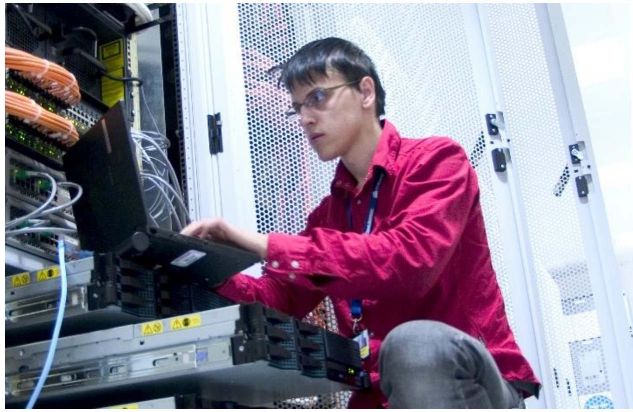
EMIL-VON-BEHRING GYMNASIUM

Exkurs: Algorithmenmuster




Algorithmenmuster

- Grundmuster von Algorithmen
- Vorteile:
  - Erleichtert das Verständnis von Algorithmen
  - Gibt Orientierung bei der Entwicklung von Lösungsalgorithmen
  - einheitliche Komplexität von Algorithmen
- Bekannte Grundmuster:
  - Verkleinerungsprinzip
  - Divide and Conquer (Teile und herrsche)
  - Dynamisches Programmieren
  - Greedy-Algorithmen
  - Backtracking

Verkleinerungsprinzip

Verkleinerungsprinzip

- Problem wird in jedem Schritt verkleinert, bis ein trivial zu lösendes Problem vorliegt.
- Vorgehen:
  - Verkleinerungsschritt
  - Erkennen, wenn Trivialfall vorliegt
  - Lösen des einfachen Problems
- Kennen wir schon – sowohl iterativ als auch rekursiv:
  - <u>iterativ</u>:

```
solange Problem nicht trivial
    verkleinere Problem
löse triviales Problem
```

  - <u>rekursiv</u>:

```
Lösung verkleinern (P: Problem)
    falls P trivial
        Lösung für P
    sonst verkleinern(verkleinere P)
```

Übungsaufgabe zum V-prinzip: Euklidischer Algorithmus

- Ein Beispiel für das Verkleinerungsprinzip ist der euklidische Algorithmus, der den größten gemeinsamen Teiler zweier Zahlen berechnet. Der euklidische Algorithmus findet häufig Anwendung in der Kryptographie.
- Beispielrechnung:

$$1071 = 1 \cdot 1029 + 42$$
$$1029 = 24 \cdot 42 + 21$$
$$42 = 2 \cdot 21 + 0$$

ggt(1071,1029) = 21

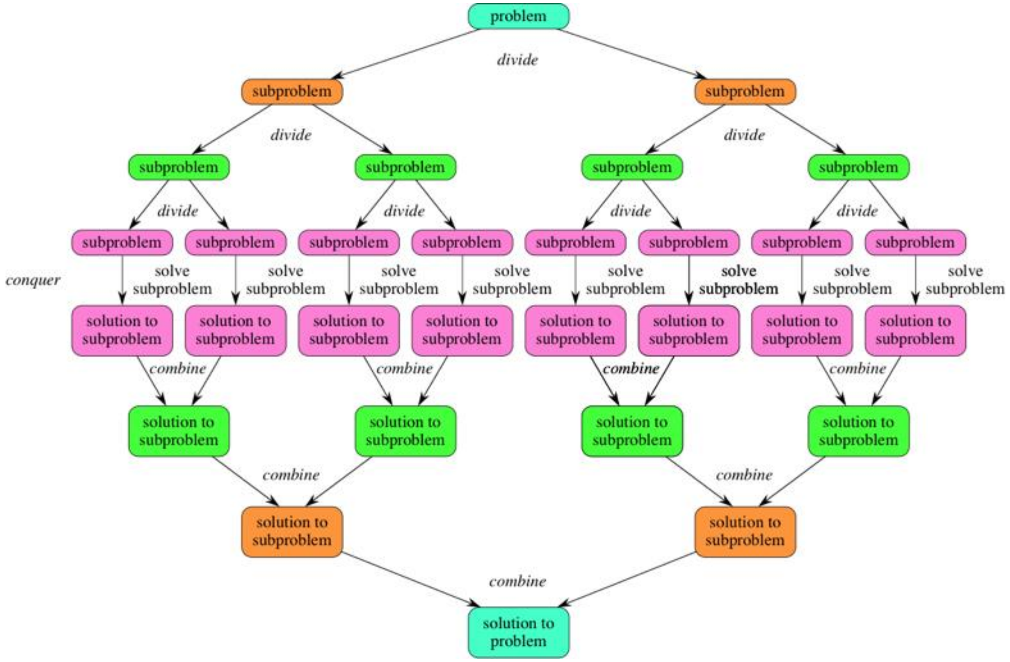
- Fünfte Grundrechenart der Informatik: Modulo-Rechnung (Divisionsrest) 17 % 5 = 2, 33 % 3 = 0
- Implementiere eine iterative und dann eine rekursive Variante.

