Übung 6

Rekursionen

Benötigt: Phanoi.m
Befehle:

Aufgabe 1: Einstufige Rekursion in tiefer Realisierung

- öffne neue Funktion „summe.m“
- Erzeuge eine Funktion zur Ermittlung der Summe der ersten n natürlichen Zahlen (n>=0) nach der Rekursion $summe(n) = \begin{cases} summe(n-1)+n & , \ n > 0 \\ 0 & , \ n = 0 \end{cases}$ durch eine tiefe Realisierung.
- Teste durch die Aufrufe und vergleiche die Reaktionen:

```
>> summe(0)
ans = 0
>> summe(1)
ans = 1
>> summe(2)
ans = 3
>> summe(200)
ans = 20100
>>
```

Aufgabe 2: Einstufige Rekursion in flacher Realisierung

- öffne neue Funktion „summef.m“
- Erzeuge eine Funktion zur Ermittlung der Summe der ersten n natürlichen Zahlen (n>=0) nach der Rekursion $summe(n) = \begin{cases} summe(n-1)+n & , \ n > 0 \\ 0 & , \ n = 0 \end{cases}$ durch eine flache Realisierung.
- Teste durch die Aufrufe und vergleiche die Reaktionen:

```
>> summef(0)
ans = 0
>> summef(1)
ans = 1
>> summef(2)
ans = 3
>> summef(200)
ans = 20100
>> summef(2000)
ans = 2001000
>>
```

Aufgabe 3: Die Türme von Hanoi (einfach)


Die Türme von Hanoi

- Definition: „Die Türme von Hanoi“
  Das Spiel besteht aus drei Stäben A, B und C, auf die mehrere gelochte Scheiben gelegt werden, alle verschieden groß. Zu Beginn liegen alle Scheiben auf Stab A, der Größe nach geordnet, mit der größten Scheibe unten und der kleinsten oben. Ziel des Spiels ist es, den kompletten Scheiben-Stapel von A nach C zu versetzen.
  Bei jedem Zug darf die oberste Scheibe eines beliebigen Stabes auf einen der beiden anderen Stäbe gelegt werden, vorausgesetzt, dort liegt nicht schon eine kleinere Scheibe. Folglich sind zu jedem Zeitpunkt des Spieles die Scheiben auf jedem Feld der Größe nach geordnet.
- öffne die Funktion „bewege.m“
- Implementiere den rekursiven Randoff'schen Algorithmus entprechend dem Pseudo-Code:

```
funktion bewege (Zahl i, Stab a, Stab b, Stab c) {
   falls (i>0) {
      bewege(i-1, a, c, b)
      Ausgabe: verschiebe oberste Scheibe von a nach c
      bewege(i-1, b, a, c)
   }
}
```

  Der Algorithmus besteht im Wesentlichen aus einer Funktion **bewege**, die vier Parameter besitzt. Mit **i** ist die Anzahl der zu verschiebenden Scheiben bezeichnet, mit **a** der Stab von dem verschoben werden soll, mit **b** der Stab, der als Zwischenziel dient und mit **c** der Stab, auf den die Scheiben verschoben werden sollen.
- Teste durch den Aufruf und vergleiche die Reaktion:

```
>> bewege(3,1,2,3)
verschiebe oberste Scheibe von Stab 1 nach Stab 3
verschiebe oberste Scheibe von Stab 1 nach Stab 2
verschiebe oberste Scheibe von Stab 3 nach Stab 2
verschiebe oberste Scheibe von Stab 1 nach Stab 3
verschiebe oberste Scheibe von Stab 2 nach Stab 1
verschiebe oberste Scheibe von Stab 2 nach Stab 3
verschiebe oberste Scheibe von Stab 1 nach Stab 3
>>
```

Aufgabe 4: Die Türme von Hanoi (fortgeschritten)

- speichere (save as) die Funktion „bewege.m“ unter dem Namen „bewege2.m“
- ersetze die Ausgabezeile (fprintf....) durch den Aufruf: **`Phanoi(a,c)`**
- ändere die beien Aufrufe von „bewege“ in „bewege2“ um
- Gebe im Kommandofenster zunächst **>> Phanoi(3)** ein, damit die Startsituation mit 3 Scheiben festgelegt wird.
- Teste durch den Aufruf und vergleiche die Reaktion:

```
>> Phanoi(3)
     1     0     0
     2     0     0
     3     0     0
>> bewege2(3,1,2,3)

     0     0     0
     2     0     0
     3     0     1

     0     0     0
     0     0     0
     3     2     1

     0     0     0
     0     1     0
     3     2     0

     0     0     0
     0     1     0
     0     2     3

     0     0     0
     0     0     0
     1     2     3

     0     0     0
     0     0     2
     1     0     3

     0     0     1
     0     0     2
     0     0     3
>>
```

Aufgabe 5: Die Türme von Hanoi (Zählung)

- Ermittle den Zusammenhang zwischen der Anzahl der Scheiben N und den benötigten Umschichtungsschritten.
- Hierzu gebe man im Kommandofenster für verschiedene Scheibenanzahlen N jeweils: **>> bewege(N,1,2,3)** ein und zähle die ausgegebenen Umschichtungen.

