Ausarbeitung Übung 03

Table of Contents

[Pipeline 2](#_Toc528447086)

[Lösungsidee 2](#_Toc528447087)

[Implementierung 2](#_Toc528447088)

[Metriken 7](#_Toc528447089)

[Testergebnisse 8](#_Toc528447090)

[Sudoku 9](#_Toc528447091)

[Lösungsidee 9](#_Toc528447092)

[Implementierung 9](#_Toc528447093)

[main.c 9](#_Toc528447094)

[Sudoku.h 11](#_Toc528447095)

[Sudoku.c 13](#_Toc528447096)

[SudokuField.h 16](#_Toc528447097)

[SudokuField.c 16](#_Toc528447098)

[Metriken 17](#_Toc528447099)

[Tests 17](#_Toc528447100)

# Pipeline

## Lösungsidee

1. **Iterativ**: In der äußersten Schleife werden alle möglichen Vielfachen der ersten Rohrlänge ausprobiert, „solange der Vorrat () reicht“. Für jede Teillänge  
   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 , da man für Rohre auch 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.

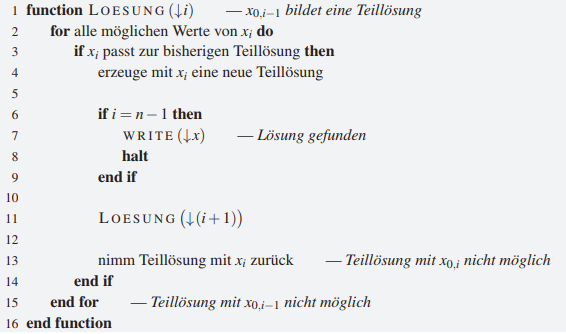


Abbildung 1 Backtracking Schablone von Peter Kulczycki

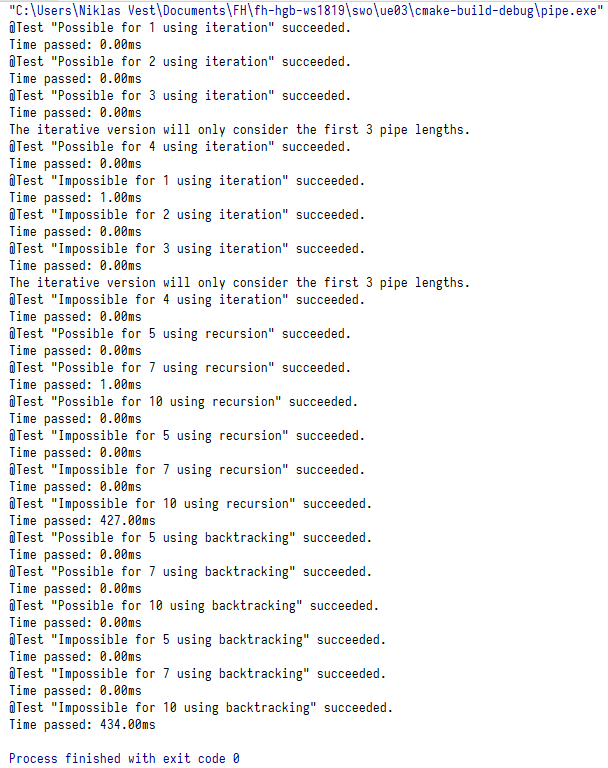
## Implementierung

#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



# 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 von 0 bis 8 die Möglichkeit beschreibt, dass der Wert 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 von möglichen Lösungswerten für genau einen Index gilt, dass . 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 auf eine Spalte mit folgender Funktion abgebildet werden:

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 für auf den Wert . 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

#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

*/\*  
 \* 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

#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

#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

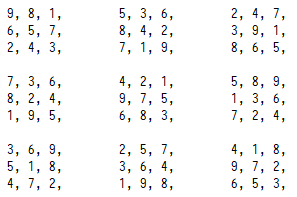


Abbildung Einfaches Sudoku

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

## Tests

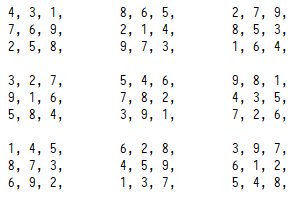


Abbildung Sehr schweres Sudoku

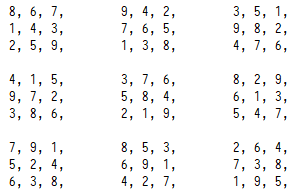


Abbildung Schweres Sudoku

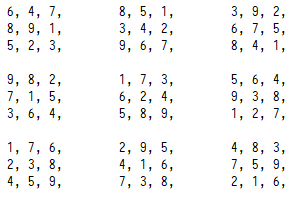


Abbildung Mittel schweres Sudoku

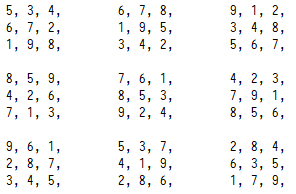


Abbildung Wikipedia Sudoku