Divide-and-Conquer

Funktionsprinzip von Divide-and-Conquer

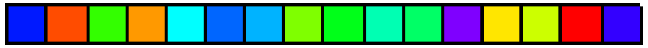
- Zerlegung eines Problems in zwei (oder mehr) Teilprobleme gleicher Art (divide)
- Lösung der Teilprobleme
- Zusammensetzen der Teillösungen zu Gesamtlösung (conquer)

```
Lösung divideAndConquer (P: Problem)
    falls P trivial
            Lösung für P
    sonst
            zusammensetzen(
                    divideAndConquer(Teil1.teileP),
                    divideAndConquer(Teil2.teileP))
```

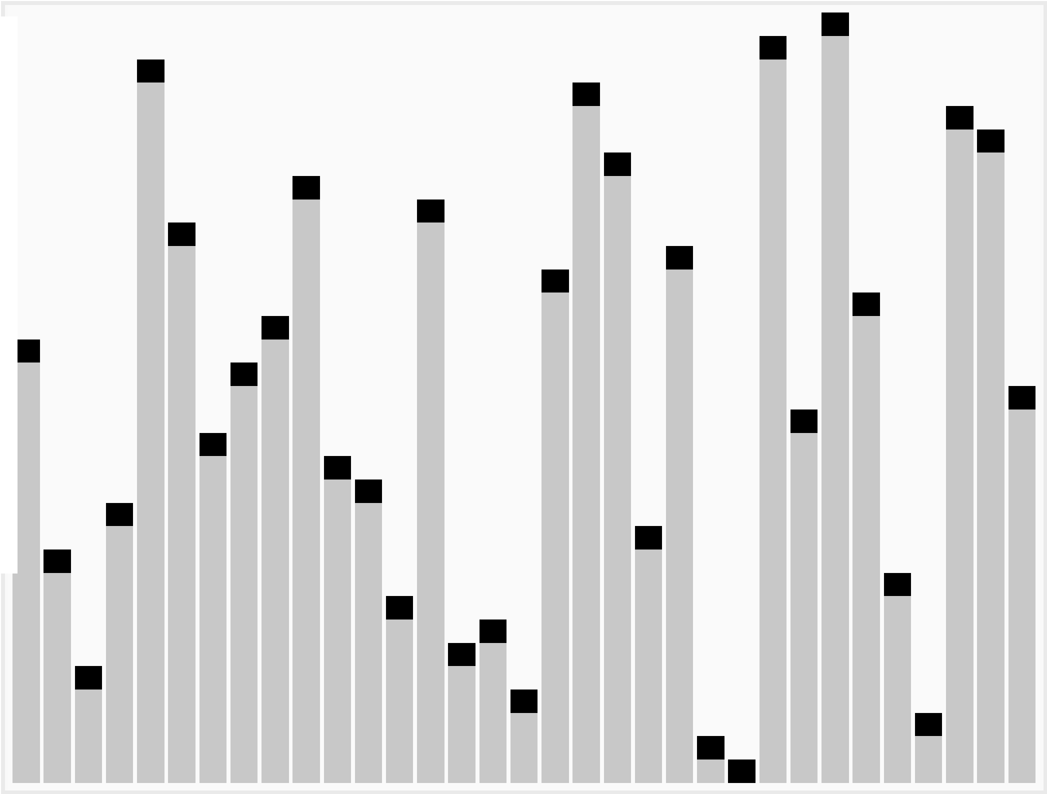


Beispiele für Divide-Conquer-Algorithmen

- MergeSort



- QuickSort



- Die Gifs sind verlinkt, einfach anklicken!

