Einführung zur Aufgabengruppe 3

Alle Literaturhinweise beziehen sich auf die Vorlesung
Modellierung und Programmierung 1

1 Rekursion (s. Kapitel Methoden) 1-2

1.1 Beispiel „Eine unendliche Geschichte“ 1-2

1.2 Beispiel „Türme von Hanoi“ 1-2

1.3 Beispiel „r = ggT(m, n)“ 1-3

1.4 Backtracking-Verfahren 1-4

1.5 Branch-and-Bound-Verfahren 1-5

2 Beispiel „Damenproblem von C. F. Gauß (1850)“ 2-7

2.1 Implementieren der Lösungsmenge 2-7

2.2 Implementieren der Rekursion 2-8

2.3 Zusatz: Benutzerschnittstelle zum Dameproblem 2-9

1 Rekursion (s. Kapitel Methoden)

Eine Rekursion definiert die Lösung eines Problems (einer Funktion, eines Verfahrens) induktiv: Ausgehend von einer Anfangslösung wird eine Lösung aus einer bereits bekannten Lösung berechnet.

Die **Fakultät** natürlicher Zahlen lässt sich rekursiv definieren: **(I) 0! = 1**
(II) n! = n * (n - 1)!

4! = 4 * 3! = 4 * 3 * 2! = 4 * 3 * 2 * 1! = 4 * 3 * 2 * 1 * 0! = 4 * 3 * 2 * 1 * 1 = 24

Direkte Rekursionen: Probleme, die sich selbst aufrufen.
Indirekte Rekursionen: Probleme, die sich wechselseitig aufrufen.

Die **Rekursionstiefe** gibt die Anzahl der verschachtelten Problemaufrufe an. Im obigen Beispiel ist die Rekursionstiefe 5.

In Java sind *direkte* und *indirekte* Rekursionen sowohl bei den Methoden als auch bei den Attributen erlaubt. Eine Ausnahme bildet hierbei die Methode `main`, da sie grundsätzlich nicht aufgerufen werden kann.

1.1 Beispiel „Eine unendliche Geschichte“

>>
Es war einmal ein Mann, der hatte sieben Söhne. Die sieben Söhne sprachen: „Vater erzähle uns eine Geschichte!“ Da fing der Vater an:

Es war einmal ein Mann, der hatte sieben Söhne. Die sieben Söhne sprachen: „Vater ...

Und wenn sie nicht gestorben sind, so erzählen sie noch heute.
<<

Beispiel ***Erzaehlung.java***

1.2 Beispiel „Türme von Hanoi“

Türme von Hanoi des Monsieur Claus,
Anagramm des Mathematikers Lucas (vor 100 Jahren)

Legende: Zu Benares in Indien gibt es einen Tempel, in dem seit vielen Jahrhunderten Priester bemüht sind, einen Turm so umzulegen, wie Brahma es ihnen selbst geboten hat. Der Turm besteht aus 64 der Größe nach geordneten und auf einer Stange aufgesteckten dünnen goldenen Plättchen. Die Aufgabe lautet, den wenigen Zentimeter hohen Turm von der ersten Stange auf die letzte Stange der drei zur Verfügung stehenden Stangen zu schaffen. Dabei darf jeweils nur eine Scheibe transportiert werden und niemals eine größere Scheibe auf eine kleinere gelegt werden. Sobald die Mönche dieses Ritual hinter

sich gebracht haben, wird der Tempel zu Staub zerfallen und die Welt mit einem Donnerschlag untergehen! Wann wird wohl mit der indischen Götterdämmerung zu rechnen sein?

Beispiel ***TvH.java***

hanoi(int k, int a, int b)

```
/**
 * hanoi: rekursive Funktion
 * @param k Anzahl der Scheiben
 * @param a Startstange
 * @param b Zielstange
 *   1.  (k-1) Scheiben von Stange a nach Stange c
 *   2.   k. Scheibe von Stange a nach Stange b
 *   3.  (k-1) Scheiben von Stange c nach Stange b
 */
public static void hanoi( int k, int a, int b)
{
  if( k > 0)
  {
    hanoi( k - 1, a, 6 - a - b);                    // 1.
                                                    // 2.
    System.out.println("" + k + ": " + a + " => " + b);
    hanoi( k - 1, 6 - a - b, b);                    // 3.
  }
}
```

