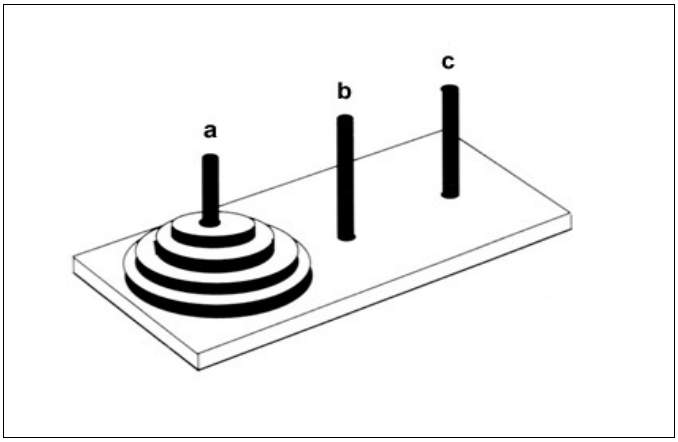
Rekursion - Teil VII - Backtracking

In diesem letzten Abschnitt zur Rekursion möchte ich noch auf einige spannende und weithin bekannte Probleme hinweisen, die in einem Kapitel über Rekursion nicht fehlen dürfen.

Türme von Hanoi



Bei diesem Spiel geht es darum, einen Stapel von gelochten Scheiben von einem Stab A zu einem Stab C zu verschieben, wobei der Stab B als Hilfsstab verwendet werden kann.

Dabei darf in jedem Schritt nur eine Scheibe umgelegt werden und eine Scheibe darf immer nur auf einer größeren Scheibe platziert werden.

Weitere Informationen findet man unter http://de.wikipedia.org/wiki/türme_von_hanoi

Probiere das Spiel aus: http://www.mathematik.ch/spiele/hanoi_mit_grafik/ oder verwende Münzen.

Man beachte, dass die Lösung einen ähnlichen Aufwand wie das Problem der Fibonacci-Zahlen hat. In der Tabelle sind die Laufzeiten angegeben, unter der Voraussetzung, dass man eine Scheibe pro Sekunde umschichtet.

Strategie der rekursiven Lösung

Anzahl Scheiben	Benötigte Zeit
5	31 Sekunden
10	17,1 Minuten
20	12 Tage
30	34 Jahre
40	348 Jahrhunderte
60	36,6 Milliarden Jahre
64	585 Milliarden Jahre

Um den rekursiven Algorithmus zu verstehen, ist die folgende Geschichte hilfreich:

Der älteste Mönch eines Klosters erhält die Aufgabe, einen Turm aus 64 Scheiben zu versetzen. Da er die komplexe Aufgabe nicht bewältigen kann oder will, gibt er dem zweitältesten Mönch die Aufgabe, die oberen 63 Scheiben auf einen Hilfsplatz zu versetzen. Er selbst (der Älteste) würde dann die große letzte Scheibe zum Ziel bringen. Dann könnte der Zweitälteste wieder die 63 Scheiben vom Hilfsplatz zum Ziel bringen.

Der zweitälteste Mönch fühlt sich der Aufgabe ebenfalls nicht gewachsen. So gibt er dem drittältesten Mönch den Auftrag, die oberen 62 Scheiben zu transportieren, und zwar auf den endgültigen Platz. Er selbst (der Zweitälteste) würde dann die zweitletzte Scheibe an den Hilfsplatz bringen. Schließlich würde er wieder den Drittältesten beauftragen, die 62 Scheiben vom Zielfeld zum Hilfsplatz zu schaffen. Dies setzt sich bis zum 64. Mönch (dem Jüngsten) fort, der die obenauf liegende kleinste Scheibe alleine verschieben kann.

Da es 64 Mönche im Kloster gibt und alle viel Zeit haben, können sie die Aufgabe in endlicher, wenn auch sehr langer Zeit erledigen.

Quelle: http://de.wikipedia.org/wiki/türme_von_hanoi

Programm: Türme von Hanoi

```
public class Hanoi {

  static void bewege(String Quelle, String Hilf, String Ziel, int n){
    if (n==1)
      System.out.println("Bewege Scheibe " +n+ " von " + Quelle + " nach " + Ziel);
    else{
      bewege(Quelle, Ziel, Hilf, n-1);
      System.out.println ("Bewege Scheibe " +n+ " von " + Quelle + " nach " + Ziel);
      bewege(Hilf, Quelle, Ziel, n-1);
    }
  }

  public static void main (String args []) {
    System.out.println ("Tuerme von Hanoi\n");
    bewege("A", "B", "C", 3);
  }
}
```

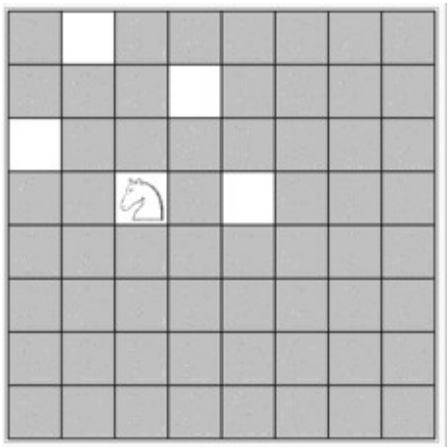
Neben den "divide and conquer"- Algorithmen gibt es noch eine weitere Klasse von Verfahren, die man kennen sollte, die *Backtracking*-Algorithmen.

