SWO3	Übung zu Softwareentwicklung mit klassischen Sprachen und Bibliotheken 3	WS 2018/19, Angabe 3

☒ Gruppe 1 (J. Heinzelreiter)

☐ Gruppe 2 (M. Hava) Name: PAPESH Konstantin Aufwand [h]: 14

☐ Gruppe 3 (P. Kulczycki) Übungsleiter/Tutor: _____ Punkte: _____

Beispiel	Lösungsidee (max. 100%)	Implement. (max. 100%)	Testen (max. 100%)
1 (40 P)	100	100	100
2 (60 P)	100	100	100

Beispiel 1: Pipeline (`src/pipe/`)

Ein Installateur muss eine Strecke von x Metern mit Rohren überbrücken. Er hat ein Lager mit verschieden langen Rohren zur Verfügung und möchte wissen, ob er diese x Meter erreichen kann, ohne eines der lagernden Rohre zersägen zu müssen. Implementieren Sie für den Installateur eine Funktion

```
bool possible (int const x, int const lengths [], int const counts [], int const n);
```

die dem Installateur diese Frage beantwortet. Dabei seien `lengths` und `counts` zwei Felder mit jeweils `n` Elementen, die für die jeweiligen Rohrlängen samt den entsprechend verfügbaren Stückzahlen stehen.

Implementieren Sie `possible` in drei Varianten. Einmal als iterative Funktion (mit `n` geschachtelten `for`-Schleifen), einmal als rekursive Funktion und einmal als Funktion, die nach dem Backtracking-Schema arbeitet.

Testen Sie Ihre Funktionen ausführlich und geben Sie auch die (empirisch ermittelten) Laufzeiten für verschiedene Problemgrößen an.

Beispiel 2: Sudoku (`src/sudoku/`)

Implementieren Sie eine Funktion

```
void sudoku_solve (int squares []);
```

die ein in `squares` (der Größe $9 \cdot 9 = 81$) gegebenes Sudoku nach dem Backtracking-Schema löst. Entnehmen Sie der Seite https://en.wikipedia.org/wiki/Sudoku alles Wesentliche über den Aufbau von Sudokus. Testen Sie Ihre Funktion ausführlich und geben Sie auch die (empirisch ermittelten) Laufzeiten für Sudokus mit verschiedenen Schwierigkeitsgraden an.

SWO31 & SWB31 Softwareentwicklung mit klassischen Sprachen und Bibliotheken – WS 2018/19
Übungsabgabe 3

Konstantin Papesh

28. Oktober 2018

3.1 Pipeline

3.1.1 Lösungsidee

Es soll ermittelt werden, ob eine gegebene Länge mit einer vorhandenen Anzahl an Rohren erreicht werden kann, ohne dass ein Rohr gekürzt werden muss. Es sind dabei drei verschiedene Lösungen auszuarbeiten:

- Iterativ
- Rekursiv
- Rekursiv mit Backtracking

Da bei der iterativen Lösung n festgelegte For-Schleifen vorhanden sein müssen, muss auch die Anzahl der verschiedenen Rohrtypen von Beginn an klar sein. Daher wird das Array fest im Code einprogrammiert und die erforderliche Anzahl an For-Schleifen in die iterative Funktion geschrieben. Für den rekursiven Approach wird so lange getestet, bis keine Rohre mehr vorhanden sind. Dabei wird rekursiv über das Array gegangen, bis man am letzten Element angekommen ist. Dann wird diese Rohre so lange zusammengesteckt, entweder bis die Länge erreicht ist oder bis keine Rohre mehr vorhanden sind. Sollte Zweiteres der Fall sein wird das nächstegrößere Rohre dazugenommen und wieder zusammengezählt. Dieser Vorgang wiederholt sich bis die erforderliche Länge auftritt oder keine Rohre mehr insgesamt vorhanden sind. Beim Backtracking wird ähnlich vorgegangen wie im rekursiven Approach, nur wird abgebrochen, sobald die Länge der zusammengesteckten Rohre die gesuchte Länge überschreiten, und direkt das nächstlängere Rohr hinzugefügt. Dadurch wird Zeit gespart.

