Power of Recursion and Backtracking Advanced

Michael Inden

https://github.com/Michaeli71/PowerOfRecursionAndBacktracking

Agenda

Workshop Contents

- **Recap Rekursion**
  - Rekursion im Überblick
  - Grafische Gebilde
  - Binärsuche
  - MergeSort
  - QuickSort

- **PART 4: Fallstricke und Abhilfen**
  - Fallstricke
  - Abhilfe: Memoization

https://github.com/Michaeli71/PowerOfRecursionAndBacktracking

Workshop Contents

- **PART 5: Backtracking**
  - Weg aus Labyrinth
  - n-Damenproblem
  - Sudoku Solver
  - Knights Tour

https://github.com/Michaeli71/PowerOfRecursionAndBacktracking

PART 1:
Recap Rekursion

Rekursion im Überblick

Rekursion

- Rekursion ist eine Vorgehensweise, bei der eine Methode sich selbst aufruft.
- Klingt merkwürdig, aber ist manchmal sehr elegant
- komplizierte Probleme in einfachere Teilprobleme zerlegen, die leichter zu lösen sind (manchmal ist das tatsächlich bereits nur ein um eins reduzierter Wert eines Parameters)
- Beispiele aus der Mathematik: Fakultät und Fibonacci-Zahlen

$$n! = \begin{cases} 1, & n = 0,\ n = 1 \\ n \cdot (n-1)!, & \forall n > 1 \end{cases}$$

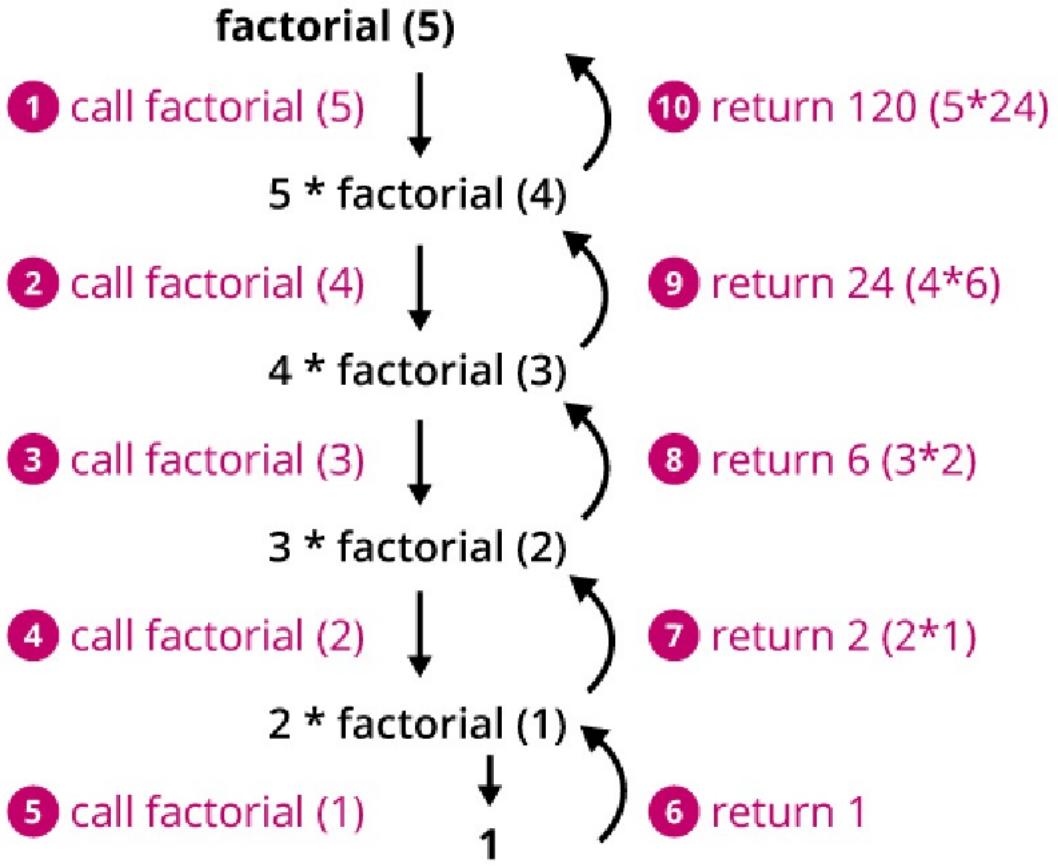
$$fib(n) = \begin{cases} 1, & n = 1 \\ 1, & n = 2 \\ fib(n-1) + fib(n-2), & \forall n > 2 \end{cases}$$

Rekursion

- **Beispiele aus der Mathematik**

$$n! = \begin{cases} 1, & n = 0,\ n = 1 \\ n \cdot (n-1)!, & \forall n > 1 \end{cases}$$

```
static int factorial(final int n)
{
    if (n == 1)
        return 1;

    return n * facatorial(n-1);
}
```



Rekursion

- **Beispiele aus der Mathematik**

$$fib(n) = \begin{cases} 1, & n = 1 \\ 1, & n = 2 \\ fib(n-1) + fib(n-2), & \forall n > 2 \end{cases}$$

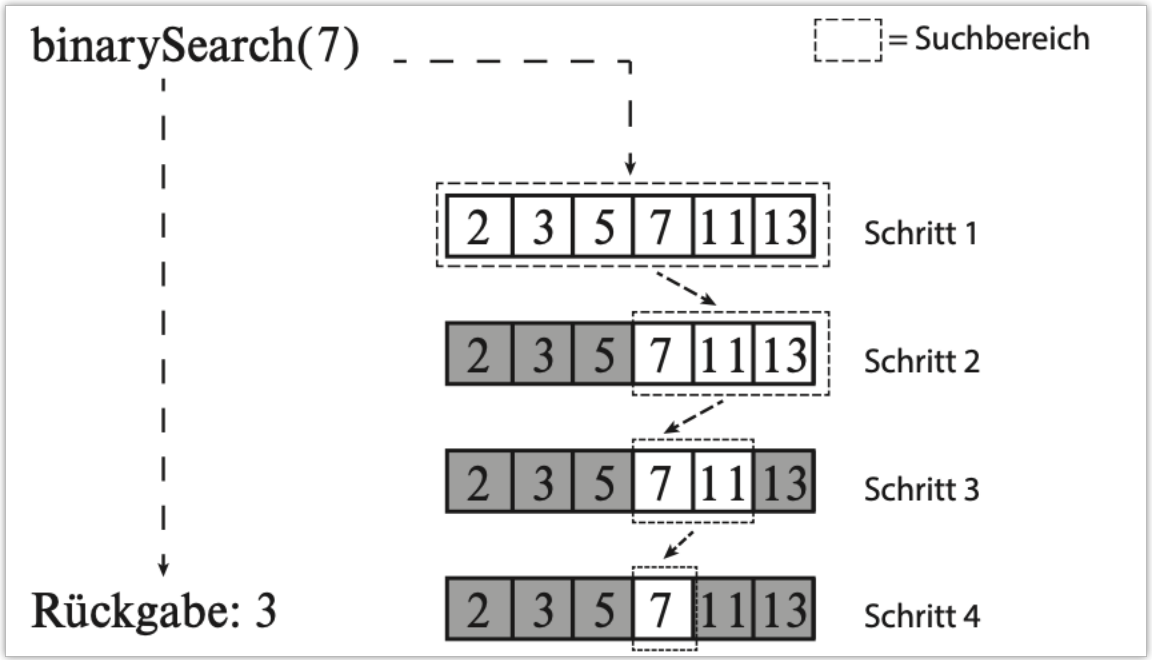
```
static long fib(final int n)
{
    if (n <= 0)
        throw new IllegalArgumentException("n must be positive and >= 1");

    // rekursiver Abbruch
    if (n == 1 || n == 2)
        return 1;

    // rekursiver Abstieg
    return fib(n - 1) + fib(n - 2);
}
```

