Übung 12

a) Die Funktion *f(n)* für die rekursive Berechnung der Fakultät gemäß $n! = n * (n-1)!$, mit $0! = 1$, benutzt einen Call Stack. Zeigen Sie unten, wie sich dieser für die Berechnung von *f(3)* auf- und wieder abbaut.

	f(2) = _____					
f(3) = 3*f(2)	f(3) = 3*f(2)					

b) Die folgenden Funktionen sollen in C *rekursiv* implementiert werden:

Summationsfkt.: $\sum n = 1 + 2 + \ldots + (n-1) + n$

```
int sum(int n);          /* Summenbildung von 1 bis n */
```

Fakultät: $n! = n * (n-1) * \ldots * 1$

```
int fakultaet(int n);    /* Fakultätsberechnung für n */
```

Fibonacci-Zahlen: $f_0 = 0, f_1 = 1, f_n = f_{n-1} + f_{n-2}$

```
int fibonacci(int n);    /* Fibonacci-Zahl von n */
```

c) Die sog. Ackermannfunktion ist für natürliche Zahlen *n*, *m* wie folgt rekursiv definiert:

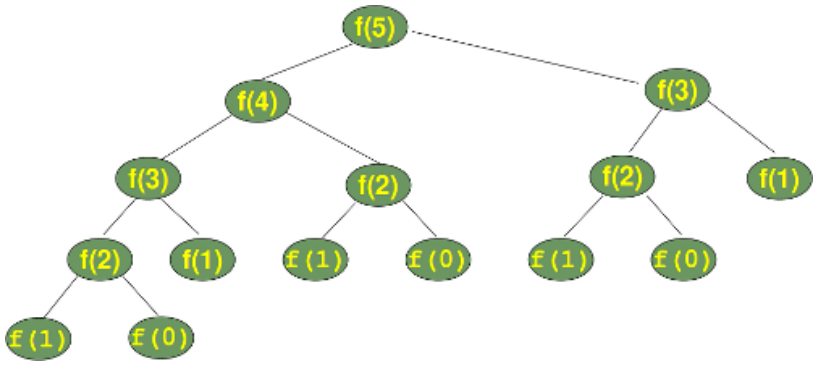
```
ackermann(0, m) = m + 1                                  ∀ m ≥ 0
ackermann(n, 0) = ackermann(n – 1, 1)                    ∀ n ≥ 1
ackermann(n, m) = ackermann(n – 1, ackermann(n, m – 1))  ∀ n, m ≥ 1
```

Schreiben Sie in C eine rekursive Funktion, welche für (sehr kleine!) natürliche Zahlen *n*, *m* die Ackermannfunktion berechnet (Zahlenbeispiel: *ackermann*(0, 1) = 2).

d) Implementieren Sie eine abgeänderte Variante der Fibonacci-Funktion aus Aufgabenteil *b)*, indem Sie basierend auf dem vorigen Aufgabenteil nun noch geeignet die Ablage in ein Array einfügen, das in *main()* deklariert wurde. Die Signatur soll wie folgt aussehen:

```
int fibonacciZwischenspeichern(int n, int array[]);
```

Unter Nutzung des zusätzlichen Arrays soll die Performance verbessert werden, indem Werte, die in einem vorangegangenen Rekursionsschritt schon berechnet wurden, bei erneutem Aufruf der Funktion wiederverwendet werden (das nachfolgende Bild dient zur Inspiration: wie man sieht, wird bei naivem Vorgehen *f(2)* unnötigerweise leider gleich dreimal berechnet). Geben Sie anschließend zum Testen alle Feldwerte aus.



e) Schreiben Sie jeweils Funktionen, die als Eingabe ein Integer-Array bekommen und folgendes zurückgeben (ggf. Hilfsfunktionen verwenden):

- *minimumRek* – rekursive Berechnung des Minimums aller Zahlen
- *minimumIter* – iterative Berechnung des Minimums aller Zahlen
- *produktRek* – rekursive Berechnung des Produkts aller Zahlen
- *produktIter* – iterative Berechnung des Produkts aller Zahlen

f) Wenn man mit Sternchen (*) ein Dreieck auf der Konsole ausgibt, dann hat die erste Zeile einen Stern (*), die zweite zwei (**), die dritte Zeile drei (***) usw. Schreiben Sie eine Funktion *int triangle(int rows)*, welche als Ergebnis die Anzahl aller Sterne zurückliefert, die ein Dreieck mit der übergebenen Anzahl an Zeilen hat. Sie dürfen keine Multiplikation oder Schleifen benutzen, stattdessen ist diese Aufgabe (wie alle hier) rekursiv zu lösen.

g) Implementieren Sie eine rekursive Funktion *int power(int b, int n)*, die b^n berechnet (z.B. power(3, 2) ist 9). Verwenden Sie keine Schleifen oder Funktionen aus *<math.h>*.

h) Mehrere Hasen stehen durchnummeriert in einer Reihe. Die ungerade nummerierten Hasen (1, 3, ...) haben zwei Ohren, die gerade nummerierten (2, 4, ...) haben drei Ohren (weil sie zu dicht am AKW wohnen). Schreiben Sie eine Funktion *int bunnyEars(int numBunnies)*, welche für die übergebene Anzahl an Hasen in einer Reihe die Gesamtzahl an Ohren zurückgibt. Nutzen Sie wieder Rekursion, aber keine Schleifen oder Multiplikation.

i) Das folgende Problem tritt bei der Verarbeitung strukturierter Strings oft auf: Sie sollen beim Einlesen feststellen, ob die Syntax korrekt geklammert ist, also jede öffnende Klammer auch eine schließende hat (und umgekehrt).
Schreiben Sie dazu eine Funktion *bool nestParenthesis (char str[], int firstChar, int lastChar)*, die feststellt ob die Klammerung im String *str* korrekt ist; d.h. es handelt sich um eine Verschachtelung von null oder mehr Paaren von Klammern, z.B. „(())“. Geben Sie als Ergebnis *true* zurück, wenn die übergebene Klammerung richtig verschachtelt ist, andernfalls *false*.
Tipp: Prüfen Sie in Ihrem Algorithmus je die ersten und letzten Zeichen des Strings um festzustellen, ob zwei Klammern zusammenpassen, und lösen Sie so das Problem rekursiv. Die Argumente *firstChar* und *lastChar* definieren dabei jeweils den Index des ersten bzw. letzten Zeichens der betrachteten Teil-Zeichenkette. Hier einige Beispiele:

"((me my))" → *true*, "(?(()))" → *true*, "(((x + y))" → *false*