1.3 Beispiel „r = ggT(m, n)“

Jede *Rekursion* ist auflösbar, d.h. man kann dafür auch eine *Iteration* verwenden. Iterationen sind durch *nichtrekursive* Methodenaufrufe in der Programmausführung *effizienter*.

iteratives Programm *r = ggT(m, n)*

```
public static int ggT( int a, int b)
  throws NPlusException
{
  if( m <= 0 || n <= 0) throw new NPlusException();
  int c;
  do
  {
    c = a % b; a = b; b = c;
  } while( c != 0);
  return a;
}
```

rekursives Programm *r = ggT(m, n)*

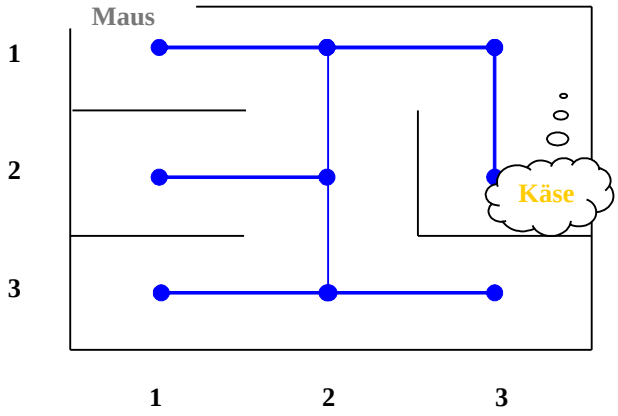
```
public static int ggT(int a, int b)
  throws NPlusException
{
  if( m <= 0 || n <= 0) throw new NPlusException();
  if( b != 0) return ggT( b, a % b);
  return a;
}
```

1.4 Backtracking-Verfahren

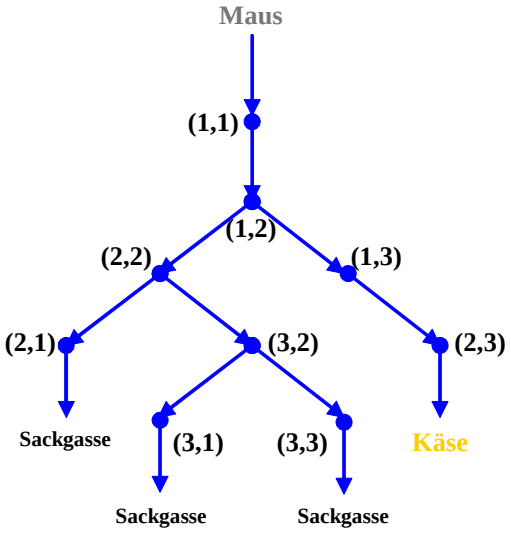
Backtracking: Zurückverfolgung, Rücksetzverfahren

Die Gesamtlösung wird schrittweise aus Teillösungen zusammengesetzt. Falls man dabei in eine Sackgasse gerät, werden der letzte Schritt oder mehrere der letzten Schritte rückgängig gemacht. Aus einer so reduzierten Teillösung versucht man, auf einen anderen Weg eine Gesamtlösung aufzubauen.

Betrachtet man ein Labyrinth, in dem eine Maus zu ihrem Käse laufen soll:



Das Problem lässt sich als Baum darstellen:



Die Maus durchläuft systematisch den Baum, dabei werden die Lösungsschritte rückgängig gemacht, die zu einer Sackgasse führen:
Durchläuft die Maus den Baum von links nach rechts, so nimmt sie zuerst den Weg ((1, 1), (1, 2), (2, 2), (2, 1)) und landet in einer Sackgasse. Dann geht sie einen Schritt zurück zum Knoten (2, 2) und durchläuft von dort aus den Teilweg ((3, 2), (3, 1)) in die nächste Sackgasse. Also wieder einen Schritt zurück zum Knoten (3, 2) und schließlich weiter zum Knoten (3, 3). Auch hier ist der Käse nicht zu finden. Jetzt bleibt der Maus nichts anderes übrig, als drei Schritte zurück zum Knoten (1, 2) zu marschieren, denn erst hier befindet sich noch eine Abzweigung, welche die Maus noch nicht passiert hat. Schließlich findet sie von diesem Knoten über den Teilweg ((1, 3), (2, 3)) zum Käse.