N	**Häufigkeit H**
1	1
2	
3	
4	
5	
6	

- Kann man aus den Resultaten eine Gesetzmäßigkeit erkennen ?

Grundlagen - Rekursionen

Auszug aus: Billmann, L.: Systemtheorie. 1st Edition ISBN 978-1-291-46832-8,
Lulu Press Morrisville USA, 2013

0.1. Rekursion

Eine weitere, häufig vorkommende Form der mathematischen Behandlung, stellt die so genannte *Rekursion* dar. Sie ähnelt der zuvor beschriebenen Iteration, ist aber im Gegensatz dazu keine reine Berechnungs- oder Vorgehens-Vorschrift, sondern stellt eine Form der Problem- oder Aufgabenstellung dar.

In diesem Sinne verstehen wir unter einer rekursiven Aufgabenstellung eine Problem, dessen Lösung auf das gleiche Problem geringerer Komplexität verweist. Als Formel geschrieben

$$x_n = f(x_{n-1}) \quad , \tag{0.1.1}$$

erinnert dies stark an die Fixpunktgleichung Gl. (3.7) einer Iteration. Allerdings ist der gravierende Unterschied darin zu sehen, dass wir zur Lösung des Problems nur endlich viele Rekursionen benötigen, um das Problem auf eine bekannte Vorgabe $\mathbf{x_0}$ zurück zu führen. Eine Erhöhung von **n** ist folglich mit keiner erhöhten Genauigkeit verbunden, wie dies bei der Iteration der Fall war. Das Problem, und nicht die Genauigkeit, legen die notwendige Anzahl **n** der Rekursionen fest.

Zur Verdeutlichung der Vorgehensweise in [1] betrachten wir die Bestimmung der *Fakultät* einer ganzen Zahl n, mit der rekursiven Definition

$$n! = \left\{ \begin{array}{c|c} 1 & n=0 \\ n(n-1)! & n>0 \end{array} \right\} \tag{0.1.2}$$

Wir erkennen einerseits die Vorgabe bzw. Festlegung von Null-Fakultät zu Eins und andererseits die Rückführung der Fakultät von n auf eine Instanz der einfacheren Fakultät (n-1).

Eine Implementierung als M-File könnte dann analog zu Gl. (0.1.2) wie folgt aussehen.

```
function p = nf_d(n)
% Deep recursion form to calculate n factorial
fprintf('call: nf(%d)\n',n)
if (n<1)
    p = 1;
else
    p = n * nf_d(n-1);
end
fprintf('return: nf(%d)=%d\n',n,p)
```

Man erkennt sehr anschaulich das typische Merkmal einer rekursiven Implementierung, das sich selbst Aufrufen der Funktion „**nf_d(n-1)**“. Diese sehr elegant und kompakt wirkende Form hat aber zur Folge, das durch jeden Aufruf der Funktion selbst, eine neue Instanz erzeugt wird und diese im Stack verwaltet werden muss. Wir sprechen daher auch von einer *tiefen Lösung*. Häufige Folge hieraus ist der sog. „Stack-Overflow“ als typischer Fehler bei rekursiven Vorgehensweisen.

Etwas weniger auffällig ist der zweite wichtige Bestandteil dieser Implementierung die sog. Abbruchbedingung durch die bekannte Vorgabe (0!:=1). Sollte man diese vergessen, so ist ein Stack-Overflow natürlich unausweichlich.

Der Programmablauf lässt sich durch die „Print“-Anweisungen am Anfang und Ende der Funktion leicht nachvollziehen. Für die Bestimmung von 6-Fakultät erhalten wir folgende Ausgabe.

```
>> nf_d(6)
call: nf(6)
call: nf(5)
call: nf(4)
call: nf(3)
call: nf(2)
call: nf(1)
call: nf(0)
return: nf(0)=1
return: nf(1)=1
return: nf(2)=2
return: nf(3)=6
return: nf(4)=24
return: nf(5)=120
return: nf(6)=720

ans =

   720
>>
```

Die Ausgabe zeigt uns, dass die Funktion sich zunächst ständig selbst aufruft, bis das Problem auf die Abbruchbedingung bzw. die Vorgabe von 0-Fakultät stößt und anschließend die Berechnung auf dem Rückweg Stück für Stück ermittelt wird. Die Rekursionstiefe oder auch Schachtelungstiefe genannt betrug in unserem Beispiel 6 und kann allgemein mit **n** angegeben werden. Es empfiehlt sich also hier bereits eine Begrenzung bei der Vorgabe von n vorzusehen, um unliebsame Überraschungen während der Bearbeitung zu vermeiden.

