Inhaltsverzeichnis

[1. Pipeline 2](#_Toc23114717)

[1.1. Lösungsidee 2](#_Toc23114718)

[1.1.1. Iterativ 2](#_Toc23114719)

[1.1.2. Rekursiv 2](#_Toc23114720)

[1.1.3. Backtracking 2](#_Toc23114721)

[1.2. Code 3](#_Toc23114722)

[1.3. Testfälle 6](#_Toc23114723)

[2. Sudoku 7](#_Toc23114724)

[2.1. Lösungsidee 7](#_Toc23114725)

[2.2. Code 9](#_Toc23114726)

[2.3. Testfälle 15](#_Toc23114727)

# Pipeline

## Lösungsidee

### 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 >= n ist.

### 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.

### 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.

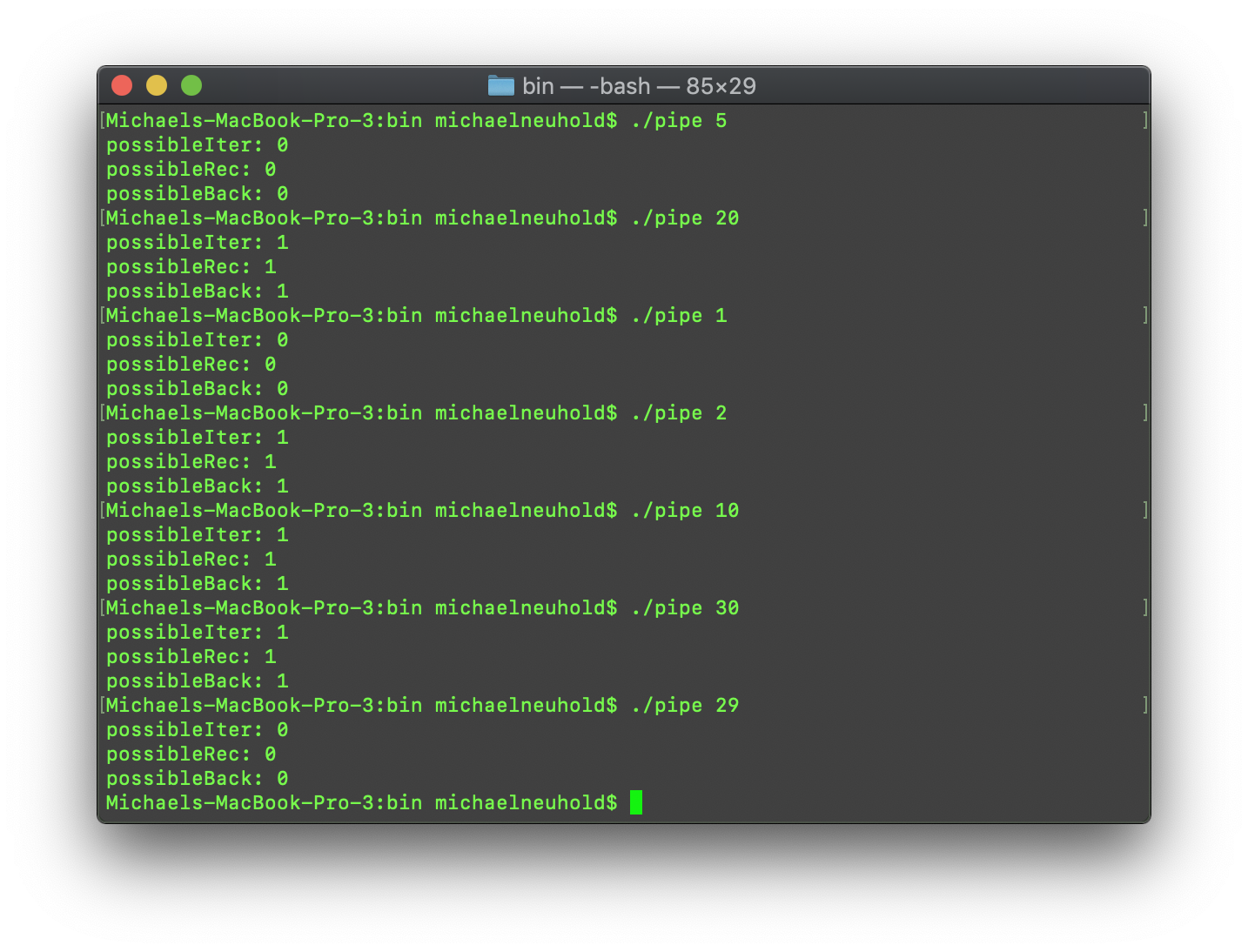
## Code

|  |
| --- |
| #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;  } |

|  |
| --- |
| #ifndef 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 safe "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;  } |

