Arbeitsauftrag GK inf12 für 08.01.2021

Ergänzungen und Vertiefungen zum Konzept "Rekursion"

Die funktionale Formulierung eines Algorithmus bedient sich der mathematischen Struktur eines Problems; wesentliche Kontrollstruktur ist die Rekursion *(Bekanntlich heißt eine Funktion oder eine Prozedur (Teilprogramm) rekursiv, wenn sie mindestens einen Aufruf ihrer selbst enthält.)*.

In höheren Programmiersprachen (Pascal, C++, Python) ist die Möglichkeit der rekursiven Formulierung implementiert, was häufig eine sehr elegante Formulierung eines Algorithmus gestattet.

Bei rekursiven Programmen kann es jedoch zu einem „stack overflow" kommen, wenn die Anzahl der gleichzeitig aktiven Aufrufe der Prozedur oder der Funktion zu groß wird. In Python ist eine Rekursionstiefe von 1000 (oder 1024?) voreingestellt; nach Import des **sys-Moduls** mittels

```
import sys
```

läßt sich über

```
sys.getrecursionlimit()
```

die aktuelle Rekursionstiefe ausgeben, und mit

```
sys.setrecursionlimit(a)
```

kann man die Rekursionstiefe auf den Wert a setzen.

Wir greifen noch mal die Funktion fact (siehe Beispiel 3 aus Arbeitsauftrag für 24.11.2020) auf, die jeder natürlichen Zahl $n \in \{0, 1, 2, \ldots\}$ deren Falkultät fact(n) zuordnet.

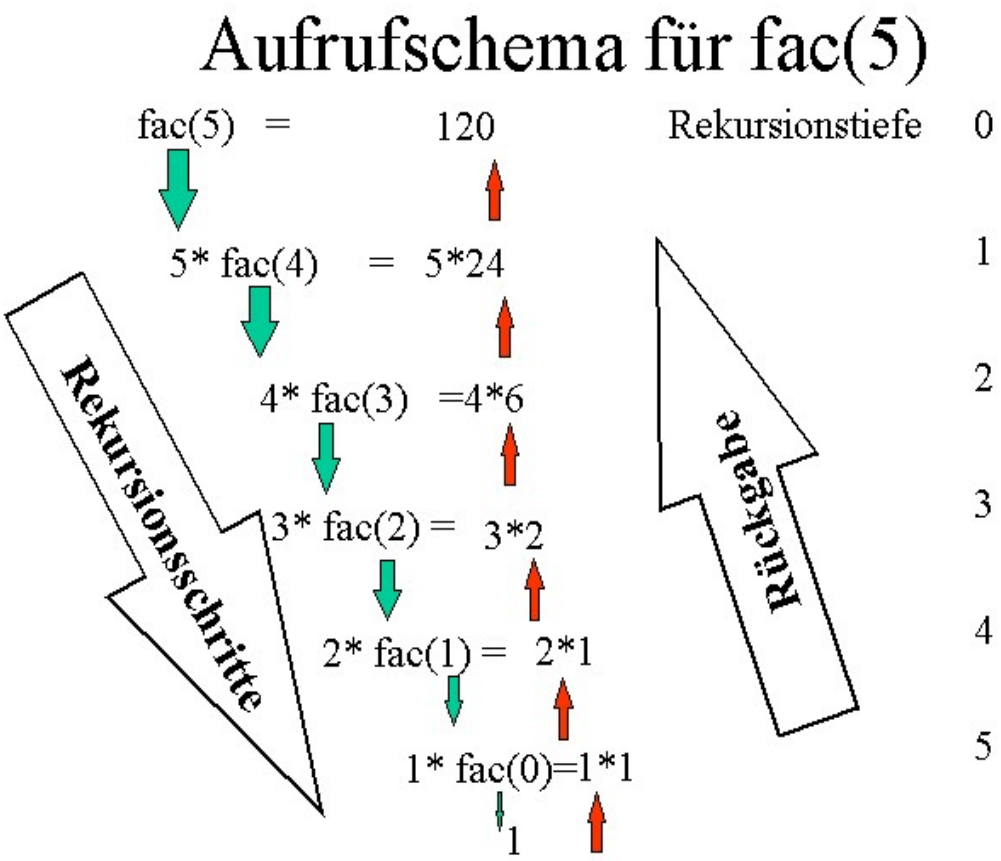
Python-Quelltext:

```
"factorial recursive"

import sys

print('Rekursionstiefe: ',sys.getrecursionlimit())
a=int(input('gewuenschte Rekursionstiefe: '))
sys.setrecursionlimit(a)
print('aktuelle Rekursionstiefe: ',sys.getrecursionlimit())
print()

def fact(x):
    global z
    z = z + 1
    if x == 0:
       return 1
    else:
       return x * fact(x - 1)

z = 0
n=int(input("n =  "))
y = fact(n)
print (n,"! = ",y)
print('Anzahl der Aufrufe: ',z)
```