Die Maus musste erst alle im Baum möglichen Wege durchlaufen, bis sie schließlich ihren Käse gefunden hat. Das ist der ungünstige Fall. Hätte man den Käse an einem der anderen Endknoten versteckt oder hätte die Maus eine andere Strategie beim Durchlaufen des Labyrinths verfolgt, so wäre sie eher am Ziel gewesen.

Allgemeine Beschreibung des Backtracking-Algorithmus
Vorausgesetzt der Weg bis zum Knoten ***K*** sei bekannt. Dann wird der Knoten ***K*** folgendermaßen behandelt:

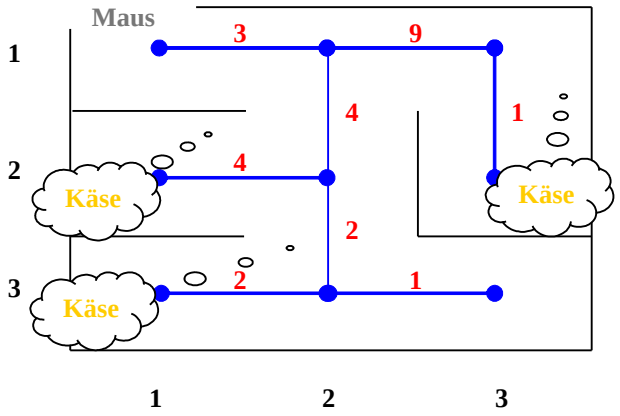
rekursion(Knoten *K*)
1. Ist der Knoten ***K*** ein Endknoten:
   a. Das Ergebnis wird verarbeitet.
   b. Ist das Problem gelöst und nur eine Lösung gesucht, wird die **Rekursion** beendet, sonst wird der **Rekursionsschritt** beendet.
2. Ist der Knoten ***K*** kein Endknoten, so werden für *jeden* direkten Nachfolger ***L*** des Knotens ***K*** folgende Schritte ausgeführt:
   a. Man nehme Knoten ***L*** in den Weg auf.
   b. Man behandle Knoten ***L***: **rekursion(Knoten *L*)**. // Rekursionsschritt
   c. Man entferne Knoten ***L*** aus dem Weg.

1.5 Branch-and-Bound-Verfahren

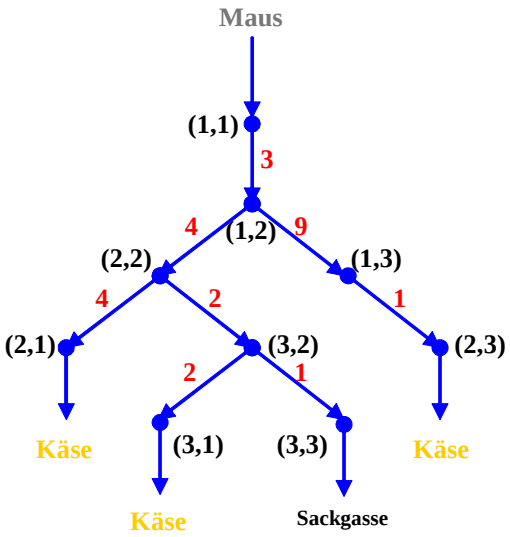
Branch: Verzweigung
Bound: Beschränkung

Dieses Verfahren ist ein *spezielles* Backtracking-Verfahren. Durch eine *Zielfunktion* lassen sich Fälle von vornherein ausschließen, so dass man die Suche nach einer Lösung beschleunigen kann. Bei der Bearbeitung dieses Problems werden die Knoten des Baumes bewertet und parallel zum Wegsuchen die Zielfunktion bis zu dieser Teillösungen berechnet. Ist dieser Wert schlechter als eine bereits gefundene Lösung, so werden die folgenden Zweige des Baumes gar nicht erst durchlaufen.

Man betrachte wiederum das Labyrinth mit der Maus. Jetzt sind aber mehrere Käsestücken im Labyrinth verteilt. Die Wege dorthin sind unterschiedlich lang. Die Maus ist lernfähig. Das Problem besteht jetzt in der Suche nach dem schnellsten Weg zu einem der Käse.