3.1.2 Implementierung

Listing 3.1: main.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdbool.h>
4 #include <time.h>
5
6 void copyTo(int const a[], int b[], int const n){
7   for(int i = 0; i < n; i++){
8     b[i] = a[i];
9   }
10 }
11
12 bool possibleIterative(int const x, int const lengths[], int const counts[], int
       const n) {
13   for(int a = 0; a <= counts[0]; a++){
14         for(int b = 0; b <= counts[1]; b++){
15             for(int c = 0; c <= counts[2]; c++){
16                 for(int d = 0; d <= counts[3]; d++){
17                     for(int e = 0; e <= counts[4]; e++){
18                         for(int f = 0; f <= counts[5]; f++){
19                             if(a*lengths[0] + b*lengths[1] + c*lengths[2] + d*
       lengths[3] + e*lengths[4] + f*lengths[5] == x)
20                                 return true;
21                         }
22                     }
23                 }
24             }
25         }
26   }
27   return false;
28 }
29
30 bool possibleRecursive(int const x, int const lengths[], int const counts[], int
       const n) {
31   int i;
32   int tempCounts[n];
33   int tempX;
34   for(i = 0; counts[i] <= 0 && i < n; i++); //get first stocked pipe length
35   if(i == n) // no pipes left
36         return false;
37   else if(lengths[i] == x) //if current pipe is fitted length is reached
38         return true;
39
40   copyTo(counts, tempCounts, n);
41   tempCounts[i] = 0; //reduce count (take pipe from stock)
42   for(int count = 0; count <= counts[i]; count++){
43         tempX = x - lengths[i]*count;
44         if(possibleRecursive(tempX, lengths, tempCounts, n))
45             return true;
46   }
47     return false;
48 }
49 bool possibleRecursiveWithBT(int const x, int const lengths[], int const counts[],
       int const n) {
```

```
50      int i;
51      int tempCounts[n];
52      int tempX;
53      for(i = 0; counts[i] <= 0 && i < n; i++); //get first stocked pipe length
54      if(i == n) // no pipes left
55          return false;
56      else if(lengths[i] == x) //if current pipe is fitted  length  is  reached
57          return true;
58
59      copyTo(counts, tempCounts, n);
60      tempCounts[i] = 0; //reduce count (take pipe from stock)
61      for(int count = 0; count <= counts[i]; count++){
62          tempX = x - lengths[i]*count;
63          if(tempX < 0)    //backtracking --> we are already above x in length
64              return false;
65          if(possibleRecursive(tempX, lengths, tempCounts, n))
66              return true;
67      }
68      return false;
69  }
70
71  int main() {
72      clock_t start_t, endIt_t, endRec_t, endRecBT_t;
73      bool posIt, posRec, posRecBT;
74    int const x = -2;
75    int const lengths[] = {3, 4, 9, 11, 13, 15};
76    int const counts[] = {20, 20, 10, 10, 10, 8};
77    int const n = 6;
78
79    start_t = clock();
80      posIt = possibleIterative(x, lengths, counts, n);
81    endIt_t = clock();
82      posRec = possibleRecursive(x, lengths, counts, n);
83      endRec_t = clock();
84      posRecBT = possibleRecursiveWithBT(x, lengths, counts, n);
85      endRecBT_t = clock();
86      printf("Total time iterative is %f with result: %i\n", ((double)(endIt_t -
        start_t)/CLOCKS_PER_SEC), posIt);
87      printf("Total time recursive is %f with result: %i\n", ((double)(endRec_t -
        endIt_t)/CLOCKS_PER_SEC), posRec);
88      printf("Total time recursive with backtracking is %f with result: %i\n", ((
        double)(endRecBT_t - endRec_t)/CLOCKS_PER_SEC), posRecBT);
89    return 0;
90  }
```

3.1.3 Testen

```
int const x = -2;
int const lengths[] = {3, 4, 9, 11, 13, 15};
int const counts[] = {20, 20, 10, 10, 10, 8};
int const n = 6;
```

Abbildung 3.1: x und das Pipe-Array im Quelltext

```
khp@KLS ~/g/f/w/s/e/a/s/p/build (master)> ./pipe
Total time iterative is 0.000006 with result: 1
Total time recursive is 0.000035 with result: 1
Total time recursive with backtracking is 0.000034 with result: 1
```

Abbildung 3.2: Laufzeiten bei x=124

```
khp@KLS ~/g/f/w/s/e/a/s/p/build (master)> ./pipe
Total time iterative is 0.000003 with result: 1
Total time recursive is 0.000074 with result: 1
Total time recursive with backtracking is 0.000072 with result: 1
```

Abbildung 3.3: Laufzeiten bei x=256

```
khp@KLS ~/g/f/w/s/e/a/s/p/build (master)> ./pipe
Total time iterative is 0.027029 with result: 0
Total time recursive is 0.168232 with result: 0
Total time recursive with backtracking is 0.166463 with result: 0
```

Abbildung 3.4: Laufzeiten bei x=1000. Diese Länge kann nicht erreicht werden.