Beachte:
Die Variablen **n**, **z** und **y** sind globale Variable; bei Aufruf **fact(n)** in der drittletzten Zeile wird der Wert der Variablen **n** an die lokale Variable **x** der Funktion **fact** übergeben (selbst wenn man innerhalb der Funktion **fact** die lokale Variable **x** mit **n** bezeichnen würde, würde für dieses **n** ein eigener lokaler Speicherplatz definiert).
Die globale Variable **z** zählt die Anzahl der Aufrufe der Funktion **fact**. Vor dem ersten Aufruf von **fact** wird **z** auf 0 gesetzt (initialisiert), und bei jeder Abarbeitung der Funktion **fact** wird **z** um 1 erhöht (inkrementiert). Die Anweisung **global z** verhindert, daß **z** innerhalb der Funktion als neue lokale Variable verstanden wird.

Aufrufschema für fact(5) (nach http://www.saar.de/~awa/jrekursion.html):

Der grüne Pfeil bedeutet jeweils „ruft auf", der rote „gibt zurück"; der Rekursionsanfang fact(0)=1 erzwingt, daß der Algorithmus abbricht (terminiert).



Aufgabe:
Erstelle zur Berechnung der **Hofstadter-Funktion** und der **Ackermann-Funktion** (Aufgabenblatt Nr. 2 vom 10.11.2020) jeweils einen Python-Programmtext, erweitert um die Möglichkeit, die Rekursionstiefe anzupassen und die Anzahl z der Aufrufe zu zählen und auszugeben; teste die Programme mit unterschiedlichen Werten.

Definition der **Hofstadter-Funktion**, die jeder natürlichen Zahl n≥1 den Wert hof(n) zuordnet:

Rekursionsanfang: hof(1) = 1
hof(2) = 1

Rekursionsvorschrift: $hof(n) = hof[n - hof(n-1)] + hof[n - hof(n-2)]\ ,\ n > 2$

Rekursionstiefe und Anzahl der Funktionsaufrufe
am Beispiel der Hofstadter-Funktion

Quellcode in Python:

```
def hof(x):
    if x == 1:
        return 1
    elif x == 2:
        return 1
    elif x > 2:
        return hof(x - hof(x - 1)) + hof(x - hof(x - 2))

endwert = int(input('Bis zu welchem Wert soll Hofstadter berechnet werden? '))
n = 1
while n <= endwert:
    y = hof(n)
    print('hof(',n,') =',y)
    n += 1
```

h(1) = 1

1 Aufruf Rekursionstiefe = 1

h(2) = 1

1 Aufruf Rekursionstiefe = 1

```
h(3) = h(3 - h(2)) + h(3 - h(1))      ←  2-fache Verschachtelung
 1           1             1          ←  # Aufrufe
     = h(3 - 1) + h(3 - 1)
     = h(2) + h(2)
        1       1                     ←  # Aufrufe
     = 1  +  1
     = 2
```

5 Aufrufe Rekursionstiefe = 2

```
h(4) = h(4 - h(3)) + h(4 - h(2))
 1           5             1          ←  # Aufrufe
     = h(4 - 2)     + h(3)

     = h(2) + h(3)
         1     5                      ←  # Aufrufe
```

13 Aufrufe insgesamt

```
     = h(4 - h(3))                          + h(4 - h(2))
     = h(4 - (h(3 - h(2)) + h(3 - h(1)))) + h(3)
     = h(4 - (h(3 - h(2)) + h(3 - h(1))))  + h(3 - h(2)) + h(3 - h(1))
       \_____________________________________/
                  3-fache Verschachtelung

     = h(4 - (h(3 - 1) + h(3 - 1)) + h(3 - 1) + h(3 - 1)
     = h(4 - (h(2) + h(2)) + h(2) + h(2)
     = h(4 - (1 + 1)) + 1 + 1
     = h(2) + 1 + 1
     = 1 + 1 + 1
     = 3
```

Rekursionstiefe = 3

08.01.2021

Sortieren durch direkte Auswahl

Wir beschränken uns zunächst darauf, eine Liste von ganzen Zahlen (hier: Zufallszahlen) der Größe nach, und zwar aufsteigend, zu sortieren. Den Algorithmus später auf andere Datenstrukturen (z. B. Namen, Verbundtypen) zu übertragen, ist vergleichsweise einfach und bereitet keine Schwierigkeiten.

