Rekursion - Teil V

Neben den einfachen, einführenden Beispielen gibt es auch einige Anwendungen, wo Rekursion wirklich sinnvoll ist, da eine iterative Lösung kaum möglich ist. Grundsätzlich lässt sich aber beweisen, dass Rekursion und Iteration gleich mächtig sind. Genauer:

Es lässt sich beweisen (mit Methoden der theoretischen Informatik), dass Iteration und Rekursion äquivalent sind. D. h., dass man jedes rekursive Programm in ein iteratives Programm umschreiben kann und umgekehrt. Allerdings ist in der Praxis meist eine der beiden Versionen deutlich kürzer und deutlich besser verständlich.

Zwei der Algorithmen, die sich durch Rekursion "leicht" lösen lassen und wo eine iterative Lösung zu erheblichen Problemen führen würde, sollen nun kurz vorgestellt werden.

Alle Permutationen einer Menge

Es gibt Probleme, die kann man dadurch lösen, dass man alle Anordnungen von Elementen einer Menge erzeugt. Beispielsweise könnte gefragt sein, auf welche Arten man *n* Personen nebeneinander in eine Reihe stellen kann oder in welcher Reihenfolge man *n* Städte besuchen kann.

Diese Aufgabe lässt sich rekursiv deutlich besser lösen als iterativ. Allerdings ist der Algorithmus recht kompliziert zu durchschauen.

```
class Permutationen {

  static void ausgabe(char[] a) {
    for (int i = 0; i < a.length; i++) {
      System.out.print(a[i]);
    }
    System.out.print(" ");
  }

  static void vertausche(char[] a, int i, int j) {
    char dummy;
    dummy = a[i];
    a[i]  = a[j];
    a[j]  = dummy;
  }

  static void perm(char[] a, int startIndex) {
    if (startIndex == a.length-1) {
      ausgabe(a);
    } else {
      for (int i = startIndex; i < a.length; i++) {
        vertausche(a, i, startIndex);
        perm(a, startIndex+1);
        vertausche(a, i, startIndex);
      }
    }
  }

  public static void main(String[] args) {
    String s = IOTools.readLine("Geben Sie den zu permutierenden String ein: ");
    perm(s.toCharArray(), 0);
  }
}
```

Ausgabe:

```
Das Programm permutiert einen String.

Geben Sie den zu permutierenden String ein:  1234

1234 1243 1324 1342 1432 1423 2134 2143 2314 2341 2431 2413 3214 3241 3124 3142
3412 3421 4231 4213 4321 4312 4132 4123
```

Analyse des Algorithmus:

Versuchen wir das Programm zu analysieren. Die beiden Methoden **ausgabe()** und **vertausche()** können wir dabei vernachlässigen.

```
static void perm(char[] a, int startIndex) {

  if (startIndex == a.length-1) {
    ausgabe(a);
  } else {
    for (int i = startIndex; i < a.length; i++) {
      vertausche(a, i, startIndex);
      perm(a, startIndex+1);
      vertausche(a, i, startIndex);
    }
  }
}
```

Der Fall n=1:

Angenommen, der String besteht nur aus der Zahl 1. Dann wird die Methode mit **perm('1',0)** aufgerufen und es wird sofort der Wert 1 auf den Bildschirm geschrieben.

Der Fall n=2:

Der Algorithmus wird mit **perm('12',0)** gestartet. Der Algorithmus verzweigt nun in den else-Fall und führt eine for-Schleife von 0 bis 1 aus.

Für i=0 vertauscht er in dem Array '12' die 1 mit sich selbst (eigentlich unnötig) und permutiert das Array beginnend bei 2. Die Kombination '12' wird im nächsten Rekursionsschritt ausgeben und der Algorithmus kehrt wieder eine Ebene höher zu der for-Schleife zurück. Abschließend wird wieder die 1 mit sich selbst vertauscht (eigentlich auch unnötig).

Für i=1 wird die 2 an die erste Stelle vertauscht und der nächste Rekursionsaufruf getätigt, der '21' ausgibt. Anschließend wird die 2 wieder nach hinten getauscht.

Der Fall n=3:

Der Algorithmus wird mit **perm('123',0)** gestartet. Der Algorithmus verzweigt nun in den else-Fall und führt eine for-Schleife von 0 bis 2 aus.

Letztlich tauscht der Algorithmus mittels der for-Schleife jedes Element des Arrays einmal nach vorne und ruft sich selbst auf.

Es stellt sich die Frage, warum danach immer wieder zurück getauscht werden muss.

