * @param sudoku_field The sudoku field to mark as solved.
 * @param solution The value with which to fill the sudoku field.
 */
void set_solution(Possibilities possibles, const SudokuField *const sudoku_field, size_t solution);

/**
 * @param possibles The array of possibilities for the sudoku.
 * @param sudoku_field The field to get the value for.
 * @return The first possible value for the specified field.
 */
size_t first_possible_value(Possibilities possibles, const SudokuField *const sudoku_field);

/**
 * @param possibles The array of possibilities for the sudoku.
 * @param sudoku_field The field to check.
 * @return True if the specified field already has a definite solution.
 */
bool field_is_solved(Possibilities possibles, const SudokuField *const sudoku_field);

/**
 * @param possibles The array of possibilities for the sudoku.
 * @param sudoku_field The sudoku field for which to return the currently
 *          set solution.
 * @return The solution for the specified sudoku field.
 */
size_t get_solution(Possibilities possibles, const SudokuField *const sudoku_field);

/**
 * Sets all possibilities to true, meaning that for the specified
 * field, all numbers will be a possible solution.
 * @param possibles The array of possibilities for the solution.
 * @param sudoku_field The field to reset.
 */
void reset_field(Possibilities possibles, SudokuField *const sudoku_field);

/**
 * @param possibles The array of possibilities for the array.
 * @param sudoku_field The field to check.
 * @return True if the value for the specified field must not be changed.
 */
bool is_fixed(Possibilities possibles, SudokuField *const sudoku_field);

/**
```

```
 * @param arr The sudoku.
 * @return An array of possibilities for a given soduko.
 */
bool ***create_possibles(const int *const arr);

/**
 * Frees the memory allocated for the array of possible sudoku solutions.
 * @param possibles The array of possible solutions.
 */
void delete_possibles(bool ***possibles);

/**
 * A function returning the index for each column in a specified row.
 * @param row The row to inspect.
 * @param i The iterating value.
 * @see check
 */
size_t each_column_in_row(size_t row, size_t i);

/**
 * A function returning the index for each row in a specified column.
 * @param col The column to inspect.
 * @param i The iterating value.
 * @see check
 */
size_t each_row_in_column(size_t col, size_t i);

/**
 * A function returning the index for each item in a specified
 * sudoku 3*3 box.
 * @param box_index The index of the box to inspect.
 * @param i The iterating value.
 * @see check
 */
size_t each_field_in_box(size_t box_index, size_t i);

/**
 * Checks whether the supplied <i>value</i> is a solution in the area
 * covered by the indices produced by the function <i>calc_index</i>.
 * To check a whether a column x contains <i>value</i>,
 * call <code>check(possibles, value, each_row_in_column, x)</code>.
 * @param possibles The array of possibilities for the sudoku.
 * @param value The value for which to check whether it's a potential solution.
 * @param calc_index The function for calulating an index into the possibilities.
 *                       Use either <i>each_row_in_column</i>, <i>each_column_in_cow</i>
 *                       or <i>each_field_in_box</i>.
 * @param item The {item}th row, column or box will be checked.
 */
bool check(Possibilities possibles, size_t value, IndexCalculationFunction calc_index, size_t item);

/**
 * @param possibles The array of possibilities for the sudoku.
 * @param sudoku_field The field for which to check whether the solution is valid.
 * @param solution The solution to check.
 * @return True if the supplied solution is a valid solution for the
 *          specified sudoku index using the array of possibilites to
 *          check its validity.
 */
bool is_partial_solution(Possibilities possibles, SudokuField sudoku_field, size_t solution);

/**
 * Solves the sudoku represented by the supplied possibilities using recursive backtracking.
 * @param possibles The array of possibilities for the sudoku.
 * @param sudoku_field The field to process.
 * @param final_solution The original sudoku which will be filled with the final solution.
 */
void backtrace_solution(Possibilities possibles, SudokuField sudoku_field, int *const final_solution);

/**
 * Prints the supplied sudoku to the standard output
 * in a formatted manner.
 * @param sudoku The sudoku to print.
 */
void print_sudoku(const int *const sudoku);
```

```
/**
 * The entry point to the sudoku solving module.
 * @param squares An array using a row-major layout for the sudoku.
 */
void sudoku(int squares[]);