Die Python-Anweisungen `range`, `list` und `len`:

a) `range`-Anweisung

Die **`range`**-Anweisung definiert einen Bereich ganzer Zahlen.

`range(10)` definiert den Bereich 0, 1, . . . , 9
`range(4,21)` definiert den Bereich 4, 5, . . . , 20
`range(4,21,3)` definiert den Bereich 4, 7, 10, . . . , 16, 19
`range(-4,3)` definiert den Bereich -4, -3, -2, -1, 0, 1, 2

Allgemein gilt:

`range(start, stop)`
definiert den Bereich **`start`**, ... , **`stop-1`** ganzer Zahlen,

`range(start, stop, step)`
definiert den Bereich **`start`**, ... mit der Schrittweite **`step`**, wobei die Zahl **`stop`** nicht mehr enthalten ist.

b) Erstellen einer Liste ganzer Zahlen

`a = list(range(4,13))` erzeugt die Liste
`[4, 5, 6, 7, 8, 9, 10, 11, 12]`;
die (in diesem Fall 9) Elemente dieser Liste nennen wir auch Komponenten, auf die man mit **a[0], a[1], . . . , a[8]** zugreifen kann (mit Erzeugung der Liste dieses Beispiels sind die Komponenten **a[0], a[1], . . . , a[8]** in dieser Reihenfolge mit den Werten **4, 5, ... , 12** belegt). Allerdings läßt sich jeder Komponente **a[i]** eine beliebige andere ganze Zahl zuweisen.

*Bemerkung: Unter einem **Feld** oder **array** verstehen wir eine Folge von Variablen gleichen Typs; mit der Anweisung* **`a = list(range(4,13))`** *haben wir also ein array **a** erzeugt mit den Komponenten a[0], a[1], . . . , a[8].*

c) **`len(a)`** bestimmt die Anzahl der Komponenten der Liste **a**, in dem Beispiel aus b) gilt somit: **`len(a) = 9`** .

1. Erstellen einer Liste mit n Komponenten, denen Zufallszahlen zugewiesen werden (n ist eine natürliche Zahl)

Vorbemerkung:
Die Python-Anweisung **`randint`** ist eine vordefinierte Funktion des random-Moduls in Python; Syntax: **`randint(r,s)`** mit ganzen Zahlen r und s, $r \leq s$, erzeugt eine Zufallszahl aus dem Intervall [r, s].
Beispiele:
`randint(1,1000)` erzeugt eine Zufallszahl aus dem Bereich 1, ... , 1000
`randint(-7,12)` erzeugt eine Zufallszahl aus dem Bereich -7, ... , 12

Ein Algorithmus, der nach Eingabe einer natürlichen Zahl **n** eine Liste aus **n** Zufallszahlen generiert, formuliert als Python-Quelltext in der Schriftart **`Courier New`**, so daß man den Quelltext unmittelbar durch *copy* und *paste* in einen Editor für Python-Programme übernehmen kann:

```
# array mit zufallszahlen

from random import randint

n = int(input('Laenge des arrays = '))
print()

# Erzeugen des arrays mit dem Namen a
# und den n Komponenten a[0], . . . , a[n-1]
a = list(range(1,n+1))

# Zuweisung von Zufallszahlen an die Komponenten des arrays a
for i in range(0,n):
    a[i] = randint(1,1000)

# Ausgabe des arrays
for i in range(0,n):
    print(a[i])
```

2. Bestimmung des kleinsten Elements der Liste aus n Komponenten

Der Inhalt des Speicherplatzes **a[0]** wird sukzessive mit den Inhalten von **a[1], . . . , a[n-1]** verglichen; falls gilt **a[i] < a[0]**, $1 \leq i \leq n-1$, werden die Inhalte der Speicherplätze **a[i]** und **a[0]** ausgetauscht; hierzu wird, bevor **a[0]** den Wert von **a[i]** erhält, der ursprüngliche Wert von **a[0]** mittels der Hilfsvariablen **temp** gesichert und nach der Zuweisung **a[0] = a[i]** mit **a[i] = temp** an **a[i]** übergeben.