```
khp@KLS ~/g/f/w/s/e/a/s/p/build (master)> ./pipe
Total time iterative is 0.027728 with result: 0
Total time recursive is 0.167133 with result: 0
Total time recursive with backtracking is 0.007825 with result: 0
```

Abbildung 3.5: Laufzeiten bei x=1. Da es kein so kurzes Rohr gibt wird auch hier 0 returned.

```
khp@KLS ~/g/f/w/s/e/a/s/p/build (master)> ./pipe
Total time iterative is 0.029172 with result: 0
Total time recursive is 0.167446 with result: 0
Total time recursive with backtracking is 0.000001 with result: 0
```

Abbildung 3.6: Laufzeiten bei x=-1. Diese Länge kann generell nie erreicht werden.

3.2 Sudoku

3.2.1 Lösungsidee

Es muss ein gegebenes Sudoku gelöst werden. Dabei wird über die Kommandozeile das Sudoku eingelesen. Das Sudoku muss dabei aus einer einfachen Zahlenreihenfolge bestehen welche das Sudoku wiederspiegelt. Dieses Sudoku wird dann vom einfachen String in ein 2-dimensionales Array geschrieben, dies macht die Verarbeitung und Lösung davon einfacher.
Danach wird dieses Array überflogen und alle unmöglichen Zahlen nach den Regeln von Sudoku für jeden Square in ein externes Array geschrieben. Dies ermöglicht es uns, für ein bestimmtes Feld alle noch möglichen Zahlen zu erfahren. Mit diesem Approach kann auch erkannt werden, ob das momentane Sudoku überhaupt noch gelöst werden kann. Denn wenn keine Zahl für ein Feld mehr möglich ist, aber dennoch noch leere Felder vorhanden sind, ist das Sudoku unlösbar und es muss gebacktrackt werden. Weiters wird überprüft, ob ein Feld die einzige Möglichkeit für eine Zahl ist. Dies bezieht sich auf den 3x3 Würfel des Sudokus. Ist die Zahl in den anderen beiden Spalten bzw. Zeilen vorhanden, muss in dem jeweiligen Feld die Zahl sein.
Hat das Programm bereits beide Überprüfungen gemacht und ist noch keine Lösung vorhanden, muss mithilfe von Backtracking gearbeitet werden. Dabei wird ein Feld, welches mehr als eine Zahl möglicht hat, auf einer der Möglichkeiten gesetzt und mit diesem Sudoku dann weitergearbeitet. Führt das Sudoku mit der eingesetzten Zahl nicht zur Lösung, wird gebacktrackt und eine andere mögliche Zahl verwendet. Dies geschied so lange, bis entweder alle Möglichkeiten erschöpft sind (Sudoku unlösbar) oder eine Lösung gefunden wird. Je nach Aufbau und Schwierigkeit des Sudokus variert die Lösungdauer.

3.2.2 Implementierung

Listing 3.2: main.c

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdbool.h>
4  #include <string.h>
5  #include <time.h>
6
7  #define ARG_COUNT 2
8  #define SIDE_LENGTH 9
9  #define BLOCK_LENGTH 3
10 #define MAX_NUM 9
11
12 bool sudoku_solve2(int squares[SIDE_LENGTH][SIDE_LENGTH]);
13 int squaresPossible[SIDE_LENGTH][SIDE_LENGTH][MAX_NUM];
14
15 void printSudoku(int squares[SIDE_LENGTH][SIDE_LENGTH]){ //prints 2d sudokus
16   for(int i = 0; i < SIDE_LENGTH; i++){
17     for(int j = 0; j < SIDE_LENGTH; j++){
18       printf("|%i", squares[i][j]);
19     }
20     printf("|\n");
```

```
21   }
22 }
23
24 void convertSudoku(int squares[], int squares2d[SIDE_LENGTH][SIDE_LENGTH]){ //
       converts sudoku from 1-dim to 2-dim.
25   int n = 0;
26   for(int i = 0; i < SIDE_LENGTH; i++){
27     for(int j = 0; j < SIDE_LENGTH; j++){
28       squares2d[i][j] = squares[n];
29       n++;
30     }
31   }
32 }
33
34 void inputNumbersIntoSudoku(int *squares, char *input){
35   for(int i = 0; i < SIDE_LENGTH*SIDE_LENGTH; i++){
36     squares[i] = (input[i])-48;
37   }
38 }
39
40 void initSquaresPossible(){ //initialises the squaresPossibleArray, so sets every number in
       every square possible .
41   for(int i = 0; i < SIDE_LENGTH; i++){
42     for(int j = 0; j < SIDE_LENGTH; j++){
43       for(int k = 0; k < MAX_NUM; k++){
44         squaresPossible[i][j][k] = true;
45       }
46     }
47   }
48 }
49
50 bool getPossibility(int row, int col, int num){ //get possiblity for a given square
51   return squaresPossible[row][col][num-1]; //NUM-1 because arrays start at 0
52 }
53
54 void setPossibility(int row, int col, int num, bool possibility){ //ATTENTION:
       Use setSquare if you want to change a square.
55   squaresPossible[row][col][num-1] = possibility; //NUM-1 because arrays start at 0