Übungsaufgaben

- Implementiere eine Nullstellensuche mit Intervallhalbierung (kein Zusammensetzen, also sprich kein Conquer-Teil):
  - Skript 10. Klasse –> Schleifen –> lokale Variablen
- Implementiere ein **iteratives** und **rekursives** Rateprogramm, welches mittels Intervallhalbierung die Zahl zwischen 1 und 1000, die du dir gedacht hattest, errät. Dabei fragt dich das Programm immer wieder, ob deine gedachte Zahl größer als eine bestimmte Zahl ist. Du antwortest über die BlueJ-Konsole mit 0 für Ja und 1 für Nein. **Siehe nächste Folie für ein Beispielablauf!**
  - Tipp: Einscannen von Werten in der Konsole funktioniert mit der Klasse Scanner –> Siehe Java-API Doc & import nicht vergessen!

Beispielausgabe für das Rateprogramm für die Zahl 333

```
Ist Ihre Zahl groesser als 500? 0:ja, 1: nein
1
Ist Ihre Zahl groesser als 250? 0:ja, 1: nein
0
Ist Ihre Zahl groesser als 375? 0:ja, 1: nein
1
Ist Ihre Zahl groesser als 313? 0:ja, 1: nein
0
Ist Ihre Zahl groesser als 344? 0:ja, 1: nein
1
Ist Ihre Zahl groesser als 329? 0:ja, 1: nein
0
Ist Ihre Zahl groesser als 337? 0:ja, 1: nein
1
Ist Ihre Zahl groesser als 333? 0:ja, 1: nein
1
Ist Ihre Zahl groesser als 331? 0:ja, 1: nein
0
Ist Ihre Zahl groesser als 332? 0:ja, 1: nein
0
Deine gedachte Zahl ist 333
```

```
Can only enter input while your programming is running
```

Übungsaufgaben

- Implementiere ein **rekursives** MergeSort, welches ein übergebenes Zahlen-Array sortiert:
  - Implementiere zunächst die Methode merge(int[] left, int[] right, int[] arr):
    - Diese Methode soll die beiden Arrays left und right zum einem Array arr zusammenfügen. Du kannst davon ausgehen, dass die beiden Arrays left und right bereits sortiert sind und zusammen die gleiche Größe haben wie das Array arr. Somit müssen die Zahlen von left und right nur in das Array arr übertragen werden.
    - Tipp: Nutze zwei Hilfsvariablen, die sich merken, an welcher Stelle, man gerade in left bzw. right ist.
  - Implementiere die Methode mergeSort(int[] arr): Falls der Rekursionsabbruchs-fall nicht greift, soll das Array arr in zwei Arrays left und right mittig aufgeteilt werden (Nutze hierfür die Methode Arrays.copyOfRange(…) Siehe Java API) und ebenfalls mit mergeSort(…) sortiert werden. Anschließend sollen die beiden Arrays left und right mittels merge(…) wieder im Array arr zusammen-gefügt werden.