Die Durchführung der Vergleiche und der ggf. erforderliche Austausch der Inhalte von **a[0]** und **a[i]** werden hier an die Funktion **min(x)** delegiert:

```
def min(x):
    for i in range(1,len(x)):
        if x[i] < x[0]:
             temp = x[0]
             x[0] = x[i]
             x[i] = temp
```

Mit dem Aufruf **min(a)** wird die Funktion **min** auf das aus den Komponenten **a[0], . . . , a[n-1]** bestehende array **a** angewendet.

```
from random import randint

n = int(input('Laenge des arrays = '))
print()

# Erzeugen des arrays mit dem Namen a
# und den n Komponenten a[0], . . . , a[n-1]
a = list(range(1,n+1))

# Zuweisung von Zufallszahlen an die Komponenten des arrays a
for i in range(0,n):
    a[i] = randint(1,1000)
```

```
# Ausgabe des arrays
for i in range(0,n):
    print(a[i])

# Bestimmen des kleinsten Elements:
# Wir definieren eine Funktion min(x), die auf
# das array a angewendet wird, das kleinste Element
# bestimmt und dieses der Komponente a[0] zuweist.

def min(x):
    for i in range(1,len(x)):
        if x[i] < x[0]:
             temp = x[0]
             x[0] = x[i]
             x[i] = temp

# Aufruf der auf das array a anzuwendenden Funktion min

min(a)

# Ausgabe der Liste mit dem kleinsten Element an der 1. Stelle
print()
for i in range(0,n):
    print(a[i])
```

Nachdem das kleinste Element der Liste **a[0], . . . , a[n-1]** dem Speicherplatz **a[0]** zugewiesen wurde, bestimmen wir das kleinste Element der „Restliste" **a[1], . . . , a[n-1]** und weisen es dem Speicherplatz **a[1]** zu.

Wenn wir dieses Verfahren sukzessive auf die weiteren „Restlisten" **a[j], . . . , a[n-1]** mit **2 ≤ j ≤ n-2** anwenden, erhalten wir ein array **a**, dessen Komponenten gemäß **a[0] ≤ a[1] ≤ . . . ≤ a[n-1]** aufsteigend sortiert sind.

Wir modifizieren die Funktion **min(x)**, indem wir einen weiteren Parameter **j** ergänzen:

```
def min(x,j):
    for i in range(j+1,len(x)):
        if x[i] < x[j]:
             temp = x[j]
             x[j] = x[i]
             x[i] = temp
```

Die mit dem Parameterwert **j** auf das array **a** angewendete Funktion **min(x,j)** ermittelt in der Liste **a[j], . . . , a[n-1]** das kleinste Element und weist es dem Speicherplatz **a[j]** zu.

3. Variante zu 2:

```
from random import randint

n = int(input('Laenge des arrays = '))
print()
```

```
# Erzeugen des arrays mit dem Namen a
# und den n Komponenten a[0], . . . , a[n-1]
a = list(range(1,n+1))

# Zuweisung von Zufallszahlen an die Komponenten des arrays a
for i in range(0,n):
    a[i] = randint(1,1000)

# Ausgabe des arrays
for i in range(0,n):
    print(a[i])

# Bestimmen des kleinsten Elements:
# Wir definieren eine Funktion min(x,j), die auf
# die Komponenten a[j], . . , a[n-1] des arrays a
# angewendet wird, das kleinste Element
# bestimmt und dieses der Komponente a[j] zuweist.

def min(x,j):
    for i in range(j+1,len(x)):
        if x[i] < x[j]:
             temp = x[j]
             x[j] = x[i]
             x[i] = temp

# Aufruf der auf das array a anzuwendenden Funktion min

min(a,0)

# Ausgabe der Liste mit dem kleinsten Element an der 1. Stelle
print()
for i in range(0,n):
    print(a[i])
```

4. Bestimmung der 2 kleinsten Elemente der Liste aus n Komponenten

```
from random import randint

n = int(input('Laenge des arrays = '))
print()

# Erzeugen des arrays mit dem Namen a
# und den n Komponenten a[0], . . . , a[n-1]
a = list(range(1,n+1))

# Zuweisung von Zufallszahlen an die Komponenten des arrays a
for i in range(0,n):
    a[i] = randint(1,1000)

# Ausgabe des arrays
for i in range(0,n):
    print(a[i])

# Wir definieren eine Funktion min(x,j), die auf
```

```
# die Komponenten a[j], . . , a[n-1] des arrays a
# angewendet wird, das kleinste Element
# bestimmt und dieses der Komponente a[j] zuweist.

def min(x,j):
    for i in range(j+1,len(x)):
        if x[i] < x[j]:
             temp = x[j]
             x[j] = x[i]
             x[i] = temp

# Aufrufe der auf das array a anzuwendenden Funktion min

min(a,0)
min(a,1)

# Ausgabe der Liste
print()
for i in range(0,n):
    print(a[i])
```