#endif //UE03_SUDOKU_H
```

Sudoku.c

```
#include <stdio.h>
#include "Sudoku.h"

void set_solution(Possibilities possibles, const SudokuField *const sudoku_field, size_t solution)
{
    for (size_t i = 0; i < 9; ++i) {
        possibles[sudoku_field->row][sudoku_field->col][i] = false;
    }
    possibles[sudoku_field->row][sudoku_field->col][solution - 1] = true;
}

size_t first_possible_value(Possibilities possibles, const SudokuField *const sudoku_field)
{
    size_t first_possible_found = false;
    size_t i = 0;
    while (i < 9 && !first_possible_found) {
        first_possible_found = possibles[sudoku_field->row][sudoku_field->col][i];
        ++i;
    }
    return i;
}

bool field_is_solved(Possibilities possibles, const SudokuField *const sudoku_field)
{
    size_t i = first_possible_value(possibles, sudoku_field);
    bool solved = i <= 9;
    while (i < 9 && solved) {
        solved = !possibles[sudoku_field->row][sudoku_field->col][i];
        ++i;
    }
    return solved;
}

size_t get_solution(Possibilities possibles, const SudokuField *const sudoku_field)
{
    return field_is_solved(possibles, sudoku_field) ?
           first_possible_value(possibles, sudoku_field) : 0;
}

void reset_field(Possibilities possibles, SudokuField *const sudoku_field)
{
    for (size_t i = 0; i < 9; ++i) {
        possibles[sudoku_field->row][sudoku_field->col][i] = true;
    }
}

bool is_fixed(Possibilities possibles, SudokuField *const sudoku_field)
{
    return possibles[sudoku_field->row][sudoku_field->col][9];
}

bool ***create_possibles(const int *const arr)
{
    // create row pointers
    bool ***possibles = (bool ***) malloc(sizeof(bool **) * 9);

    for (size_t row = 0; row < 9; ++row) {

        // create column pointers
        bool **cols = (bool **) malloc(sizeof(bool *) * 9);
        possibles[row] = cols;
```

```
        for (size_t col = 0; col < 9; ++col) {

            SudokuField current;
            current.row = row;
            current.col = col;

            // create array of possibilities + const flag
            possibles[row][col] = (bool *) malloc(sizeof(bool) * 10);

            size_t index = field_to_index(&current);
            // the 10th element in the array is a virtual const flag;
            // if it is set to true, the element must no be changed
            possibles[row][col][9] = arr[index] != 0;
            if (is_fixed(possibles, &current)) {
                set_solution(possibles, &current, (size_t) arr[index]);
            } else {
                reset_field(possibles, &current);
            }
        }
    }
    return possibles;
}

void delete_possibles(bool ***possibles)
{
    for (size_t row = 0; row < 9; ++row) {
        for (size_t col = 0; col < 9; ++col){
            free(possibles[row][col]);
        }
        free(possibles[row]);
    }
    free(possibles);
}

size_t each_column_in_row(size_t row, size_t i)
{
    return row * 9 + i;
}

size_t each_row_in_column(size_t col, size_t i)
{
    return i * 9 + col;
}

size_t each_field_in_box(size_t box_index, size_t i)
{
    // absolute box position within the sudoku
    size_t box_row = box_index / 3;
    size_t box_col = box_index % 3;

    // currently processed item within the box
    size_t row_within_box = i / 3;
    size_t col_within_box = i % 3;

    // absolute indices
    size_t row = box_row * 3 + row_within_box;
    size_t col = box_col * 3 + col_within_box;
    return row * 9 + col;
}

bool check(Possibilities possibles, size_t value, IndexCalculationFunction calc_index, size_t item)
{
    bool possible = true;
    size_t i = 0;
    while (i < 9 && possible) {
        SudokuField current = index_to_field(calc_index(item, i));
        possible =
                !(field_is_solved(possibles, &current) &&
                  (get_solution(possibles, &current) == value));
        ++i;
    }
    return possible;
}
```