- **Beispiele aus der Algorithmik: String umdrehen**

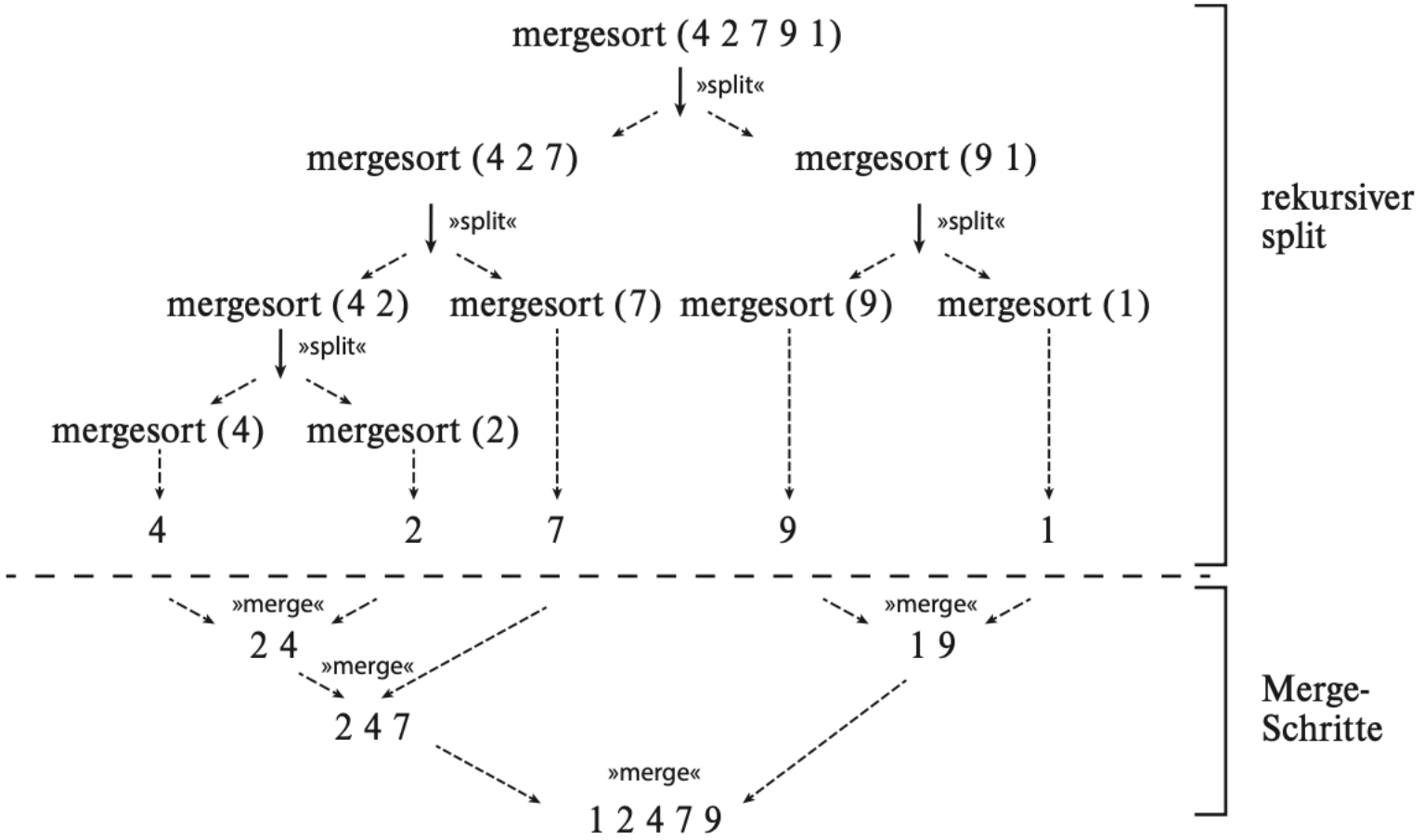
```
static String reverseString(final String input)
{
    if (input.length() <= 1)
        return input;

    final char firstChar = input.charAt(0);
    final String remaining = input.substring(1);

    return reverseString(remaining) + firstChar;
}


static String reverseStringShort(final String input)
{
    return input.length() <= 1 ?
           input : reverseStringShort(input.substring(1)) + input.charAt(0);
}
```

Rekursion

- Binärsuche arbeitet auf sortierten Datenbeständen
- Effiziente Suche in logarithmischer Zeit
- Algorithmus: die jeweils zu verarbeitenden Bereiche werden halbiert und danach wird im passenden Teilstück weitergesucht.