Dynamische Programmierung

Übungsaufgabe: Fibonacci-Zahlen rekursiv vs. iterativ

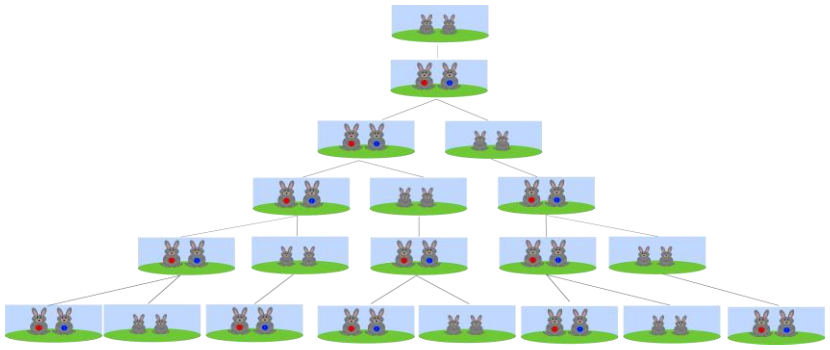
- Implementiere eine **rekursive** und eine **iterative** Variante um die n-te Fibonacci-Zahl auszugeben. fib(n) = fib(n-1) + fib(n-2)
  - Fibonacci-Folge: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …
  - Da diese Zahlenfolge sehr stark wächst, kommen die herkömmlichen Datentypen schnell an ihre Grenze. **Benutze daher die Java-Klasse BigInteger**, die beliebig große Zahlen darstellen kann. => Siehe Java-API Doc
  - Nutze eine Helper-Methode (analog zu den Graphenalgorithmen).

- **Teste anschließend deine Methoden, indem du sehr große Fibonacci-Zahlen berechnen lässt. Was fällt dir auf?**

Fibonacci-Zahlen rekursiv vs. iterativ

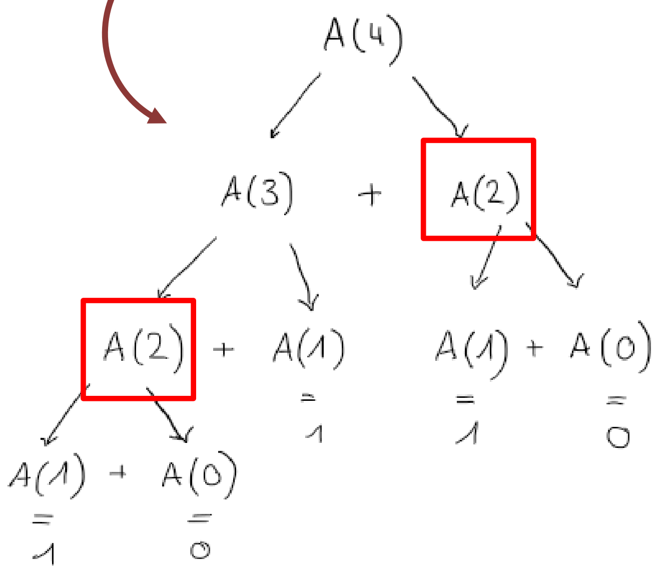
- Idee der Rekursion als Problemlösestrategie: Komplexe Sachverhalte mit rekursiv formulierten Regeln wiederholt zurückführen auf eine einfachere Aufgabe derselben Art. Schrittweiser Aufbau der Gesamtlösung aus den Lösungen der vorherigen Schritte.
- Rekursive programmierte Algorithmen
  - Sind langsamer, da jeder Funktionsaufruf Rechenzeit kostet
  - Belegen viel zusätzlichen Speicher auf dem Heap
  - Sind oft übersichtlicher und mit weniger Code zu realisieren
- Rekursion und Iteration sind gleich mächtige Sprachmittel.
  => Theoretische Informatik: WHILE-berechenbar ⇔ μ-rekursiv

Ein Blick auf die Fibonacci-Folge



https://de.wikipedia.org/wiki/Datei:FibonacciRabbit.svg

- Definition der Fibonacci-Folge:
  - $A(0) = 0$
  - $A(1) = 1$
  - $A(n) = A(n-1) + A(n-2)$



Wiederholtes Berechnen desselben Wertes notwendig!
Was passiert, wenn wir A(25) berechnen wollen?

Sehr viel repetitives Rechnen! → Ungünstig für den Aufwand (Komplexität) des Algorithmus. Abhilfe?