5. Bestimmung der 3 kleinsten Elemente der Liste aus n Komponenten

```
. . . . . .
. . . . . .

# Aufrufe der auf das array a anzuwendenden Funktion min

min(a,0)
min(a,1)
min(a,2)

. . . . . .
. . . . . .
```

6. Sortieren der aus den Komponenten a[0], ... , a[n-1] bestehenden Liste a

Wir sortieren das array **a** mit den Komponenten **a[0], . . . , a[n-1]**, indem wir die Funktion **min(x,j)** mit j = 0, 1, . . . , n-2 nacheinander auf das array a anwenden; die wiederholte Anwendung realisieren wir mit einer while-Schleife, deren Schleifenindex j mit dem Wert 0 initialisiert wird:

```
j = 0
while j <= n-2:
    min(a,j)
    j +=1
```

Der hier vorgestellte Algorithmus ist unter der Bezeichnung

„Sortieren durch direkte Auswahl"

bekannt.

Der folgende in Python codierte Algorithmus sortiert aufsteigend ein array **a** der Länge **n**, dessen Komponenten **a[0], . . . , a[n-1]** Zufallszahlen aus dem Bereich 1, . . . , 100000 zugewiesen wurden:

```
# sorting by direct selection

from random import randint

n = int(input('Laenge des arrays = '))
print()

# Erzeugen des arrays mit dem Namen a
# und den n Komponenten a[0], . . . , a[n-1]
a = list(range(1,n+1))

# Zuweisung von Zufallszahlen an die Komponenten des arrays a
for i in range(0,n):
    a[i] = randint(1,100000)

# Ausgabe des arrays
for i in range(0,n):
    print(a[i])

# Die auf die Komponenten a[j], . . , a[n-1] des arrays a
# angewendete Funktion min(x,j) bestimmt das kleinste Element
# und weist dieses der Komponente a[j] zu.

def min(x,j):
    for i in range(j+1,len(x)):
        if x[i] < x[j]:
             temp = x[j]
             x[j] = x[i]
             x[i] = temp

# Aufrufe der auf das array a anzuwendenden Funktion min

j = 0
while j <= n-2:
    min(a,j)
    j +=1

# Ausgabe der sortierten Liste

print()
print('Sortierte Liste:')

for i in range(0,n):
    print(a[i])
```

Selbach
update 26.01.2021

SelectionSort mit Ermittlung des Zeitbedarfs zur Laufzeit:

```
# sorting by direct selection
# Nach Eingabe einer natuerlichen Zahl n wird ein
# aus n Komponenten bestehendes array sortiert.

from random import randint
import time

n = int(input('Laenge des arrays: '))
print()

# Erzeugen des arrays mit dem Namen a
# und den n Komponenten a[0], . . . , a[n-1]
a = list(range(1,n+1))

# Zuweisung von Zufallszahlen an die Komponenten des arrays a
for i in range(0,n):
    a[i] = randint(1,1000000)

# Ausgabe des arrays
r = int(input('Wieviele Elemente sollen angezeigt werden? '))
print()
for i in range(0,r):
    print(a[i])

# Die auf die Komponenten a[j], . . , a[n-1] des arrays a
# angewendete Funktion min(x,j) bestimmt das kleinste Element
# und weist dieses der Komponente a[j] zu.

def min(x,j):
    for i in range(j+1,len(x)):
        if x[i] < x[j]:
            temp = x[j]
            x[j] = x[i]
            x[i] = temp

# Aufrufe der auf das array a anzuwendenden Funktion min
# mit Ermittlung des Zeitbedarfs zur Laufzeit

start = time.time()

j = 0
while j <= n-2:
    min(a,j)
    j +=1

end = time.time()


# Ausgabe der sortierten Liste

print()
print('Sortierte Liste:')
print()

for i in range(0,r):
    print(a[i])

print()
print('Zeitaufwand zum Sortieren von',n,'Elementen: {:7.3f} s'.format(end-start))
```

```
# sorting by direct selection
# Nach Eingabe einer natuerlichen Zahl n wird ein
# aus n Komponenten bestehendes array sortiert.

from random import randint
import time

n = int(input('Laenge des arrays: '))
print()

# Erzeugen des arrays mit dem Namen a
# und den n Komponenten a[0], . . . , a[n-1]
a = list(range(1,n+1))

# Zuweisung von Zufallszahlen an die Komponenten des arrays a
for i in range(0,n):
    a[i] = randint(1,1000000)

# Ausgabe des arrays
r = int(input('Wieviele Elemente sollen angezeigt werden? '))
print()
for i in range(0,r):
    print(a[i])

# Die auf die Komponenten a[j], . . , a[n-1] des arrays a
# angewendete Funktion min(x,j) bestimmt das kleinste Element
# und weist dieses der Komponente a[j] zu.

def min(x,j):
    for i in range(j+1,len(x)):
        if x[i] < x[j]:
             temp = x[j]
             x[j] = x[i]
             x[i] = temp

# Aufrufe der auf das array a anzuwendenden Funktion min
# mit Ermittlung des Zeitbedarfs zur Laufzeit

start = time.time()

j = 0
while j <= n-2:
    min(a,j)
    j +=1

end = time.time()


# Ausgabe der sortierten Liste

print()
print('Sortierte Liste:')
print()

for i in range(0,r):
    print(a[i])

print()
print('Zeitaufwand zum Sortieren von',n,'Elementen: {:7.3f}
s'.format(end-start))
```