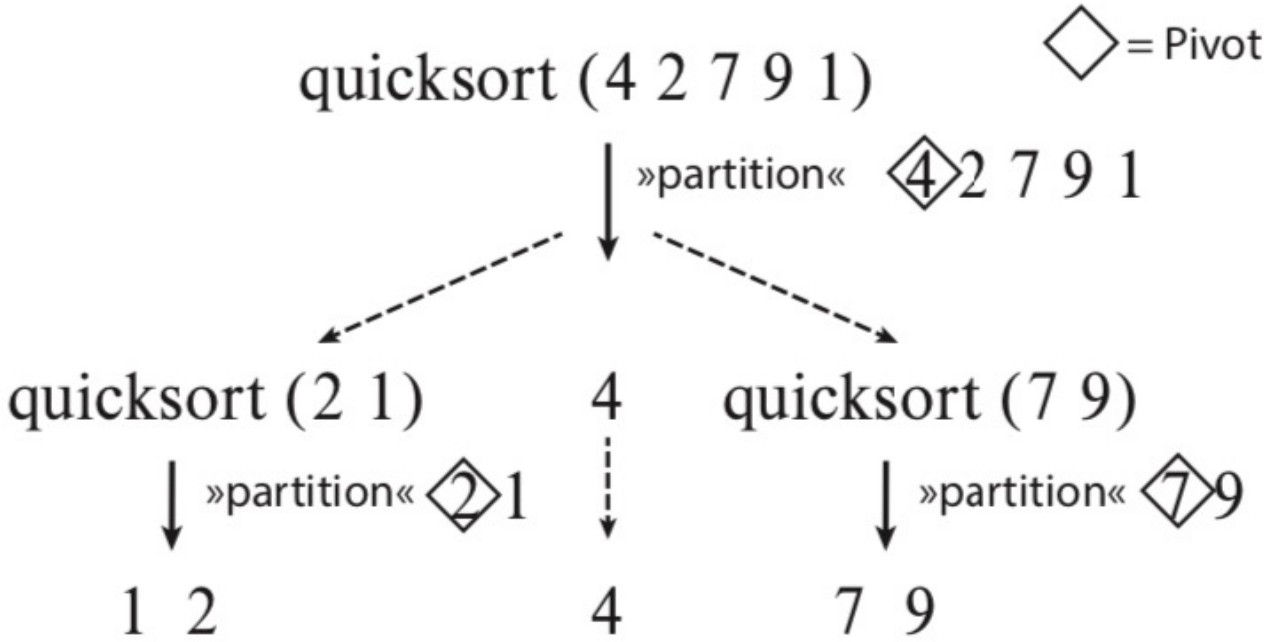


- **Merge Sort**


mergesort (4 2 7 9 1)
»split«
mergesort (4 2 7)
mergesort (9 1)
»split«
»split«
mergesort (4 2)
mergesort (7)
mergesort (9)
mergesort (1)
»split«
mergesort (4)
mergesort (2)
4
2
7
9
1
»merge«
2 4
»merge«
1 9
»merge«
2 4 7
»merge«
1 2 4 7 9
rekursiver split
Merge-Schritte

Lösungsstrategie / Algorithmus

- **Quick Sort** basiert auf einem **Divide-and-Conquer**-Ansatz
- **Zerteilt** den zu sortierende Datenbestand **in** immer **kleinere Teile**.
- Spezielles Element (***Pivot***) **legt** die **Unterteilung fest**.
- Alle **Elemente** des Teilbereichs mit Wert **kleiner oder gleich** werden **links** bzw. die **größeren** werden **rechts** vom Pivot **anordnet**.
- **Rekursive Wiederholung** bis die Teilbereiche nur noch einelementig sind.



Flood Fill
(sogar mit Muster)

Aufgabenstellung

Schreibe eine Implementierung so, dass eine Fläche mit Muster gefüllt werden kann!

```
## o                          #
#  ###################      ##
  #         ##              ###
  #         ##               ##
#  ################          #
##                           ##
###                         ###
##                           ##
#                            #
```

```
##..|..|..|..|..|..|..|..#
#*-###################--*-##
.|#          ##|..|..|..|###
-*#          ##*--*--*--*-##
#|.#################|..|..#
##--*--*--*--*--*--*--*-##
###.|..|..|..|..|..|..|###
##--*--*--*--*--*--*--*-##
#|..|..|..|..|..|..|..|..#
```

```java
void floodFill(char[][] values, int x,
               int y, char[][] pattern)
{
    if (x < 0 || y < 0 ||
        y >= values.length ||
        x >= values[y].length)
        return;

    if (values[y][x] == ' ')
    {
        values[y][x] = findFillChar(x, y, pattern);


        floodFill(values, x, y - 1, pattern);
        floodFill(values, x + 1, y, pattern);
        floodFill(values, x, y + 1, pattern);
        floodFill(values, x - 1, y, pattern);
    }
}
```

```python
def flood_fill(values2dim, x, y, pattern):
    max_y, max_x = get_dimension(values2dim)

    if x < 0 or y < 0 or \
       x >= max_x or y >= max_y:
        return

    if values2dim[y][x] == ' ':
        values2dim[y][x] = find_fill_char(y, x,
                                          pattern)

        flood_fill(values2dim, x, y - 1, pattern)
        flood_fill(values2dim, x + 1, y, pattern)
        flood_fill(values2dim, x, y + 1, pattern)
        flood_fill(values2dim, x - 1, y, pattern)
```

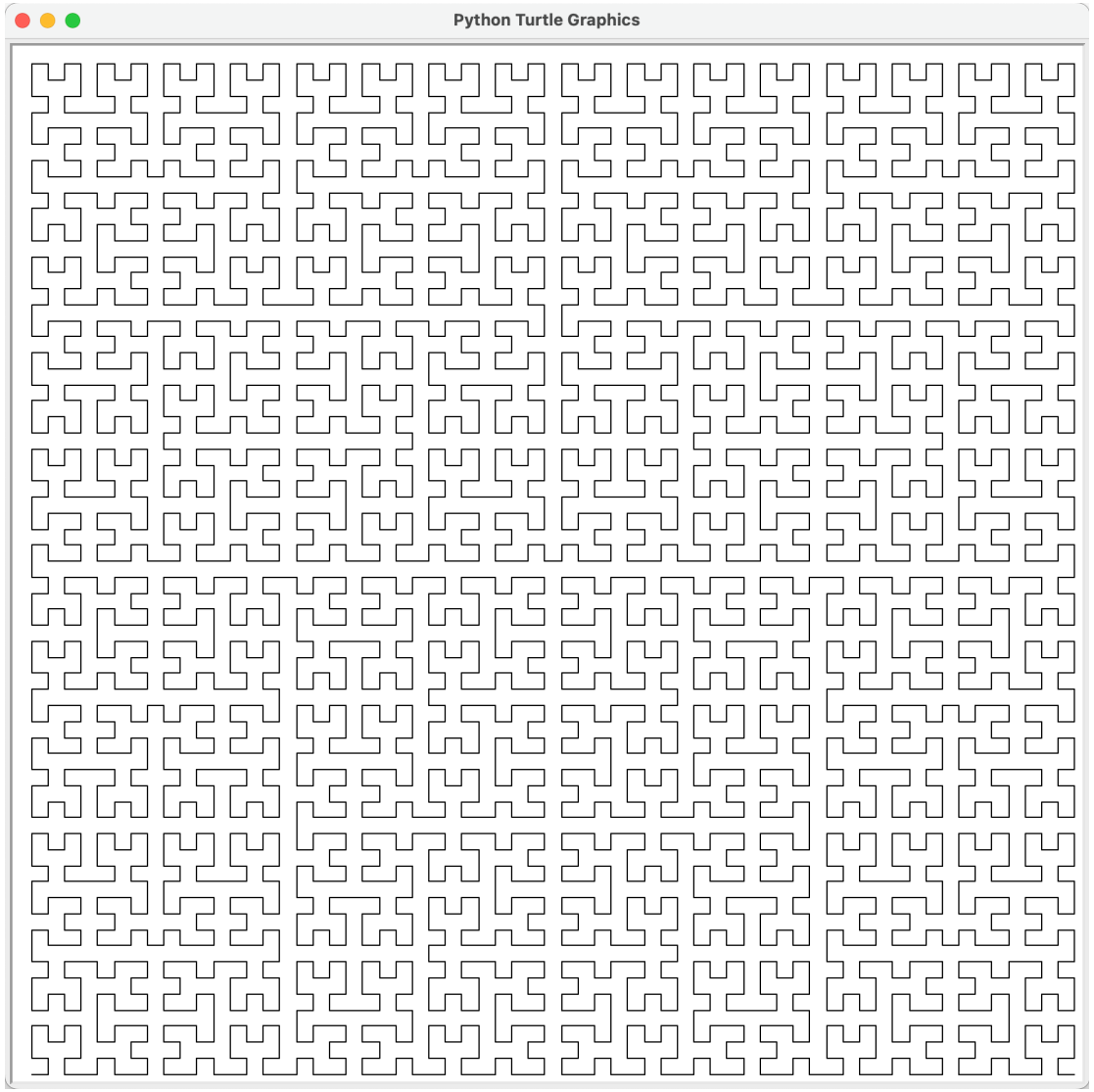
```java
char findFillChar(int x, int y,
                  char[][] pattern)
{
    final int maxY = pattern.length;
    final int maxX = pattern[0].length;

    return pattern[y % maxY][x % maxX];
}
```

```python
def find_fill_char(y, x, pattern):
    max_y, max_x = get_dimension(pattern)

    return pattern[y % max_y][x % max_x]
```