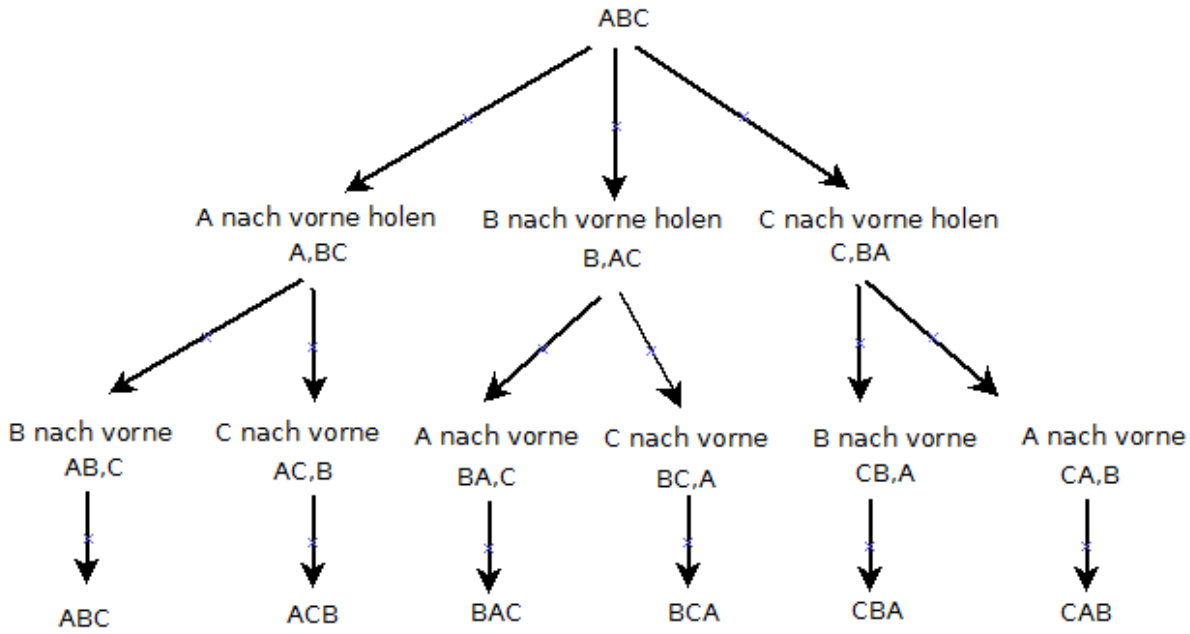
Wenn man das Zurücktauschen wegließe, könnte man nicht mehr davon ausgehen, dass die for-Schleife wirklich jedes Element einmal nach vorne getauscht hat, denn in den tieferen Verzweigungen der Rekursion werden ja auch wieder Vertauschungen durchgeführt (die dann nicht mehr rückgängig gemacht würden). Durch das Zurücktauschen ist nach Beendigung eines Rekursionsschritts stets wieder die ursprüngliche Anordnung gegeben.

Die folgende Grafik veranschaulicht die Aufrufe mit der Eingabe 'ABC'.

Aufruf:

```
perm('ABC',0)
```

Rekursionsstruktur:



Ausgabe:

```
Das Programm permutiert einen String.
Geben Sie den zu permutierenden String ein:  ABC
ABC ACB BAC BCA CBA CAB
```

Die Potenzmenge einer Menge

Ein weiterer Algorithmus, der thematisch zu den Permutationen passt, ist die Erzeugung einer Potenzmenge. Dabei bezeichnet man die Menge aller Teilmengen einer Menge als Potenzmenge. Die Menge M ={a,b} hat z. B. die Potenzmenge P(M)={{},{a},{b},{ab}}. Man beachte, dass es hier nicht auf die Reihenfolge der Elemente ankommt.

Diesen Algorithmus habe ich einmal gebraucht, als ich die Aufgabe 2 des 28. Bundeswettbewerb Informatik gelöst habe. Dort ging es darum, die Belegung der Tasten eines Handys zu optimieren (vgl. nächste Seite).

Da die Buchstaben auf den Tasten alphabetisch angeordnet sind, ist eine Belegung der Tasten eindeutig definiert, wenn man die ersten Buchstaben auf den Tasten bestimmt. Damit ist es also nötig, um alle möglichen Belegungen zu bestimmen, alle 8-elementigen Teilmengen der Menge M={a,b,c,...z} zu berechnen. Dazu habe ich die komplette Potenzmenge berechnet und sie so gefiltert, dass sie nur 8-elementige Teilmengen ausgibt. Mit diesen Daten konnte man dann weiter rechnen und die bzgl. der Häufigkeitsverteilung der Buchstaben in einer Sprache optimale Belegung der Tastatur für eine Sprache berechnen.