Das Stack-Problem lässt sich umgehen, wenn man die Rekursion nicht durch rekursive Funktionsaufrufe realisiert, sondern durch Schleifen eine wiederholte Abarbeitung erreicht. Wir bezeichnen diese Form dann entsprechend als *flache Lösung*, wie folgendes M-File zeigt.

```
function p = nf_f(n)
% Flat recursion form to calculate n factorial
p = 1;
if n > 1                % may be discarded
  for i=2:n
    p = i * p;
  end
end                     % may be discarded
```

Die dargestellte Implementierung sieht zwar nicht mehr so elegant aus, hat aber eine *Rekursionstiefe* von 0 und damit keine Stack-Probleme mehr. Sie liefert nicht nur das gleiche Ergebnis, sondern dieses auch noch bedeutend schneller, da sich die Verwaltung auf die Bereitstellung von **p** beschränkt.

Betrachten wir ein weiteres Beispiel einer *Rekursion* zur Bestimmung des größten gemeinsamen Teilers zweier ganzer Zahlen, wie es in sehr kompakter Form von Dijkstra[1] formuliert und bewiesen wurde:

$$gcd(m,n)=\begin{cases} m & | \quad m=n \\ gcd(m-n,n) & | \quad m>n \\ gcd(n-m,m) & | \quad m<n \end{cases} \tag{0.1.3}$$

Eine Implementierung als tiefe Rekursion könnte entsprechend nachfolgendem M-File erfolgen.

```
function ret = DijkstraGCD_d(m, n)
% Deep recursion form for greatest common divisor of m, n
fprintf('call: gcd(%d,%d)\n',m,n)
if (m == n)
    ret = m;
elseif ( m > n )
    ret = DijkstraGCD_d(m-n, n);
else
    ret = DijkstraGCD_d(m, n-m);
end
fprintf('return: gcd(%d,%d)=%d\n',m,n,ret)
```

Der Programmablauf lässt sich wiederum durch die „Print“-Zeilen verdeutlichen. Man erkennt den Abstieg bis zur Rekursionstiefe von 6 für das gewählte Zahlenbeispiel. Nachdem die Lösung gefunden wurde, wird der Rückweg ohne weitere Berechnungen beschritten. Eine allgemeine Angabe zur Rekursionstiefe ist hier nicht so einfach möglich.

1) Edsger Wybe Dijkstra, 1930-2002, Niederländischer Mathematiker und Computerwissenschaftler mit mehr als 1300 Veröffentlichungen, der beispielsweise den ersten Algol 60 Compiler programmierte.


```
>> DijkstraGCD_d(54,15)
call: gcd(54,15)
call: gcd(39,15)
call: gcd(24,15)
call: gcd(9,15)
call: gcd(9,6)
call: gcd(3,6)
call: gcd(3,3)
return: gcd(3,3)=3
return: gcd(3,6)=3
return: gcd(9,6)=3
return: gcd(9,15)=3
return: gcd(24,15)=3
return: gcd(39,15)=3
return: gcd(54,15)=3

ans =

     3
```

Eine Implementierung als flache Rekursion könnte entsprechend nachfolgendem M-File erfolgen. Man sollte die beiden Implementierungen ruhig für verschiedene Zahlenwerte ausprobieren, um die erheblichen Zeitunterschiede bei der Aufgabenbewältigung zu erfahren. Hierzu sind aber die „Print“-Anweisungen zu entfernen, um einen fairen Vergleich zu ermöglichen.

```
function ret = DijkstraGCD_f(m, n)
% Flat recursion form for greatest common divisor of m, n
while (m ~= n)
    if ( m > n )
        m = m-n;
    else
        n = n-m;
    end
end
ret = m;
```

Beiden Beispielen gemeinsam ist, dass sie entsprechend Gl. (0.1.1) beschrieben werden können und deswegen auch Rekursionen 1. Ordnung genannt werden.

Zusammenfassend können wir sagen, dass *Rekursionen* eine sehr kompakte Beschreibung darstellen, die nicht nur für mathematische Problemstellungen von großer Bedeutung sind. Besonders vorteilhaft ist der Einsatz von Such-Algorithmen zur Lösung von Fragestellungen bei Optimierungen, Sortierungen und Komprimierungen oder bei Strategien für Spiele und autonome Systeme. Der Vorteil liegt hier ganz eindeutig darin, dass es stets einen soliden Rückweg zur Ausgangslage gibt, was dem Suchen nach einer Lösung sehr nahe kommt.

Die Frage der Implementierung als flache oder tiefe Rekursion ist eigentlich eher zweitrangig und mehr durch die Komplexität der Aufgabenstellung bzw. die vorhandenen Ressourcen bestimmt.