Rekursion

- **Grafische Figuren**

```
def hilbert_curve(n, turtle, angle=90):
    if n <= 0:
        return

    turtle.left(angle)
    hilbert_curve(n - 1, turtle, -angle)
    turtle.forward(1)
    turtle.right(angle)
    hilbert_curve(n - 1, turtle, angle)
    turtle.forward(1)
    hilbert_curve(n - 1, turtle, angle)
    turtle.right(angle)
    turtle.forward(1)
    hilbert_curve(n - 1, turtle, -angle)
    turtle.left(angle)
```



PART 4:
Fallstricke und Abhilfen

Fallstricke

Rekursion

- **Endlosaufrufe**

```
// Achtung: Zur Demonstration bewusst falsch
static void infiniteRecursion(final String value)
{
    infiniteRecursion(value);
}

static int factorialNoAbortion(final int number)
{
    return number * factorialNoAbortion(number - 1);
}
```

Rekursion

- **Endlosaufrufe**

```
static int calcLengthParameterValues(final String value, int count)
{
    if (value.length() == 0)
        return count;

    System.out.println("Count: " + count);
    final String remaining = value.substring(1);

    return calcLengthParameterValues(remaining, count++);
}
```

Rekursion

- **Berechnungsdauer**

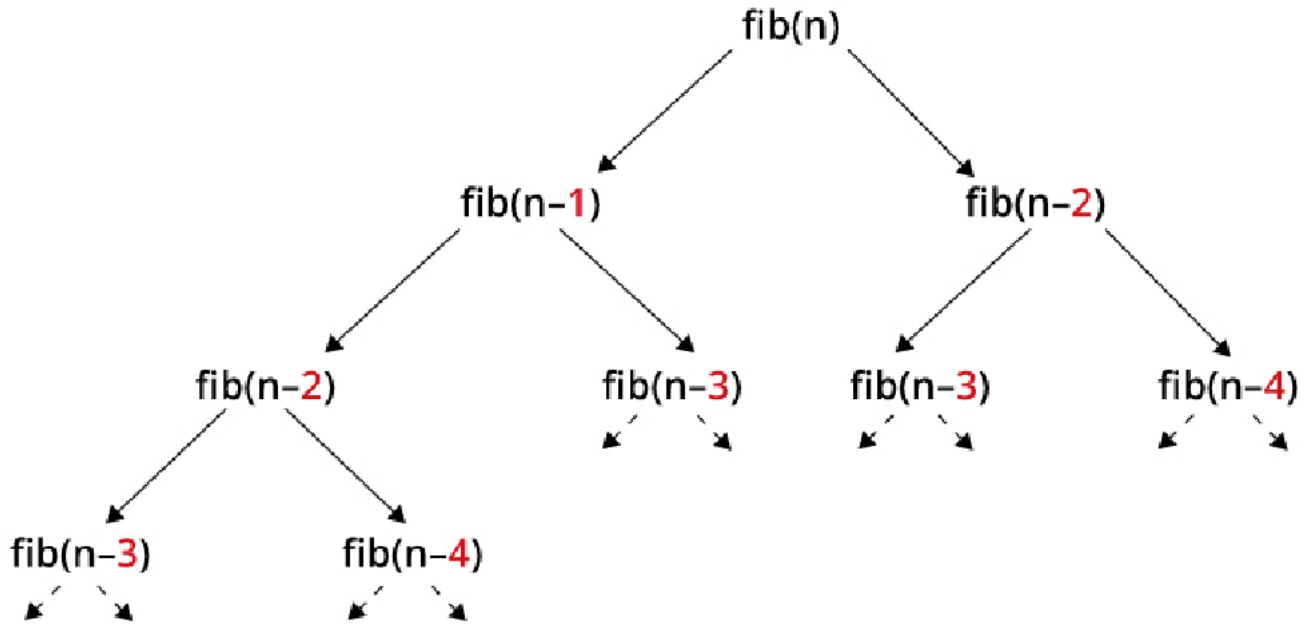
```
static long fibRec(final int n)
{
    if (n <= 0)
        throw new IllegalArgumentException("must be positive and >= 1");

    // rekursiver Abbruch
    if (n == 1 || n == 2)
        return 1;

    // rekursiver Abstieg
    return fibRec(n - 1) + fibRec(n - 2);
}


public static void main(String[] args)
{
    System.out.println("fibRec(42) = " + fibRec(42)); // sofort
    System.out.println("fibRec(45) = " + fibRec(45)); // ca. 3 s
    System.out.println("fibRec(50) = " + fibRec(50)); // ca. 23 s
    System.out.println("fibRec(70) = " + fibRec(70)); // take a break ...
}
```

DEMO

`Fibonacci.java`

Woran könnte das liegen?

- **Ursache für die hohe Berechnungsdauer**



- **Selbst bei diesem exemplarischen Aufruf erkennt man, dass diverse Aufrufe mehrmals erfolgen, etwa für *fib(n − 4)* und *fib(n − 2)*, aber insbesondere dreimal für *fib(n − 3)*. Das führt sehr schnell zu aufwendigen und langwierigen Berechnungen. Später Optimierung durch Memoization.**

Rekursion

- **Beispiel: Berechne die möglichen Kombinationen, eine beliebige Summe aus 2 und 1 CHF / Euro zusammenzusetzen**

- **Beispiel 7 Euro u.a.:**

Rekursion

- **Beispiel: Berechne die möglichen Kombinationen, eine beliebige Summe aus 5, 2 und 1 CHF zusammenzusetzen**