Das Problem lässt sich in einen bewerteten Baum abbilden:



Die Maus durchläuft systematisch den Baum. Die Wegezeiten werden parallel dazu berechnet. Lösungsschritte werden rückgängig gemacht, wenn sie zu einer Sackgasse kommt, Käse findet oder ihre Wegezeit schlechter als bereits gefundene Lösungszeiten ist.

Arbeitet die Maus den Baum wiederum von links nach rechts ab, so erhält sie über den Weg ((1, 1), (1, 2), (2, 2), (2, 1)) nach 11 Zeiteinheiten ihren Käse. Diesen Weg merkt sie sich als den vorläufig besten. Der nächste Weg vom Knoten (2, 2) über die Knoten (3,2) und (3, 1) liefert mit ebenfalls 11 Zeiteinheiten kein besseres Ergebnis. Da nur ein Weg mit der besten Zeit gesucht wird, so viel Käse schafft die Maus nicht, braucht sie sich diesen Weg nicht zu merken. Kehrt die Maus an den Knoten (3, 2) zurück und läuft zum Knoten (3, 3), so erreicht sie diesen Endknoten in nur 10 Einheiten. Leider landet die Maus hier in einer Sackgasse. Untersucht man jetzt den letzten möglichen Weg, so muss die Maus zurück bis zum Knoten (1, 2). Man kann bereits am Knoten (1, 3) die Suche abbrechen, denn die Wegzeit bis dorthin liegt mit 12 Einheiten schon über den bisher gefundenen Wert. Das Verfahren bricht ab, alle möglichen Wege sind untersucht. Der beste Weg war der zuerst gefundene Weg ((1, 1), (1, 2), (2, 2), (2, 1)) mit 11 Zeiteinheiten.

Allgemeine Beschreibung des Branch-and-Bound-Algorithmus
Vorausgesetzt der Weg bis zum Knoten ***K*** und seine Bewertung ***B*(*K*)** seien bekannt. Dann wird der Knoten ***K*** folgendermaßen behandelt:

rekursion(Knoten *K*)
1. Ist der Knoten ***K*** ein Endknoten:
   a. Ist die Bewertung ***B*(*K*)** besser als vorher berechnete Wege, so werden der Weg und seine Bewertung als bisher bester Weg abgespeichert.
   b. Der **Rekursionsschritt** wird beendet.
2. Ist der Knoten ***K*** kein Endknoten, so werden für jeden direkten Nachfolger ***L*** des Knotens ***K*** folgende Schritte ausgeführt:
   a. Man nehme Knoten ***L*** in den Weg auf und berechne aus ***B*(*K*)** seine Bewertung ***B*(*L*)**.
   b. Ist Bewertung ***B*(*L*)** besser als die bisherige beste Wegbewertung, so behandle man Knoten ***L***: **rekursion(Knoten *L*).** **// Rekursionsschritt**
   c. Man entferne Knoten ***L*** aus dem Weg und setze die Bewertungsfunktion auf ***B*(*K*)** zurück.

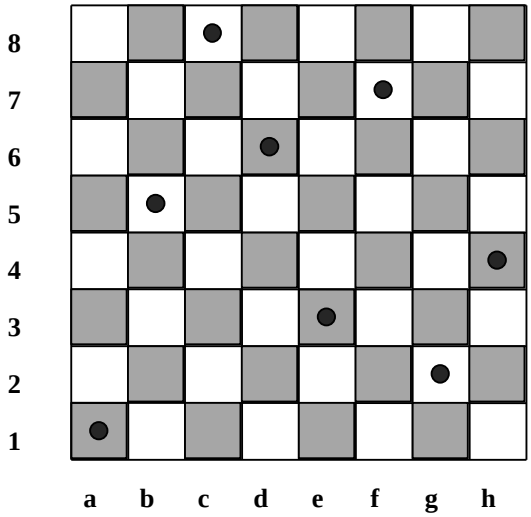
2 Beispiel „Damenproblem von C. F. Gauß (1850)“

Man setzt 8 Damen auf ein Schachbrett so, dass sie sich gegenseitig nicht bedrohen. Gesucht sind alle Lösungen:

- Jede Zeile/ Spalte enthält genau eine Dame.
- Jede positive/ negative Diagonale enthält höchstens eine Dame.