Aufwandsbetrachtung „Sortieren durch direkte Auswahl"

Wir formulieren einen funktionalen Zusammenhang zwischen dem zeitlichen Aufwand, um eine Liste von n Datenelementen der Größe nach zu sortieren, und der Anzahl n der Datenelemente.

Wertzuweisungen, Abfragen, Rechenoperationen sind elementare Anweisungen, die eine bestimmte Rechenzeit erfordern; obwohl diese Rechenzeiten mit fortschreitender Leistungsfähigkeit der Hardware immer kürzer werden, gerät man rasch an Grenzen der praktischen Durchführbarkeit eines Algorithmus, wenn die Anzahl der abzuarbeitenden Anweisungen zu stark, z. B. exponentiell, wächst.

Wesentlicher Baustein des Algorithmus „Sortieren durch direkte Auswahl" ist die Funktion **`min(x,j)`**, die das kleinste Element des arrays **`a[j], . . , a[n-1]`** ermittelt und dieses der Komponente **`a[j]`** zuweist.

```
def min(x,j):
    for i in range(j+1,len(x)):
        if x[i] < x[j]:
            temp = x[j]
            x[j] = x[i]
            x[i] = temp
```

Der Schleifenrumpf der in der der Funktion **`min(x,j)`** implementierten for-Schleife besteht aus 3 Wertzuweisungen und 1 Abfrage, die wir gedanklich als ganzes zum Anweisungsblock **A** zusammenfassen:

```
def min(x,j):
    for i in range(j+1,len(x)):
                        A
j = 0
while j <= n-2:
    min(a,j)
    j = j+1
```

Wir überlegen, wie oft der Block **A** abgearbeitet wird, indem wir zunächst die Anzahl **`z(j)`** dieser Abarbeitungen in Abhängigkeit vom Schleifenindex **`j`** notieren:

Index j	**Aufruf**	**Index i**	**z(j)**
j = 0	min(x,0)	1 ≤ i ≤ n-1	n-1
j = 1	min(x,1)	2 ≤ i ≤ n-1	n-2
j = 2	min(x,2)	3 ≤ i ≤ n-1	n-3
j = 3	min(x,3)	4 ≤ i ≤ n-1	n-4
....	....	....	....
j = n-2	min(x,n-2)	n-1 ≤ i ≤ n-1	1

Gesamtzahl z der Abarbeitungen von Anweisungsblock **A**:

$$z = z(0) + z(1) + z(2) + \ldots + z(n-2) = (n-1) + (n-2) + (n-3) + \ldots + 1$$

$$= \sum_{k=1}^{n-1} k$$

$$= \tfrac{1}{2} \cdot (n-1) \cdot n \qquad \textit{(vgl. Anmerkung)}$$

$$= \tfrac{1}{2} \cdot (n^2 - n)$$

$$\approx \tfrac{1}{2} \cdot n^2 \qquad \text{für große n}$$

Ergebnis:
Die Anzahl der abzuarbeitenden elementaren Anweisungen und damit der Zeitaufwand wachsen quadratisch mit der Anzahl n der zu sortierenden Datensätze.