## Testfälle

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

# Sudoku

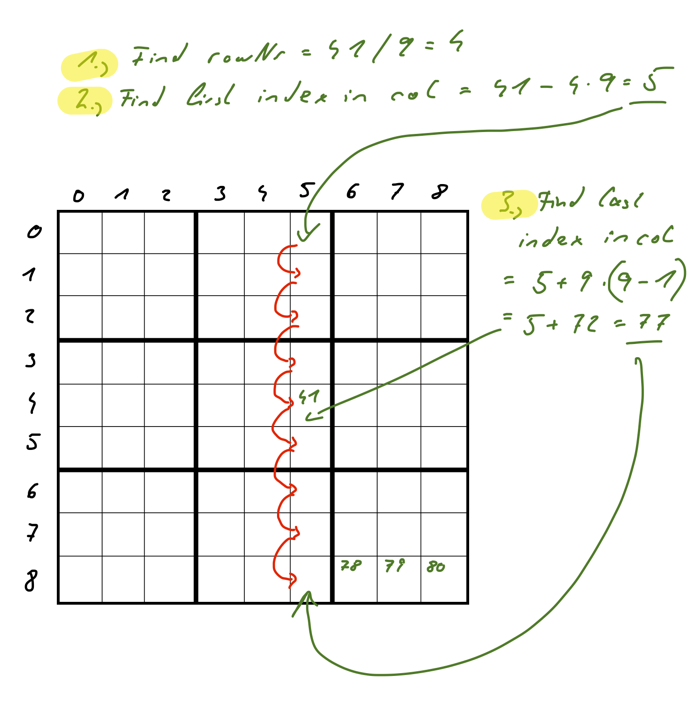
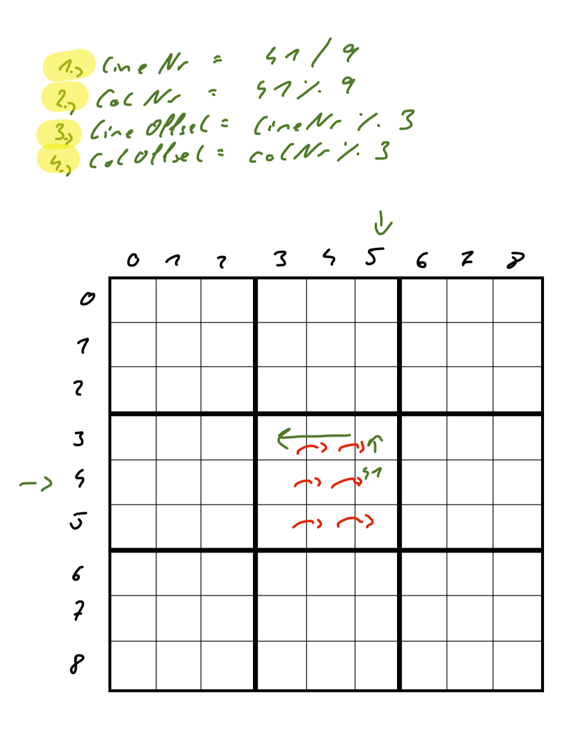
## Lösungsidee

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.

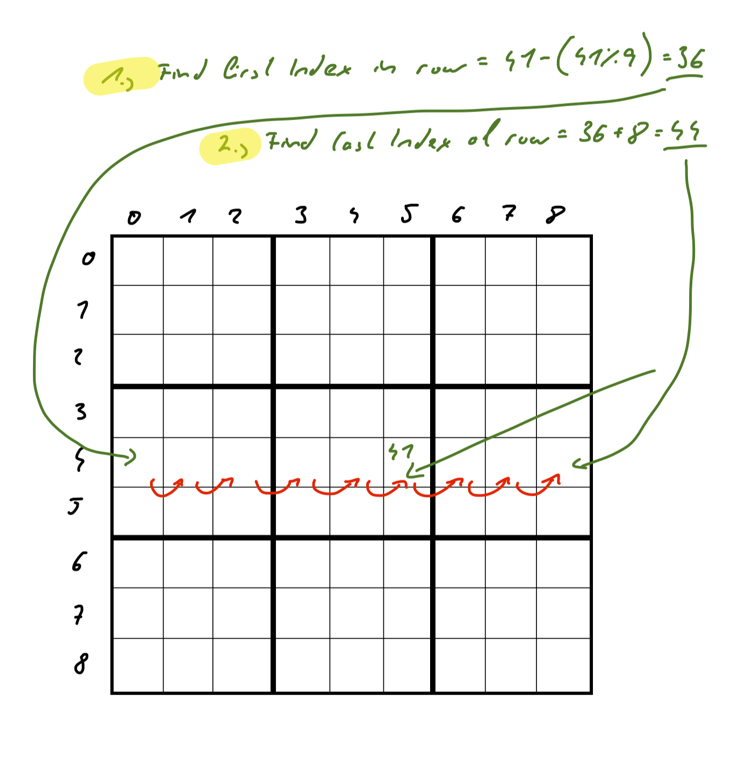
In der Funktion sudoku\_solve wird das gewählte Sudoku zunächst ausgegeben und im Anschluss wird eine rekursive Funktion aufgerufen, die TRUE liefert wenn das Sudoku gelöst werden konnte und FALSE liefert wenn es nicht gelöst werden konnte. In dieser Rekursiven Funktion wird zunächst nach dem nächsten 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.

## 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 betweene 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\_MILLISEC);  } |

## 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.