Gauß gelang es nicht, alle 92 Lösungen (bzw. 12 Lösungen, ohne Rotationen und Spiegelungen) zu ermitteln.

Lösungsbeispiel a1 b5 c8 d6 e3 f7 g2 h4

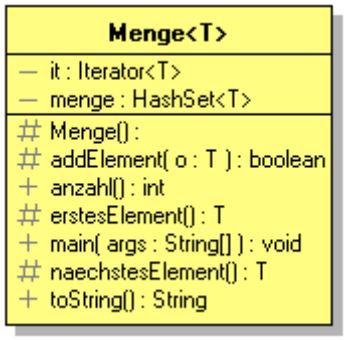


2.1 Implementieren der Lösungsmenge

Eine Klasse **`DameLoesung`** stellt eine Lösung des Dameproblems dar. Insbesondere werden durch Überschreiben der Methode **`public boolean equals( Object obj)`** der Klasse **`java.lang.Object`** rotations- und spiegelgleiche Lösungen als gleich erkannt. Da gleiche Lösungen einen gleichen Hashcode brauchen, wird auch die Methode **`public int hashCode()`** aktualisiert.

Mehrere Damelösungen werden in einer Menge zusammengefasst. Durch das Überschreiben der Methoden **`equals`** und **`hashCode`** werden dabei rotations- und gespiegelte Lösungen als gleich aufgefasst und nur einmal abgespeichert.

Eine generische Klasse **`Menge<T>`** definiert übliche Mengenmethoden für Objekte der Klasse **`HashSet<T>`**. Sie wird oft bei Anwendungen, die mit Mengen arbeiten, gebraucht und wurde deshalb bewusst so allgemein gehalten. In dieser Klasse wird gleichzeitig zur Menge selbst ein Iterator bereitgestellt, welcher einen sequentiellen Zugriff auf die Elemente der Menge ermöglicht.



Modellierung und Programmierung **Klassendiagramm, Dokumentation, *Menge.zip***

Im Beispiel erhält man die Menge aller Damelösungen des Dameproblems als Objekt der Klasse **`Menge<DameLoesung>`**.

2.2 Implementieren der Rekursion

Schließlich wird mit dem Backtracking-Algorithmus das Dameproblem gelöst:

Backtracking-Algorithmus
Vorausgesetzt, ***s*** Spalten seien so mit Damen besetzt, dass diese sich nicht gegenseitig bedrohen. Dann wird die (***s* + 1**). Spalte folgendermaßen behandelt, ***r* = *s* + 1**:

setzeDame(Spalte *r*)
1. ***r* > 8**:
   a. Lösung verarbeiten,
   b. Rekursionsschritt abbrechen.

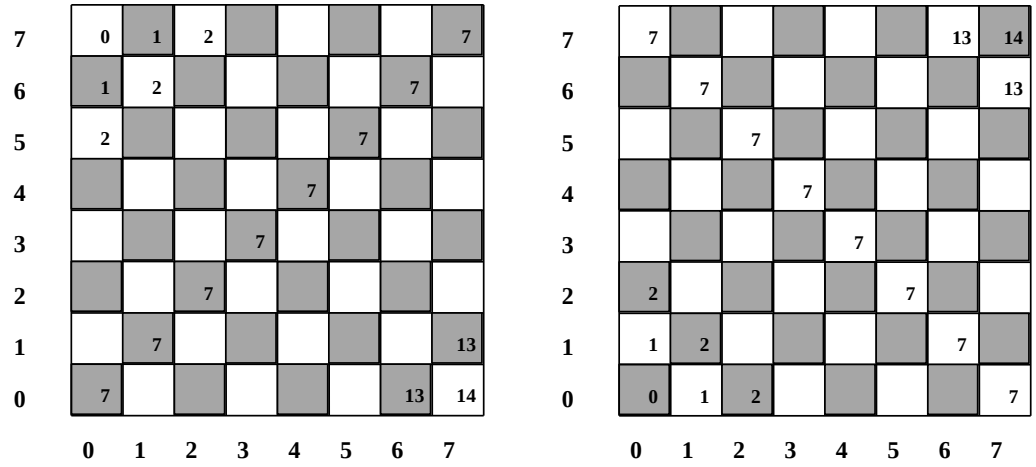
2. ***r* <= 8**, so werden die Zeilen ***z* = 1 .. 8** wie folgt behandelt:
   Wird das ***Feld*** (***z***, ***r***) von den schon gesetzten Damen nicht bedroht, so wird
   a. eine Dame auf das ***Feld*** (***z***, ***r***) gesetzt,
   b. die ***r* + 1**. Spalte wird ausgewertet: **setzeDame(Spalte *r* + 1)**,
   c. die Dame wird wieder aus dem ***Feld*** (***z***, ***r***) entfernt.