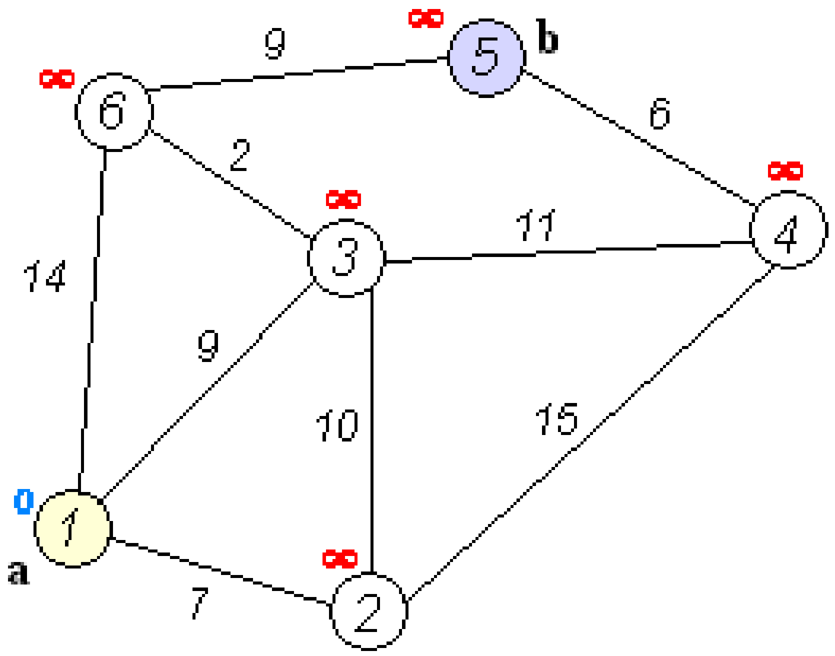
Übungsaufgabe: Dynamische Programmierung fibDP(n)

- Problem: Mehrfachberechnung von Teillösungen → Lösung: Teilergebnisse merken und wiederverwenden!
- Vorteil: Verbesserung der Laufzeit-Effizienz
- Nachteil: mehr Speicherplatz benötigt
- Wie machen wir das jetzt für unsere Fibonacci-Zahlen, wenn wir fibDP(n) berechnen wollen?
  - Array der Größe n+1 **als Attribut** anlegen
  - Basisfälle eintragen
  - Eigentliche Methode zur Berechnung der Fibonacci-Zahlen erstellen
    - Prüfen, ob an Arrayposition der zu berechnenden Zahl bereits ein Eintrag steht, dann diesen zurückgeben
    - Wenn nicht, rekursiver Aufruf der Fibonacci-Berechnung & abspeichern des hier berechneten Wertes
- Der iterative Algorithmus ist trotzdem schneller, weil dieser weniger Lese-/Schreibe-Operationen benötigt

Greedy-Algorithmen

Funktionsprinzip (Aufgabe im Skript zu Graphen)

- *Gierige* Strategie:
  - Erledige immer als nächstes den noch nicht bearbeiteten **fettesten** (d. h. größten/ kleinsten/ teuersten/ billigsten/ optimalsten/ …) Teil des Problems.
  - Wählen des vielversprechendsten Wegs
  - Normalerweise liefern Greedy-Algorithmen recht gute Ergebnisse, müssen aber nicht immer zur optimalen Lösung führen.
- Bsp.: Dijkstra-Algorithmus
  - Berechnung des kürzesten Weges zweier Knoten in einem Graphen
  - Wählt von allen erreichbaren Knoten jeweils den mit dem geringsten Gesamtaufwandskosten, der keinen Zyklus erzeugt, bis alle Knoten integriert sind



Das Gif sind verlinkt, einfach anklicken!

EMIL-VON-BEHRING GYMNASIUM

Backtracking

Funktionsweise

- Backtracking (Rücksetzverfahren):
  - Problemlösungsverfahren
  - systematisches Durchsuchen des Suchraums
  - dabei Anwendung von trial-and-error (Versuch-Und-Irrtum)
- falls für eine „Entscheidung“ mehrere Möglichkeiten existieren:
  - alle Möglichkeiten rekursiv durchprobieren (**rekursiver Aufruf in if-Bedingung**)
    - falls eine Möglichkeiten zum Erfolg führt: gut
      - Suche kann abgebrochen werden (außer man will alle Lösungen finden)
    - andernfalls:
      - Entscheidung war wohl falsch…
      - Entscheidung „rückgängig machen“