```
bool is_partial_solution(Possibilities possibles, SudokuField sudoku_field, size_t solution)
{
    size_t box_row = sudoku_field.row / 3;
    size_t box_col = sudoku_field.col / 3; // intended division, NO MODULO!
    size_t box_index = box_row * 3 + box_col;
    return check(possibles, solution, each_column_in_row, sudoku_field.row) &&
           check(possibles, solution, each_row_in_column, sudoku_field.col) &&
           check(possibles, solution, each_field_in_box, box_index);
}

void solve_final(Possibilities possibles,  int *const dest)
{
    for (size_t i = 0; i < 81; ++i) {
        SudokuField current = index_to_field(i);
        dest[i] = (int) get_solution(possibles, &current);
    }
}

void backtrace_solution(Possibilities possibles, SudokuField sudoku_field, int *const final_solution)
{
    if (is_fixed(possibles, &sudoku_field)) {
        // if the field is fixed, we can skip it
        if (is_last_field(&sudoku_field)) {
            solve_final(possibles, final_solution);
        } else {
            backtrace_solution(possibles, next_field(&sudoku_field), final_solution);
        }
    } else {
        // otherwise we have to check all numbers
        for (size_t i = 1; i < 10; ++i) {

            // if we found a possible solution
            if (is_partial_solution(possibles, sudoku_field, i)) {
                // mark it as such
                set_solution(possibles, &sudoku_field, i);

                // if this was the last field, we are done
                if (is_last_field(&sudoku_field)) {
                    solve_final(possibles, final_solution);
                    return;
                }

                // otherwise we keep searching
                backtrace_solution(possibles, next_field(&sudoku_field), final_solution);
                // and backtrace this boe
                reset_field(possibles, &sudoku_field);
            }
        }
    }
}

void print_sudoku(const int *const sudoku)
{
    printf("===============================\n");
    for (size_t row = 0; row < 9; ++row) {
        for (size_t col = 0; col < 9; ++col) {
            printf("%d, ", sudoku[row * 9 + col]);
            if ((col + 1) % 3 == 0) {
                printf("\t");
            }
        }
        printf("\n");
        if ((row + 1) % 3 == 0) {
            printf("\n");
        }
    }
}

void sudoku(int *squares)
{
    bool ***possibles = create_possibles(squares);
    backtrace_solution(possibles, (SudokuField) {0, 0}, squares);
    delete_possibles(possibles);
}
```

SudokuField.h

```c
#ifndef UE03_SUDOKUFIELD_H
#define UE03_SUDOKUFIELD_H

#include <stdlib.h>

/**
 * A neat lil bool.
 */
typedef enum
{
    false, true
} bool;

/**
 * A 2D point representing one field in a sudoku.
 */
typedef struct
{
    size_t row;
    size_t col;
} SudokuField;

/**
 * @param f The field to project.
 * @return The 2D index (on a 9*9 basis) projected
 * onto 1D array.
 */
size_t field_to_index(const SudokuField *const f);

/**
 * @param index The index to project.
 * @return The 1D index projected onto a 9 * 9
 *          matrix.
 */
SudokuField index_to_field(size_t index);

/**
 * @param prev The predecessor to the desired field.
 * @return The field to the right of the supplied field
 *          if the column < 8 or the first column in the
 *          next row if the column = 8.
 */
SudokuField next_field(const SudokuField *const prev);

/**
 * @param field The field to check.
 * @return True if the row AND the column are equal to 8.
 */
bool is_last_field(const SudokuField *const field);

#endif //UE03_SUDOKUFIELD_H
```

SudokuField.c

```c
#include "SudokuField.h"

size_t field_to_index(const SudokuField *const f)
{
    return f->row * 9 + f->col;
}

SudokuField index_to_field(size_t index)
{
    SudokuField f;
    f.row = index / 9;
    f.col = index % 9;
    return f;
}
```

```
SudokuField next_field(const SudokuField *const prev)
{
    SudokuField next;
    bool wraps = prev->col == 8;
    next.row = wraps ? prev->row + 1 : prev->row;
    next.col = wraps ? 0 : prev->col + 1;
    return next;
}

bool is_last_field(const SudokuField *const field)
{
    return field->row == 8 && field->col == 8;
}
```

Metriken

Schwierigkeit	Zeit in ms
Wikipedia (?)	46.0
Einfach	8.0
Mittel	5.0
Schwer	**34.0**
Sehr schwer	**2.0**

Tests

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

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

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

Abbildung 6 Wikipedia Sudoku

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

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

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

Abbildung 5 Mittel schweres Sudoku

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

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

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

Abbildung 2 Einfaches Sudoku

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

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

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

Abbildung 4 Schweres Sudoku

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

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

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

Abbildung 3 Sehr schweres Sudoku