56 }
57
58 void setImpossibleForSquare(int row, int col, int num){ //removes num from all
       possibilities of other squares.
59   for(int i = 0; i < SIDE_LENGTH; i++) {
60     setPossibility(i, col, num, false);
61   }
62   for(int j = 0; j < SIDE_LENGTH; j++) {
63     setPossibility(row, j, num, false);
64   }
65   int rowStart = row / BLOCK_LENGTH;
66   int colStart = col / BLOCK_LENGTH;
67   rowStart = rowStart * BLOCK_LENGTH;
68   colStart = colStart * BLOCK_LENGTH;
69   for(int i = rowStart; i < rowStart + BLOCK_LENGTH; i++){
70     for(int j = colStart; j < colStart + BLOCK_LENGTH; j++){
71       setPossibility(i, j, num, false);
72     }
73   }
```

```
74   for(int i = 1; i < MAX_NUM+1; i++) { // as there is a number in this field set all other
       numbers false too.
75     setPossibility(row, col, i, false);
76   }
77 }
78
79 void removeImpossibleNumbers(int squares[SIDE_LENGTH][SIDE_LENGTH]){ //checks sudoku
       for the first time. Should only be called once.
80   for(int i = 0; i < SIDE_LENGTH; i++) {
81     for(int j = 0; j < SIDE_LENGTH; j++) {
82       if(squares[i][j] != 0)
83         setImpossibleForSquare(i, j, squares[i][j]);
84     }
85   }
86 }
87
88 void setSquare(int squares[SIDE_LENGTH][SIDE_LENGTH], int row, int col, int num){ //
       interface to set square to a number.
89   squares[row][col] = num;
90   setImpossibleForSquare(row, col, num);
91 }
92
93 bool checkForSingle(int squares[SIDE_LENGTH][SIDE_LENGTH]){ //checks if there is a single
       possibility in the sudoku.
94   bool found = false;
95   for(int i = 0; i < SIDE_LENGTH; i++){
96     for(int j = 0; j < SIDE_LENGTH; j++){
97       int possibleNumbersCount = 0;
98       int possibleNumber = 0;
99       for(int num = 1; num < MAX_NUM+1; num++){
100        if(getPossibility(i,j,num)){
101          possibleNumbersCount++;
102          possibleNumber = num;
103        }
104      }
105      if(possibleNumbersCount == 1){
106        setSquare(squares, i, j, possibleNumber);
107        found = true;
108      }
109    }
110  }
111  return found;
112 }
113
114 bool isFinished(int squares[SIDE_LENGTH][SIDE_LENGTH]){ //checks if there's a number in
       every square.
115   for(int i = 0; i < SIDE_LENGTH; i++){
116     for(int j = 0; j < SIDE_LENGTH; j++){
117       if(squares[i][j] == 0)
118         return false;
119     }
120   }
121   return true;
122 }
123
124 //get next square where there still is a possibility for numbers. Returns 0 if no squares are
       found.
```

```
125 void nextPossibility(int *possibleNumbers, int possibleNumbersArr[MAX_NUM], int *
       nextPosRow, int *nextPosCol){
126   *possibleNumbers = 0;
127   *nextPosRow = 0;
128   *nextPosCol = 0;
129   for(int i = 0; i < SIDE_LENGTH; i++){
130     for(int j = 0; j < SIDE_LENGTH; j++){
131       for(int num = 1; num < MAX_NUM+1; num++){
132         if(getPossibility(i,j,num)){
133           possibleNumbersArr[(*possibleNumbers)] = num;
134                     (*possibleNumbers)++;
135         }
136       }
137       if((*possibleNumbers) > 0){
138                 (*nextPosRow) = i;
139                 (*nextPosCol) = j;
140         return;
141       }
142     }
143   }
144 }
145
146 //checks if there's only one possible number to enter in sudoku.
147 bool checkForOnlyPossibility(int squares[SIDE_LENGTH][SIDE_LENGTH]){
148     bool morePossibilities = false;
149     bool found = false;
150     for(int row = 0; row < SIDE_LENGTH; row++){
151         for(int col = 0; col < SIDE_LENGTH; col++){
152             for(int num = 1; num < MAX_NUM+1; num++) {
153                 if (getPossibility(row, col, num)) { //number possible
154                     int rowStart = row / BLOCK_LENGTH;
155                     int colStart = col / BLOCK_LENGTH;
156                     rowStart = rowStart * BLOCK_LENGTH;
157                     colStart = colStart * BLOCK_LENGTH;
158                     for (int i = rowStart; i < rowStart + BLOCK_LENGTH && !
       morePossibilities; i++) {
159                         if (i != row) {
160                             for (int j = 0; j < SIDE_LENGTH; j++) {