→ Backtracking findet garantiert eine Lösung, wenn überhaupt eine existiert

- Beispiel für Backtracking: Tiefendurchlauf eines Graphen

Übungsaufgabe zu Backtracking

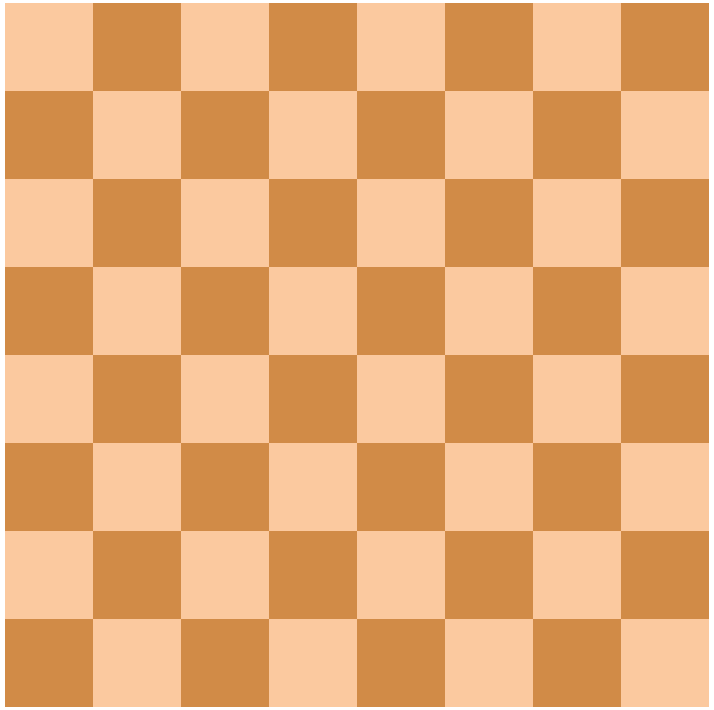
Folgende Methode soll das Feld a (garantiert der Länge 2n und beim ersten Aufruf von außen mit 0 initialisiert) mittels rekursivem Backtracking so mit Zahlen 1 <= x <= n befüllen, dass jedes x genau zweimal in a vorkommt und der Abstand zwischen den Vorkommen genau x ist. Sie soll genau dann true zurückgeben, wenn es eine Lösung gibt. Beispiele:

- n = 2 → false
- n = 3 → [3; 1; 2; 1; 3; 2]
- n = 4 → [4; 1; 3; 1; 2; 4; 3; 2]

```
boolean fill(int n, int[] a) {
        if (n <= 0) {
                return true;
        }
//TODO
        return false;
}
```

Weiteres Beispiel: Das Damenproblem

- Das Gif sind verlinkt, einfach anklicken!



- Acht Damen sollen auf einem Schachbrett so aufgestellt werden, dass keine zwei Damen einander gemäß ihren in den Schachregeln definierten Zugmöglichkeiten schlagen können. Für Damen heißt dies konkret: Es dürfen keine zwei Damen auf derselben Reihe, Linie oder Diagonale stehen.
- 92 mögliche Lösungen für das 8x8 Feld – aber wie findet man diese?
- Idee des Backtracking:
  - Algorithmus so lange ausführen, bis man an eine Grenze kommt (hier: es kann keine Dame mehr aufgestellt werden ohne Regeln zu verletzen)
  - Wenn das der Fall ist: zurück zum letzten Schritt, anderen Folgeschritt testen
  - → Versuchen, gültige Teillösung zu finden, auf dieser restlichen Weg zum Ziel aufbauen
  - → Wenn das nicht möglich ist, versuchen andere Teillösung zu finden

Zentral: Zurücksetzen von bereits getätigten Schritten

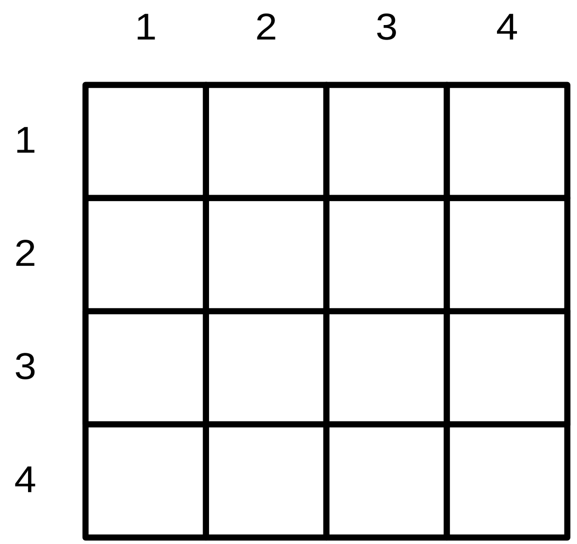
Wie funktioniert das konkret beim Damenproblem?