Anmerkung: $\sum_{k=1}^{n} k = \frac{1}{2} \cdot n \cdot (n+1)$

19.01.2021

Varianten des Python-Quellcodes zum Algorithmus „Sortieren durch direkte Auswahl“

Informatik 12

Hinweis:
Python-Programmcode ist in der Schriftart **Courier New** gesetzt; der Code läßt sich einfach mit copy and paste in einen Editor für Python-Programmtexte übernehmen.

```
# sorting by direct selection
# Ein aus n Komponenten bestehendes array wird sortiert.

from random import randint
import time

n = int(input('Laenge des arrays: '))
print()

# Erzeugen des arrays mit dem Namen a
# und den n Komponenten a[0], . . . , a[n-1]
a = list(range(1,n+1))

# Zuweisung von Zufallszahlen an die Komponenten des arrays a
for i in range(0,n):
    a[i] = randint(1,1000000)

# Ausgabe des arrays
r = int(input('Wieviele Elemente sollen angezeigt werden? '))
print()
for i in range(0,r):
    print(a[i])

# Die auf die Komponenten a[j], . . , a[n-1] des arrays a
# angewendete Funktion min(x,j) bestimmt das kleinste Element
# und weist dieses der Komponente a[j] zu.

def min(x,j):
    for i in range(j+1,len(x)):
        if x[i] < x[j]:
            temp = x[j]
            x[j] = x[i]
            x[i] = temp

# Aufrufe der auf das array a anzuwendenden Funktion min
# mit Ermittlung des Zeitbedarfs zur Laufzeit

start = time.time()

j = 0
while j <= n-2:
    min(a,j)
```

```
    j +=1

end = time.time()


# Ausgabe der sortierten Liste

print()
print('Sortierte Liste:')
print()

for i in range(0,r):
    print(a[i])

print()
print('Zeitaufwand zum Sortieren von',n,'Elementen: {:7.3f}
s'.format(end-start))
```

Der rot markierte Programmcode umfaßt die Definition der Funktion min(x,j) und deren wiederholte Aufrufe innerhalb der while-Schleife.

Diese Formulierung mit einer Funktion min(x,j) ist zweckmäßig, um den Aufwand und damit das Wachstum der Rechenzeit in Abhängigkeit von der Anzahl n der zu sortierenden Datenelemente zu ermitteln.

Wir können **min(x,j)** und die **while-Schleife** zu einer neuen Funktion **sort(x)** zusammenfassen:

```
# sorting by direct selection
# Ein aus n Komponenten bestehendes array wird sortiert.

from random import randint
import time

n = int(input('Laenge des arrays: '))
print()

# Erzeugen des arrays mit dem Namen a
# und den n Komponenten a[0], . . . , a[n-1]
a = list(range(1,n+1))

# Zuweisung von Zufallszahlen an die Komponenten des arrays a
for i in range(0,n):
    a[i] = randint(1,1000000)

# Ausgabe des zu sortierenden arrays
r = int(input('Wieviele Elemente sollen angezeigt werden? '))
print()
```

```
for i in range(0,r):
    print(a[i])


# Definition der Funktion sort

def sort(x):
    j = 0
    while j <= n-2:
        for i in range(j+1,len(x)):
            if x[i] < x[j]:
                temp = x[j]
                x[j] = x[i]
                x[i] = temp
        j +=1
    return x   # kann auch weggelassen werden


# Aufruf der auf das array a anzuwendenden Funktion sort
# mit Ermittlung des Zeitbedarfs zur Laufzeit

start = time.time()

sort(a)

end = time.time()


# Ausgabe der sortierten Liste

print()
print('Sortierte Liste:')
print()

for i in range(0,r):
    print(a[i])

print()
print('Zeitaufwand zum Sortieren von',n,'Elementen: {:7.3f}
s'.format(end-start))
```

Die Variante mit der auf das Array **a** anzuwendenden Funktion **sort** ist insofern flexibel, als daß man einen externen Programmierer mit der Aufgabe betrauen kann, eine Funktion zu konzipieren, die, angewandt auf ein array **a**, den Sortiervorgang übernimmt (in der professionellen Softwareentwicklung ist es üblich, ein komplexes Problem in Teilprobleme zu zerlegen); z. B. läßt sich der folgende Code mit Erfolg verwenden
(Quelle: https://en.wikipedia.org/wiki/Talk:Selection_sort#Implementations):

```
def selectionsort(list):
    for passesLeft in range(0,len(list)-1,+1):
        min   = passesLeft
        for index in range(passesLeft+1,len(list),+1):
            if (list[index]<list[min]):
                min = index
        list[min],list[passesLeft]=list[passesLeft],list[min]
    return list
```

Ersetze einfach die Funktion sort(x) durch selectionsort(list), der Aufruf lautet dann natürlich selectionsort(a) statt sort(a).

selectionsort scheint etwas effizienter zu laufen als sort. (Grund?)