Datentypen zur Realisierung:

int spalten[8]:	Feld für Zeilennummer, in der Dame steht.
boolean zeilen[8]:	Markierungsfeld, Zeile bedroht/nicht bedroht
boolean posDiag[8]:	Markierungsfeld, positive Diagonale bedroht/nicht bedroht
boolean negDiag[8]:	Markierungsfeld, negative Diagonale bedroht/nicht bedroht

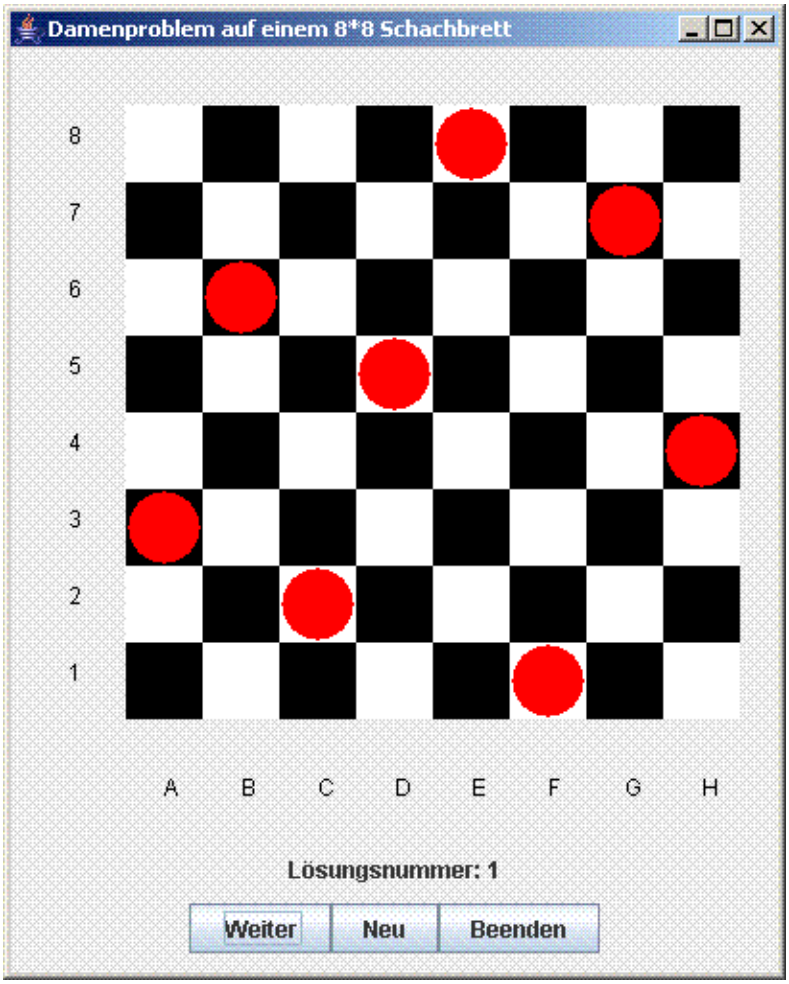
Indexrechnung:	spalten a..h:	0..7
	zeilen 1..8:	0..7
	posDiag 1..15:	7 + Spalte - Zeile
	negDiag 1..15:	Spalte + Zeile

posDiag: negDiag:



Modellierung und Programmierung **Klassendiagramm, Dokumentation, *Dame.zip***

2.3 Zusatz: Benutzerschnittstelle zum Dameproblem



Modellierung und Programmierung mit MVC-Architektur
Klassendiagramm, Dokumentation, *DameMVC.zip*