- **Beispiel 7 CHF u.a.:**

- **Rekursive Berechnung**

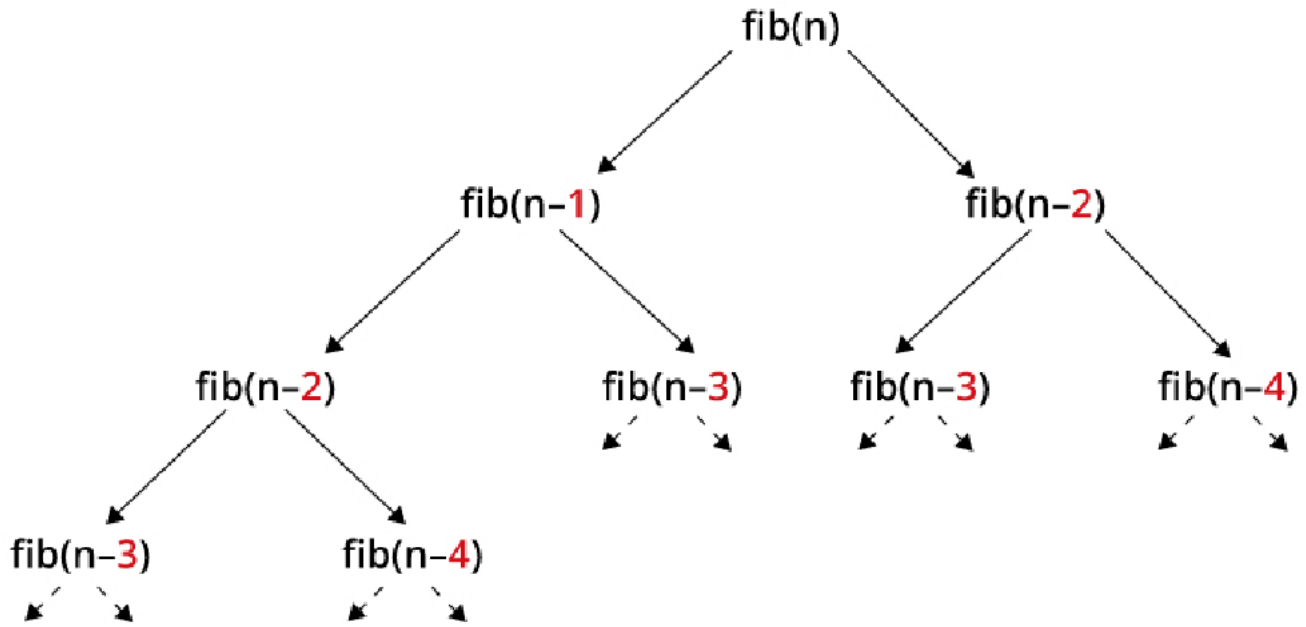
```
def coins_5_1(n):
    if n < 0:
        return 0

    if n == 0 or n == 4:
        return 1

    return coins_5_1(n - 5) + coins_5_1(n - 1)


# 5, 2, 1
def coins_5_2_1(n):
    if n < 0:
        return 0

    if n == 0 or n == 1 or n == 4:
        return 1

    return coins_5_2_1(n - 5) + coins_5_2_1(n - 2) + coins_5_2_1(n - 1)
```

DEMO

`coins.py`

Abhilfe: Memoization

- **Recap Ursache für die hohe Berechnungsdauer**



- **Selbst bei diesem exemplarischen Aufruf erkennt man, dass diverse Aufrufe mehrmals erfolgen, etwa für *fib(n − 4)* und *fib(n − 2)*, aber insbesondere dreimal für *fib(n − 3)*. Das führt sehr schnell zu aufwendigen und langwierigen Berechnungen. Später Optimierung durch Memoization.**

Habt ihr eine Idee, wie man vorgehen könnte?

Rekursion

- **Man speichert Daten zwischen und folgt den Ideen das Caching!**

- **Dazu muss man Zustand übergeben => dazu hatten wir schon Parameter kennengelernt und den Trick der Hilfsmethode**

```
static long fibonacciOptimized(final int n)
{
    return fibonacciMemo(n, new HashMap<>());
}
```

Rekursion

- **Dazu muss man Zustand übergeben => dazu hatten wir schon Parameter kennengelernt und den Trick der Hilfsmethode**

```
static long fibonacciMemo(final int n, final Map<Integer, Long> lookupMap)
{
    if (n <= 0)
        throw new IllegalArgumentException("must be > 0");

    // MEMOIZATION: prüfe, ob vorberechnetes Ergebnis
    if (lookupMap.containsKey(n))
        return lookupMap.get(n);

    // normaler Algorithmus mit Hilfsvariable für Resultat
    long result = 0;
    if (n == 1 || n == 2)
        result = 1;
    else
        result = fibonacciMemo(n - 1, lookupMap) + fibonacciMemo(n - 2, lookupMap);

    // MEMOIZATION: speichere berechnetes Ergebnis
    lookupMap.put(n, result);

    return result;
}
```

Rekursion

- **Dazu muss man Zustand übergeben => dazu hatten wir schon Parameter kennengelernt und den Trick der Hilfsmethode**

```
static long fibonacciOptimizedNicer(final int n)
{
    return fibonacciMemoOpt(n, new HashMap<>(Map.of(1, 1L, 2, 1L)));
}

static long fibonacciMemoOpt(final int n, final Map<Integer, Long> lookupMap)
{
    if (n <= 0)
        throw new IllegalArgumentException("must be > 0");

    // MEMOIZATION: prüfe, ob vorberechnetes Ergebnis
    if (lookupMap.containsKey(n))
        return lookupMap.get(n);

    // normaler Algorithmus mit Hilfsvariable für Resultat
    long result = fibonacciMemoOpt(n - 1, lookupMap) + fibonacciMemoOpt(n - 2, lookupMap);

    // MEMOIZATION: speichere berechnetes Ergebnis
    lookupMap.put(n, result);

    return result;
}
```

DEMO

MemoizationExamples.java

Rekursion

```
def coins_5_2_1(n):
    if n < 0:
        return 0

    if n == 0 or n == 1 or n == 4:
        return 1

    return coins_5_2_1(n - 5) + coins_5_2_1(n - 2) + coins_5_2_1(n - 1)
```

- **Mit Memoization**

```
def coins_5_2_1(n):
    return coins_5_2_1_memo(n, {0: 1, 1: 1, 4: 1})


def coins_5_2_1_memo(n, cache):
    if n < 0:
        return 0

    if n in cache:
        return cache[n]

    cache[n] = coins_5_2_1_memo(n - 5, cache) + coins_5_2_1_memo(n - 2, cache) + coins_5_2_1_memo(n - 1, cache)
    return cache[n]
```