161                                 morePossibilities = getPossibility(i, j, num);
162                             }
163                         }
164                     }
165                     for (int j = colStart; j < colStart + BLOCK_LENGTH && !
       morePossibilities; j++) {
166                         if (j != col) {
167                             for (int i = 0; i < SIDE_LENGTH; i++) {
168                                 morePossibilities = getPossibility(i, j, num);
169                             }
170                         }
171                     }
172                     if (!morePossibilities) {
173                         setSquare(squares, row, col, num);
174                         found = true;
175                     }
176                 }
177             }
178         }
```

```
179     }
180     return found;
181 }
182 
183 void copySquares(int from[SIDE_LENGTH][SIDE_LENGTH], int to[SIDE_LENGTH][SIDE_LENGTH
        ]){
184   for(int i = 0; i < SIDE_LENGTH; i++){
185     for(int j = 0; j < SIDE_LENGTH; j++){
186       to[i][j] = from[i][j];
187     }
188   }
189 }
190 
191 void copyPossibilities(int from[SIDE_LENGTH][SIDE_LENGTH][MAX_NUM], int to[
        SIDE_LENGTH][SIDE_LENGTH][MAX_NUM]){
192   for(int i = 0; i < SIDE_LENGTH; i++){
193       for(int j = 0; j < SIDE_LENGTH; j++){
194         for(int k = 0; k < MAX_NUM; k++){
195           to[i][j][k] = from[i][j][k];
196         }
197       }
198     }
199 }
200 
201 void sudoku_solve(int squares[]) { //interface according to assignment. Real work is done
        in sudoku_solve2.
202   int squares2d[SIDE_LENGTH][SIDE_LENGTH];
203   convertSudoku(squares, squares2d);
204   sudoku_solve2(squares2d);
205 }
206 
207 bool sudoku_solve2(int squares[SIDE_LENGTH][SIDE_LENGTH]) { //true if solution is found,
        else false
208   bool solved = false;
209   initSquaresPossible();
210   removeImpossibleNumbers(squares);
211   for(;;){
212     if(!checkForSingle(squares) && !checkForOnlyPossibility(squares)) { //get
        desperate and guess
213             int possibleNumbers;
214             int possibleNumbersArr[MAX_NUM];
215             int nextPosRow, nextPosCol;
216             nextPossibility(&possibleNumbers, possibleNumbersArr, &nextPosRow, &
        nextPosCol);
217             if (possibleNumbers == 0) //there are more numbers possible, but sudoku still
        isn't solved -- backtrack.
218                 return false;
219             int tSquares[SIDE_LENGTH][SIDE_LENGTH];
220             int tPossibilities[SIDE_LENGTH][SIDE_LENGTH][MAX_NUM];
221             copyPossibilities(squaresPossible, tPossibilities); //as squaresPossible is
        global, backup it.
222             for (int i = 0; i < possibleNumbers; i++) {
223                 copySquares(squares, tSquares); //backup squares
224                 setSquare(tSquares, nextPosRow, nextPosCol, possibleNumbersArr[i]);
225                 solved = sudoku_solve2(tSquares);
226                 copyPossibilities(tPossibilities, squaresPossible); //restore
        squaresPossible to global so we don't mess up.
```

```
227                 if(solved) return true;
228             }
229             return false;
230         }
231         if(isFinished(squares)) {
232             printSudoku(squares);
233             return true;
234         }
235   }
236 }
237
238 int main(int argc, char* argv[]) {
239   clock_t start, end;
240   double cpuTotal;
241
242   int squares[SIDE_LENGTH*SIDE_LENGTH];
243   if(argc != ARG_COUNT) {
244         printf("Usage: %s sudoku_string\n", argv[0]);
245         printf("Example: %s
      530070000600195000098000060800060003400803001700020006060000280000419005000080079\
      n", argv[0]);
246         return EXIT_FAILURE;
247     } else if(strlen(argv[1]) != SIDE_LENGTH*SIDE_LENGTH) {
248       printf("Sudoku string size must be %i!", SIDE_LENGTH*SIDE_LENGTH);
249         return EXIT_FAILURE;
250   }
251     inputNumbersIntoSudoku(squares, argv[1]);
252     start = clock();
253   sudoku_solve(squares);
254   end = clock();
255   cpuTotal = ((double) (end-start)) / CLOCKS_PER_SEC;
256   printf("Time taken in seconds: %f\n", cpuTotal);
257   return 0;
258 }
```