Selbstverständlich läßt sich der Programmcode auch ohne eine Funktion sort(x) formulieren (zwei ineinander verschachtelte Schleifen):

```
from random import randint
import time

n = int(input('Laenge des arrays: '))
print()

a = list(range(1,n+1))

for i in range(0,n):
    a[i] = randint(1,1000000)

r = int(input('Wieviele Elemente sollen angezeigt werden? '))
print()
for i in range(0,r):
    print(a[i])


# Sortieren:

start = time.time()

j = 0
while j <= n-2:
        for i in range(j+1,len(a)):
```

```
            if a[i] < a[j]:
                 temp = a[j]
                 a[j] = a[i]
                 a[i] = temp
        j +=1

end = time.time()



print()
print('Sortierte Liste:')
print()

for i in range(0,r):
    print(a[i])

print()
print('Zeitaufwand zum Sortieren von',n,'Elementen: {:7.3f}
s'.format(end-start))
```

Ausblick:

Der Algorithmus „Sortieren durch direkte Auswahl" kann hinsichtlich seines zeitlichen Aufwands, der quadratisch (also insbesondere polynomial) mit der Anzahl n der zu sortierenden Datensätze wächst, noch als effizient gelten; allerdings erfordert das Sortieren von 100 000 Zufallszahlen bereits eine Rechenzeit in der Größenordnung von 10 – 20 min (je nach Rechner), so daß sich die Frage nach einem effizienteren Algorithmus stellt.

Tatsächlich arbeiten quicksort oder mergesort wesentlich effizienter. Ihr könnt gerne mal das Python-Programm

https://kalle2k.lima-city.de/computerscience/Informatik_13/MergeSort_final.py.txt

downloaden und ausprobieren; ein absoluter Anfänger in Python hatte sich einzelne Python-Teilcodes im Netz zusammengesucht und zu einem Algorithmus „mergesort", der Zufallszahlen sortiert, zusammengebunden.

Selbach
26.01.2021

10. Die Hofstadter-Funktion ist rekursiv definiert (n natürliche Zahl):

    Rekursionsanfang: hof(1) = 1
    hof(2) = 1

    Rekursionsvorschrift: $hof(n) = hof[n - hof(n-1)] + hof[n - hof(n-2)],\ n>2$

    Formuliert man den Algorithmus zur Berechnung der Hofstadter-Funktion als Python-Programm mit rekursivem Funktionsaufruf, haben wir die Erfahrung gemacht, daß die Rechenzeit für große Werte von n sehr schnell wächst; der Grund ist die mit n sehr schnell wachsende Anzahl gleichzeitig aktiver Aufrufe der Funktion hof.

    Dieses ungünstige Laufzeitverhalten läßt sich umgehen, indem man den Algorithmus zur Berechnung der Hofstader-Funktion iterativ formuliert.

    Vorschlag zur iterativen Formulierung:
    Definiere ein array a mit den Komponenten a[0], a[1], a[2], ... und setze a[0] = hof(1) = 1, a[1] = hof(2) = 1.
    Den weiteren Komponenten a[2], a[3], . . . werden in dieser Reihenfolge die Werte hof(3), hof(4), ... zugewiesen.

    Konzipiere und teste das iterativ formulierte Python-Programm!

11. Zusatzaufgabe:

    Die Fibonacci-Folge $\{a_i\}$ ist wie folgt definiert:

    $$\mathbf{a_1 = a_2 = 1}$$

    $$\mathbf{a_n = a_{n-1} + a_{n-2}} \quad \text{für } n \geq 3$$

    Schreibe und teste ein Python-Programm zur Berechnung der Fibonacci-Folge.

12. Der als Python-Programm formulierte Algorithmus
    sorting_by_direct_selection.py.txt auf
    https://kalle2k.lima-city.de/computerscience/Informatik_12/sorting/
    sortiert ein array von Zufallszahlen aufsteigend, d. h. die sortierte Liste beginnt mit dem kleinsten Element.

    Modifiziere das Programm so, daß das Sortieren absteigend erfolgt.