DEMO

`coins_memo.py`

Exercises Part 4

https://github.com/Michaeli71/PowerOfRecursionAndBacktracking

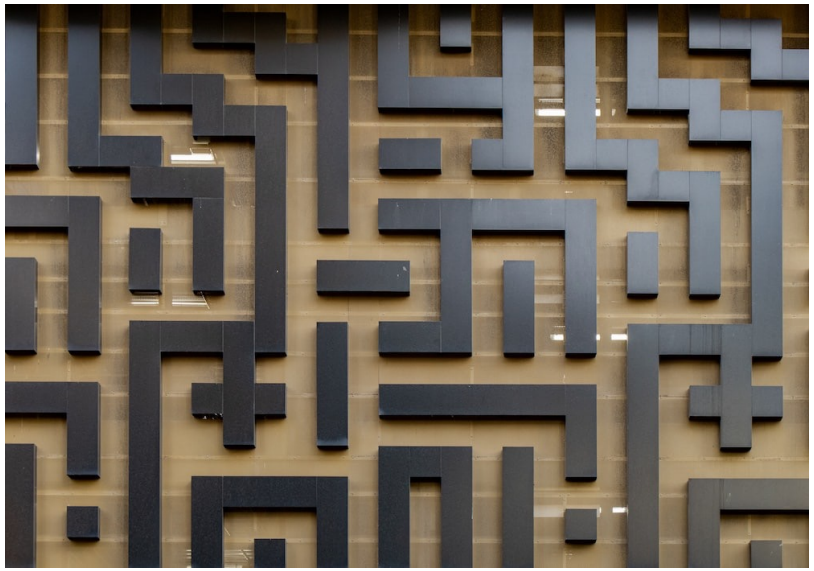
PART 5:
Backtracking

Backtracking

- Backtracking ist eine Problemlösungsstrategie, die auf Versuch und Irrtum beruht und alle möglichen Lösungen untersucht.
- Wenn ein Fehler entdeckt wird, werden die vorherigen Schritte zurückgesetzt, daher der Name Backtracking.
- Das Ziel ist es, schrittweise durch geschicktes Ausprobieren zu einer Lösung zu gelangen.
- Wenn dabei ein Fehler auftritt, versucht man einen anderen Weg zur Lösung.

- Dabei werden potentiell alle möglichen (und damit vielleicht auch sehr viele) Wege beschritten.
- Das hat allerdings auch einen Nachteil, nämlich eine recht lange Laufzeit, bis das Problem gelöst ist.

Weg aus Labyrinth



Aufgabenstellung

Schreiben Sie eine Methode / Funktion `findWayOut()` / `find_way_out()`, die von einer beliebigen Position aus einen Weg zu allen Ausgängen ermittelt und die jeden gefundenenAusgang mit FOUND EXIT protokolliert. Dabei kann man sich nur in die vier Himmelsrichtungen bewegen, nicht aber diagonal.

```
##################################
# #          #     #      #   #   X#X#
#  #####  ####  ##    ##   #   # ###   #
#  ##   #     #   ## ##   #   #       # #
#     #   ###   # ## ##    #    ### # #
# #    ####      ##   ##      ###   # #
####    #      ####       #   # ####   #
######    #########     ##    # ###   #
##      #   X X####X  #   #   # ###   ##
##################################
```

```
FOUND EXIT: x: 30, y: 1
FOUND EXIT: x: 17, y: 8
FOUND EXIT: x: 10, y: 8
##################################
#.#          #....#.....#   #...X#X#
#..#####  ####.##...##..#   #.###   #
# .##   #     #..## ## .#   #..    # #
# ...#   ###..#.## ## ..#...### # #
# # ..####.....##   ##.... ###   # #
#### ..#...   ####  ....#   # ####   #
######...#########...##    # ###   #
##      #..X X####X.#   #   # ###   ##
##################################
```

Lösungsstrategie / Algorithmus

- Der Algorithmus zum Auffinden eines Wegs aus einem Labyrinth schaut ausgehend von der aktuellen Position, ob es einen Weg in die vier Himmelsrichtungen gibt.
- Dazu werden schon besuchte benachbarte Felder mit dem Zeichen '.' markiert, so wie man es in der Realität etwa mit Steinchen machen würde.
- Das Ausprobieren wird so lange fortgesetzt, bis man auf ein 'X' als Lösung, eine Mauer in Form eines '#' oder ein bereits besuchtes Feld (markiert durch '.') trifft.
- Existiert ausgehend von einer Position keine einzige mögliche Richtung, so nutzt man Backtracking, nimmt den zuletzt eingeschlagenen Weg zurück und probiert von dort die verbliebenen anderen Wege.

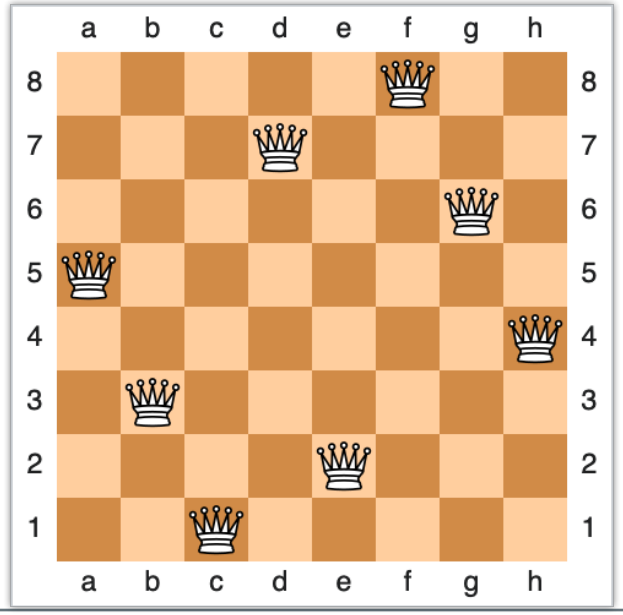
```
static boolean findWayOut(final char[][] values, final int x, final int y)
{
    if (x < 0 || y < 0 || x > values[0].length || y >= values.length)
        return false;

    if (values[y][x] == 'X')
    {
        System.out.println(String.format("FOUND EXIT: x: %d, y: %d", x, y));
        return true;
    }

    if (values[y][x] == '#')
        return false;

    // already visited
    if (values[y][x] == '.')
        return false;

    if (values[y][x] == ' ')
    {
        // mark as visitied
        values[y][x] = '.';

        …
                                                    …
                                                    // try up. left, down, right
                                                    final boolean up = findWayOut(values, x, y - 1);
                                                    final boolean left = findWayOut(values, x + 1, y);
                                                    final boolean down = findWayOut(values, x, y + 1);
                                                    final boolean right = findWayOut(values, x - 1, y);

                                                    final boolean foundAWay = up || left || down || right;
                                                    if (!foundAWay)
                                                    {
                                                        // Falscher Weg, somit Feldmarkierung löschen}
                                                        values[y][x] = ' ';
                                                    }
                                                    return foundAWay;
                                                }
                                                throw new IllegalStateException("wrong char in labyrinth");
}
```

```java
static boolean findWayOutV2(final char[][] board, final int x, final int y)
{
    if (board[y][x] == '#')
        return false;

    boolean found = board[y][x] == 'X';
    if (found)
        System.out.printf("FOUND EXIT: x: %d, y: %d%n", x, y);

    board[y][x] = '#';
    found = found | findWayOutV2(board, x + 1, y);
    found = found | findWayOutV2(board, x - 1, y);
    found = found | findWayOutV2(board, x, y + 1);
    found = found | findWayOutV2(board, x, y - 1);
    return found;
}
```

```
FOUND EXIT: x: 10, y: 8
FOUND EXIT: x: 12, y: 8
FOUND EXIT: x: 30, y: 1
FOUND EXIT: x: 17, y: 8
################################
################################X#
################################  #
#####  ######################### #
######  ######################### #
################################ #
########################## ####  #
########################## ###  #
########################## ###  ##
################################
```