Aufgabe 2: Handytasten

Die traditionelle Benutzung von Handytastaturen ist trotz des T9-Systems nicht ausgestorben: Um den n-ten Buchstaben einer Taste zu erreichen, muss die Taste n-mal gedrückt werden. Leider wurde die Zuweisung von Buchstaben zu Tasten (die Tastenbelegung, s. Bild unten) nicht optimiert; manche häufige Buchstaben, wie z.B. das 's', sind nur mit mehrmaligem Tastendrücken erreichbar.

1	2 ABC	3 DEF
4 GHI	5 JKL	6 MNO
7 PQRS	8 TUV	9 WXYZ
*	0 ␣	#

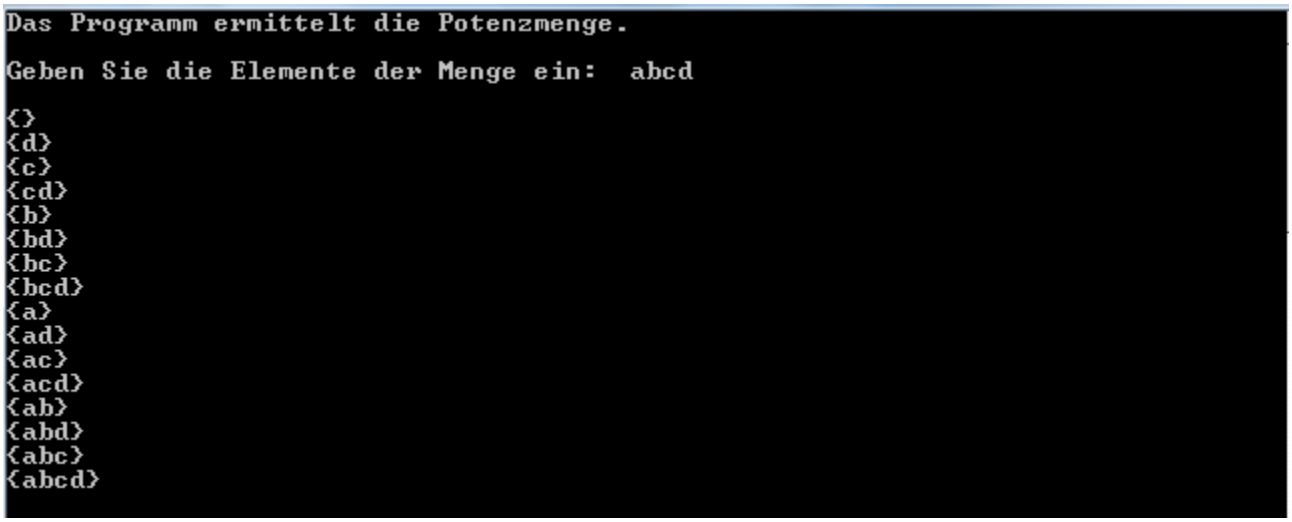
1	2 AB	3 CD
4 EFG	5 HIJK	6 LM
7 NOPQ	8 RS	9 TUV WXYZ
*	0 ␣	#

Bei einigen modernen Mobiltelefonen wird die Tastatur nur noch auf dem Bildschirm dargestellt, und sie wird durch Berührung des Bildschirms bedient. In der entsprechenden Software lässt sich die Tastenbelegung leicht anpassen. Damit wird es sinnvoll, für jede Sprache mit ihren unterschiedlichen Buchstabenhäufigkeiten eine Tastenbelegung so zu berechnen, dass die Zahl von Tastendrücken durchschnittlich minimal wird. Dabei sollen die Buchstaben wie üblich auf den Tasten 2 bis 9 in alphabetischer Reihenfolge angeordnet werden. Das rechte Bild zeigt eine optimale Belegung für Englisch.

Das Programm zur Bestimmung der Potenzmenge:

```
class Potenzmenge {

  static void pot(String p, String s) {
    if (s.length()==0) {
      if (p.equals(""))
        System.out.println("{}");
      else
        System.out.println("{"+p+"}");
    }else{
     pot(p,s.substring(1,s.length()));
     pot(p+s.charAt(0),s.substring(1,s.length()));
    }
  }

  public static void main(String[] args) {
    System.out.println("Das Programm ermittelt die Potenzmenge.\n");
    String s = IOTools.readLine("Geben Sie die Elemente der Menge ein: ");

    pot("", s);

  }
}
```

Ausgabe:



Das Programm ist geradezu erschreckend kurz. Dazu ist es aber auch richtig kompliziert zu verstehen. Wir werden an dieser Stelle darauf verzichten es näher zu untersuchen. Wer möchte, kann gerne mal einen Vortrag über die Arbeitsweise des Programms im Unterricht halten.