3.2.3 Testen

```
khp@KLS ~/g/f/w/s/o/a/s/s/cmake-build-debug (master)> ./sudoku
Usage: ./sudoku sudoku_string
Example: ./sudoku 530070000600195000098000060800060003400803001700020006060000280000419005000080079
```

Abbildung 3.7: Ausgabe wenn nur Programmname eingegeben wird.

```
./sudoku 356002010701056300000000000064050729109500047081720906300000000000436010503070094 6
|3|5|6|8|9|2|7|1|4|
|7|2|1|4|5|6|3|8|9|
|4|8|9|1|7|3|6|5|2|
|6|4|3|5|8|7|2|9|1|
|2|9|5|6|3|1|4|7|8|
|8|1|7|2|4|9|5|6|3|
|5|6|2|9|1|4|8|3|7|
|9|7|4|3|6|8|1|2|5|
|1|3|8|7|2|5|9|4|6|
Time taken in seconds: 0.000240
```

Abbildung 3.8: Laufzeit bei einem leichten Sudoku.

```
./sudoku 100000005008945010075003400090056100400090007006370040007100820060728500800000006
|1|4|9|2|8|7|3|6|5|
|6|3|8|9|4|5|7|1|2|
|2|7|5|6|1|3|4|9|8|
|7|9|2|4|5|6|1|8|3|
|4|1|3|8|9|2|6|5|7|
|5|8|6|3|7|1|2|4|9|
|3|5|7|1|6|9|8|2|4|
|9|6|4|7|2|8|5|3|1|
|8|2|1|5|3|4|9|7|6|
Time taken in seconds: 0.001211
```

Abbildung 3.9: Laufzeit bei einem mittlerem Sudoku.

```
./sudoku 300020058000570360000003001600300085001806400480002006200100000014065000850030007
|3|9|6|4|2|1|7|5|8|
|1|4|2|5|7|8|3|6|9|
|5|7|8|6|9|3|2|4|1|
|6|2|7|3|4|9|1|8|5|
|9|3|1|8|5|6|4|7|2|
|4|8|5|7|1|2|9|3|6|
|2|6|3|1|8|7|5|9|4|
|7|1|4|9|6|5|8|2|3|
|8|5|9|2|3|4|6|1|7|
Time taken in seconds: 0.000857
```

Abbildung 3.10: Laufzeit bei einem schwerem Sudoku.

```
./sudoku 600000000040038590083100600059700060400010009070006340007001250068390010000000003
|6|1|5|2|7|9|8|3|4|
|7|4|2|6|3|8|5|9|1|
|9|8|3|1|5|4|6|7|2|
|2|5|9|7|4|3|1|6|8|
|4|3|6|8|1|5|7|2|9|
|8|7|1|9|2|6|3|4|5|
|3|9|7|4|8|1|2|5|6|
|5|6|8|3|9|2|4|1|7|
|1|2|4|5|6|7|9|8|3|
Time taken in seconds: 0.001526
```

Abbildung 3.11: Laufzeit bei einem schwerem+ Sudoku.

Abbildung 3.12: Laufzeit bei einem ultraschwerem Sudoku.

Abbildung 3.13: Laufzeit bei einem unlösbaren Sudoku.

Abbildung 3.14: Ausgabe bei fehlerhafter Eingabe.