DEMO

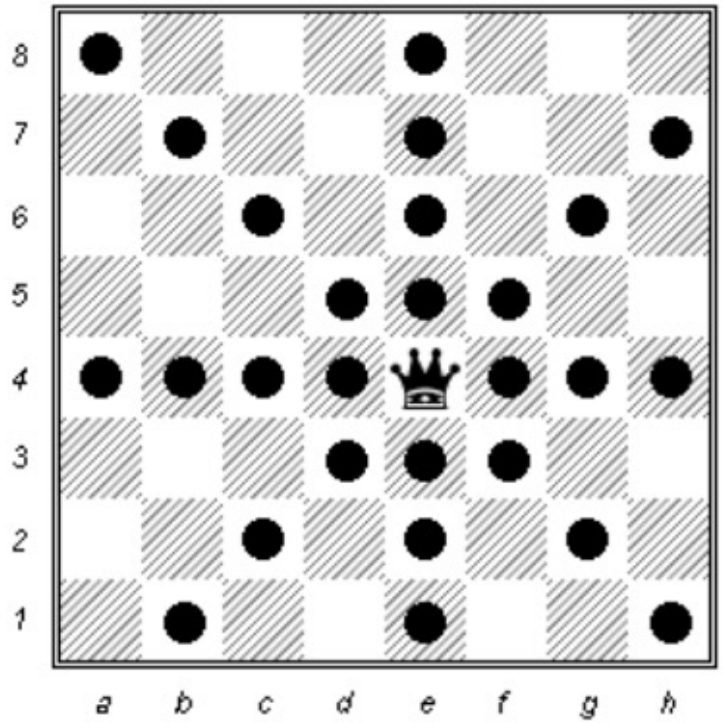
n-Damenproblem



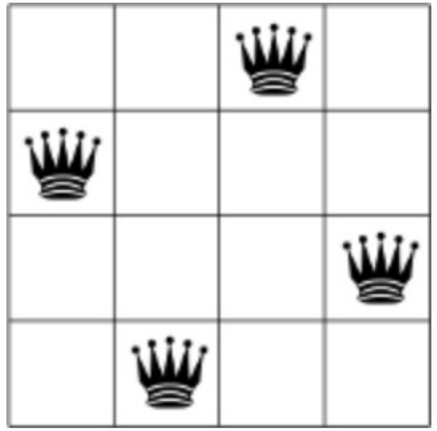
https://en.wikipedia.org/wiki/Eight_queens_puzzle

Lösungsstrategie / Algorithmus

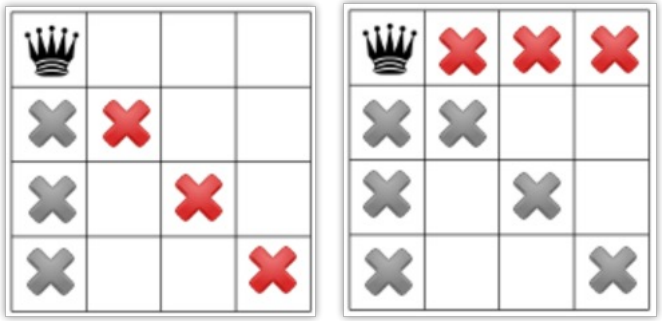
Es sollen auf einem NxN-Spielfeld (Schachbrett)
auf jeder Zeile jeweils eine Königin platziert werden,
sodass sich diese nicht schlagen können, also diagonal,
horizontal und vertikal keine Überschneidung besteht.



Beispiel für 4x4



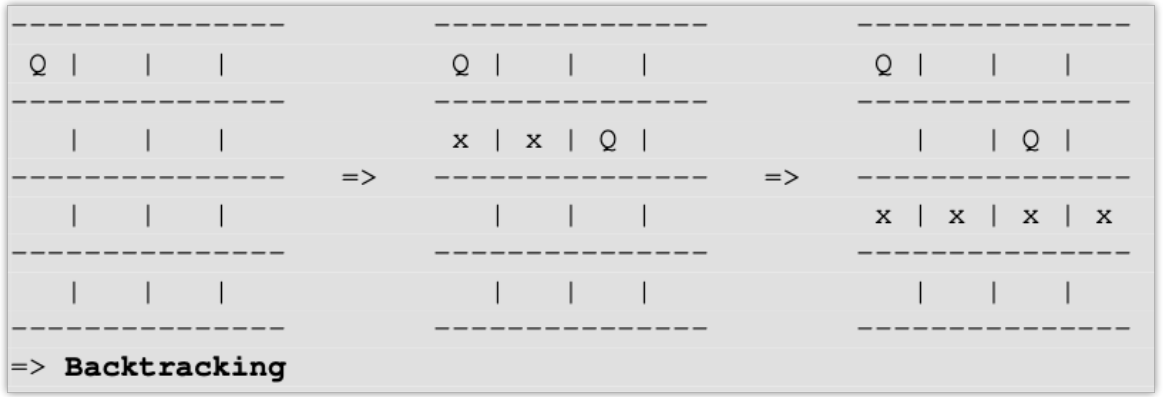
Ausschluss

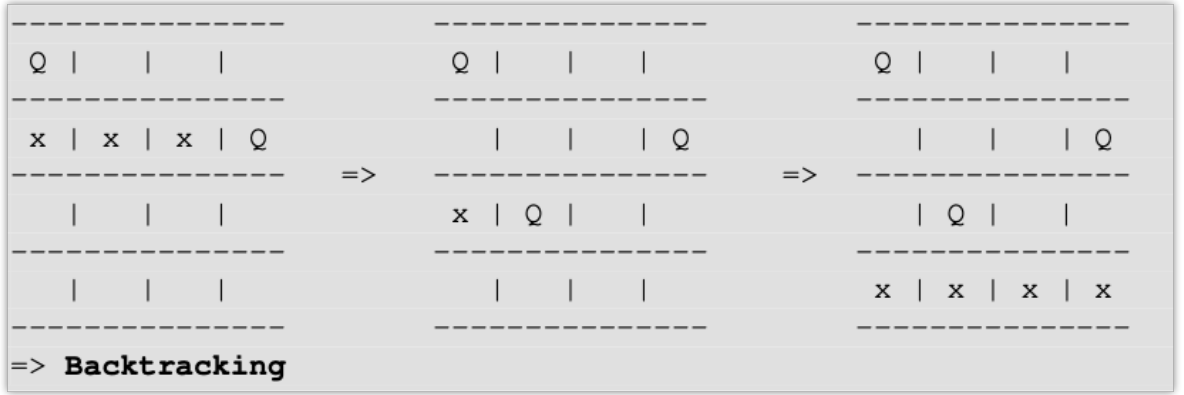


https://cpothmann.de/wp-content/uploads/2021/07/AB-Stack-Backtracking.pdf

Lösungsstrategie / Algorithmus

1. Start with an empty board of size n x n.
2. Place the first queen in the top-left corner.
3. Move to the next row and try to place a queen in each column until a valid position is found or all columns have been tried.
4. If a valid position is found, move to the next row and repeat step 3.
5. If all columns have been tried in the current row without finding a valid position, backtrack to the previous row and move the queen to the next column in that row. Repeat step 3.
6. If all rows have been tried without finding a solution, backtrack to the previous row and move the queen to the next column in that row. Repeat step 3.
7. If a solution is found, exit (alternatively collect more solutions)





https://www.javatpoint.com/n-queens-problems

Lösungsstrategie / Algorithmus

Weitere Informationen sind zu finden unter:

- https://developers.google.com/optimization/cp/queens?hl=de
- https://www.mathematik.ch/spiele/N_Damenproblem/ (mit Animation)
- https://www.geeksforgeeks.org/n-queen-problem-backtracking-3/
- https://de.wikipedia.org/wiki/Damenproblem
- https://www.programmieraufgaben.ch/aufgabe/damenproblem/k998inzt
- https://weiterbildung-informatik.wollw.de/sek2-chapter19/part2/
- https://www.hochbegabte-begleiten.de/images/puzzle/Eight.pdf

Knights Tour

1	48	31	50	33	16	63	18
30	51	46	3	62	19	14	35
47	2	49	32	15	34	17	64
52	29	4	45	20	61	36	13
5	44	25	56	9	40	21	60
28	53	8	41	24	57	12	37
43	6	55	26	39	10	59	22
54	27	42	7	58	23	38	11

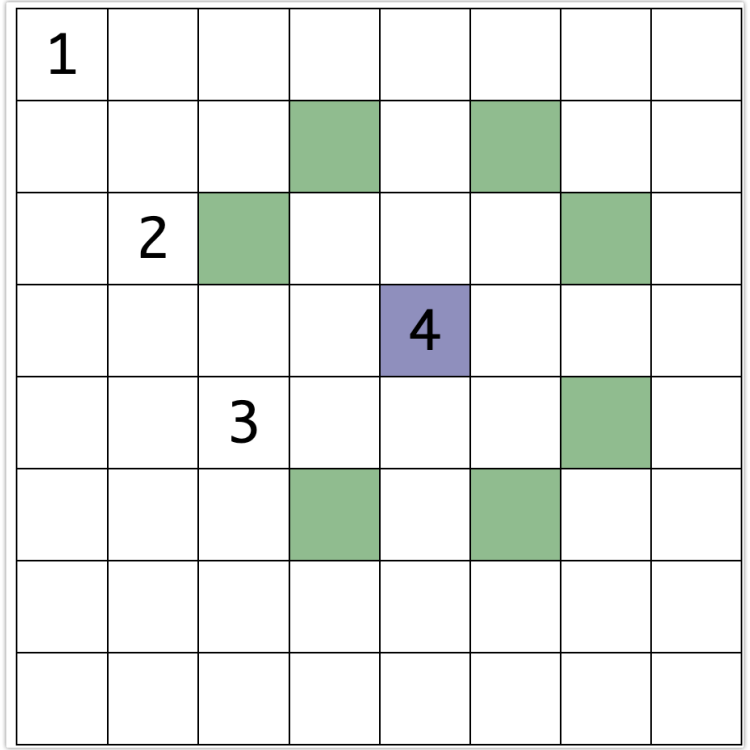
https://en.wikipedia.org/wiki/Knight%27s_tour

Lösungsstrategie / Algorithmus

Ein Springer besitzt im Schachspiel spezifische Spielschritte. Er kann sich von seinem Ausgangsfeld 2 Felder in eine beliebige Richtung nach vorne und 1 Feld nach links oder rechts bewegen.

Der Springer hat er variierende Bewegungsmöglichkeiten. Befindet er sich in der Mitte des Spielfeldes, so hat er im optimalen Fall 8 Zielfelder, die er anspringen kann.

Mit diesen Bewegungsvarianten ist er in der Lage, jedes Feld des Schachbretts zu erreichen.



http://alt.web49.s218.goserver.host/info-wsf.de/index.php/Springerproblem.html

Lösungsstrategie / Algorithmus

Ein Lösungsvorschlag für das Springerproblem, ist diese:

1	16	13	8	21
12	7	20	3	14
17	2	15	22	9
6	11	24	19	4
25	18	5	10	23

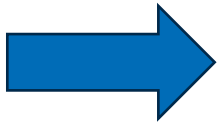
Lösungsfindung:

1. Auflisten aller möglichen Spielzüge von der jetzigen Position
2. Wählen der ersten/nächsten Zielkoordinaten
3. Wenn die Position unbesucht ist, setzte Springer dorthin - Weiter bei 1
4. Wenn die Position bereits besucht oder außerhalb des Spielbrettes - Weiter bei 2 => hier Backtracking integrieren
5. Wiederholen, bis alle Felder besucht worden sind oder es keine Lösung gibt

Sudoku Solver

Lösungsstrategie / Algorithmus

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



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

Lösungsstrategie / Algorithmus

Schreiben Sie eine Methode `solveSudoku(int[][])`, die für ein teilweise initialisiertes Spielfeld, das als Parameter übergeben wird, eine eventuell gültige Lösung ermittelt.

Algorithmus:

- Um Sudoku zu lösen, verwenden wir Backtracking. Wie bei anderen Backtracking-Problemen kann Sudoku durch schrittweises Ausprobieren gelöst werden.
- In diesem Fall bedeutet das, dass man verschiedene Zahlen für jedes der leeren Quadrate ausprobiert. Nach den Sudoku-Regeln darf die aktuelle Ziffer nicht bereits horizontal, vertikal oder in einem 3 × 3-Block vorkommen.
- Wenn wir eine gültige Wertzuweisung finden, können wir rekursiv an der nächsten Stelle weitermachen, um zu testen, ob wir eine Lösung finden. Wenn keine gefunden wird, versuchen wir das Verfahren mit der nächsten Ziffer.
- Wenn jedoch keine der Ziffern von 1 bis 9 zu einer Lösung führt, müssen wir zurückgehen, um andere mögliche Wege zur Lösung zu untersuchen.

Lösungsstrategie / Algorithmus

1. Find the next empty field. To do this, skip all fields that are already filled.This can also change lines.
2. If no more empty field exists until the last row, we have found the solution.
3. Otherwise we try out the digits from 1 to 9:
    a. Is there a conflict? Then we have to try the next digit.
    b. The digit is a possible candidate. We call our method recursively for the following position (next column or even next row).
    c. If the recursion returns false, this digit does not lead to a solution and we use backtracking.

Exercises Part 5

https://github.com/Michaeli71/PowerOfRecursionAndBacktracking