- Wir reduzieren erstmal auf ein kleineres Feld…

→ 4 Damen auf dem 4x4-Feld unterbringen, sodass sie sich nicht gegenseitig schlagen können

Vorgehen:

- für jede Zeile führe aus:
  - prüfe ob Feld von bereits gesetzter Dame bedroht ist
  - falls Feld nicht bedroht ist
    - setze Dame dort hin
    - setze Bedrohungen, die von gesetzter Dame ausgehen
    - falls letzte Spalte erreicht
      - gib die Lösung aus
    - falls letzte Spalte noch nicht erreicht
      - Führe Vorgehen erneut in der nächsten Spalte durch
    - zum Schluss lösche die Bedrohung und die Dame



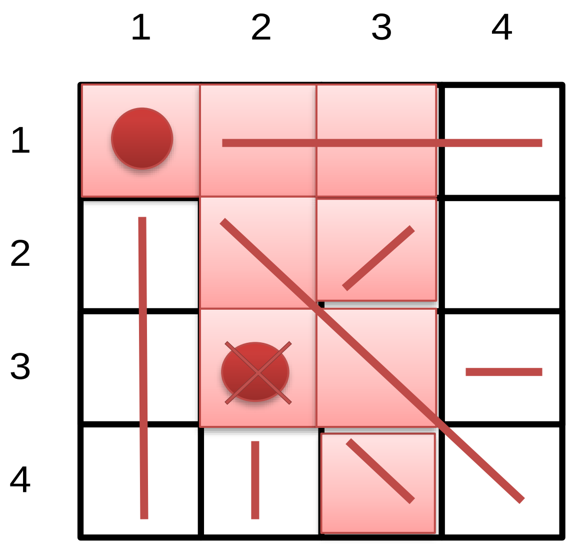
Wie funktioniert das konkret beim Damenproblem?

- Wir reduzieren erstmal auf ein kleineres Feld…

→ 4 Damen auf dem 4x4-Feld unterbringen, sodass sie sich nicht gegenseitig schlagen können

Vorgehen:
- für jede Zeile führe aus:
    - prüfe ob Feld von bereits gesetzter Dame bedroht ist
    - falls Feld nicht bedroht ist
        - setze Dame dort hin
        - setze Bedrohungen, die von gesetzter Dame ausgehen
        - falls letzte Spalte erreicht
            - gib die Lösung aus
        - falls letzte Spalte noch nicht erreicht
            - Führe Vorgehen erneut in der nächsten Spalte durch
    - zum Schluss lösche die Bedrohung und die Dame

Fortsetzen des vorherigen Aufrufs des Vorgehens



usw.… bis Lösung gefunden wurde, dann wird sie graphisch ausgegeben

Nochmal zum Nachvollziehen: https://www.youtube.com/watch?v=B3Vr1r3J8iI

Übungsaufgabe N-Damenproblem

- Implementiere das N-Damenproblem:
  - Nutze hierfür das Backtracking-Algorithmusmuster ähnlich zu der Aufgabe auf der vorherigen Folie.
  - Deinem Programm sollte ein Wert übergeben werden können, sodass man die „Größe des Schachfelds“ und damit auch die Anzahl der Damen bestimmen kann.
  - Tipp: Es bietet sich in jedem Fall an eine Methode zu schreiben die überprüft, ob eine Dame auf einem gewissen Feld stehen darf.
  - Tipp: Schreibe eine Methode den aktuellen Stand und/oder das Ergebnis auf der Konsole entsprechend dem Schachmuster ausgibt. Dies erleichtert das Verifizieren der Lösung und das Debugging.
  - Beispielausgabe für das 4-Damenproblem:

```
0 1 0 0
0 0 0 1
1 0 0 0
0 0 1 0
```