Backtracking-Algorithmen

Backtracking geht nach dem "*trial and error"-Prinzip* vor (Versuch und Irrtum), das heißt, es wird versucht, eine erreichte Teillösung zu einer Gesamtlösung auszubauen. Wenn absehbar ist, dass eine Teillösung nicht zu einer endgültigen Lösung führen kann, wird der letzte Schritt beziehungsweise die letzten Schritte zurückgenommen, und es werden stattdessen alternative Wege probiert. Auf diese Weise ist sichergestellt, dass alle in Frage kommenden Lösungswege ausprobiert werden. Mit Backtracking-Algorithmen wird eine vorhandene Lösung entweder gefunden (unter Umständen nach sehr langer Laufzeit), oder es kann definitiv ausgesagt werden, dass keine Lösung existiert.

Quelle: http://de.wikipedia.org/wiki/Backtracking

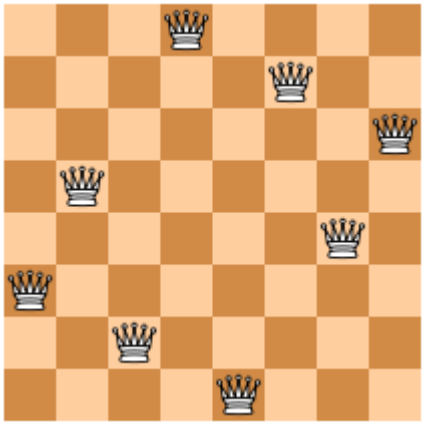
Springerproblem



Ein typisches Beispiel für ein Problem, das mittels Backtracking gelöst werden kann, ist das *Springerproblem*. Dabei versucht man einen Springer auf einem Schachbrett so zu bewegen, dass jedes Feld genau einmal besucht wird. Eine Animation findet man beispielsweise unter http://de.wikipedia.org/wiki/Springerproblem

Der Algorithmus beginnt bei einem Startfeld und speichert in einer Liste alle von dort erreichbaren Felder. Das Erste dieser Felder wird anschließend betreten und es wird wieder eine Liste aller zur Verfügung stehenden Felder angelegt (Rekursion). Entweder schafft es der Springer auf diese Art alle Felder des Bretts zu besuchen oder er erreicht einen Punkt, an dem ihm keine Felder mehr zur Verfügung stehen. Wenn keine erlaubten Züge mehr vorhanden sind, wird der Rekursionsschritt verlassen und wieder zu einer Ebene höher verzweigt. Dort wird der nächste Zug aus der Liste als möglicher nächster Schritt verwendet. Auf diese Art werden letztlich alle möglichen Zugfolgen abgearbeitet und es kann entschieden werden, ob eine Lösung existiert. Wenn man die Züge speichert, kann die Lösung auch angegeben werden.

8-Damenproblem



Das 8-Damenproblem ist dem Springerproblem sehr ähnlich. Bei dem 8-Damenproblem sollen 8 Damen auf einem Schachbrett so positioniert werden, dass sie sich gegenseitig nicht bedrohen. Auch dieses Problem lässt sich gut mittels Backtracking lösen. Mehr Informationen findet man unter:

http://de.wikipedia.org/wiki/Damenproblem

Labyrinth

Auch bei der Lösung von Labyrinthproblemen kommt häufig Rekursion zum Einsatz. Wie schon bei dem Springerproblem startet man bei einer Startposition. Entweder ist diese Position das Ziel oder man markiert diese Position als bereits untersucht und speichert sich alle von dort aus erreichbaren Felder in einer Liste (z.B. einer Queue). Nun folgt der Rekursionsschritt. Man entnimmt ein noch nicht untersuchtes Feld aus der Queue und prüft, ob es das Ziel ist. Falls ja, bricht man ab, sonst markiert man das Feld als besucht, speichert alle von dort aus erreichbaren Felder hinten in die Queue. Anschließend folgt der nächste Rekursionsaufruf. Man entnimmt wieder ein noch nicht besuchtes Feld aus der Queue usw. Letztlich wird man so das Ziel finden, wenn es denn überhaupt erreichbar ist. Man kann also entscheiden, ob das Ziel erreichbar ist oder nicht.

Um einen kürzesten Weg vom Start zum Ziel angeben zu können, muss man allerdings noch ein wenig mehr Aufwand betreiben, z.B. kann man jedes Feld mit einer Zahl markieren, die den Abstand vom Startpunkt angibt. Der Startpunkt bekommt, wenn er untersucht wurde, die Zahl 0. Alle von dort aus erreichbaren Felder, die in der Queue gespeichert werden, bekommen die Zahl 1 usw. Wenn nun das Ziel gefunden wurde (und z.B. die Zahl *n* hat), muss man lediglich ein Nachbarfeld des Ziels suchen, das die Zahl *n-1* hat. Von dort aus sucht man ein Nachbarfeld mit der Zahl *n-2* usw. Auf diese Weise kann man sich so vom Ziel in *n* Schritten zum Start durchhangeln.