Arbeitsblatt Nr. 2 inf13 08.02.2023

## Sorting and Searching

### I. Imperative und funktionale Formulierung eines Algorithmus

**IMPERATIVER ANSATZ:**
Der in einer Programmiersprache formulierte Quellcode besteht aus einer Folge von ausführbaren Anweisungen, die in vorgegebener Reihenfolge nacheinander abgearbeitet werden; wesentliche Kontrollstruktur: **Iteration**
Bei einer for-Schleife ist die Anzahl der Schleifendurchläufe a priori bekannt.
Bei einer while-Schleife erfolgt die Abfrage der als Boolescher Term formulierten Bedingung vor Eintritt in den Schleifenrumpf (kopfgesteuerte Schleife), die repeat-Schleife (nicht in Python; in Java: do . . . while) fragt die Bedingung nach Durchlaufen des Schleifenrumpfs ab (fußgesteuerte Schleife).
Der Nachweis der Korrektheit des iterativ formulierten Algorithmus gestaltet sich mitunter schwierig; mögliche Methode: Auffinden und Formulierung einer *Schleifeninvariante*, Nachweis der Richtigkeit der Schleifeninvariante durch das Beweisverfahren *Vollständige Induktion*.

**FUNKTIONALER ANSATZ:**
Die Formulierung des Programmtexts (Quellcodes) orientiert sich der inneren, in der Regel mathematischen Struktur eines Algorithmus.
Wesentliche Kontrollstruktur: **Rekursion**
Vorteile:

- Elegante Formulierung des Quellcodes
- Da der funktional formulierte Algorithmus im wesentlichen eine am Problem orientierte Funktion (oder Prozedur) auswertet, gestaltet sich der Nachweis der Korrektheit vergleichsweise einfach.

Nachteil:

- Im Vergleich zu iterativen Lösungen in der Regel erhöhter Speicherplatzbedarf zur Laufzeit (Beispiel: Fibonacci-Folge mit exponentiellem Wachstum der Anzahl der Funktionsaufrufe)

Beispiel:

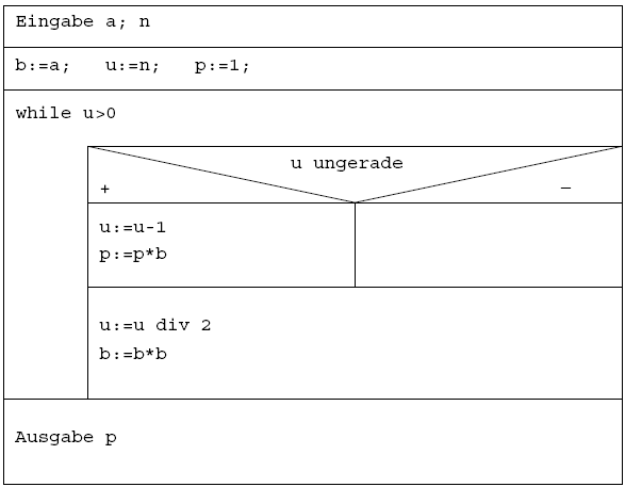
Programmspezifikation: nach Eingabe einer nicht negativen reellen Zahl **a** und einer natürlichen Zahl **n** ist die Potenz $\mathbf{a^n}$ zu berechnen.

Iterative Formulierungen:

a)

```
# Potenz a^n iterativ

a = float(input('a = '))
n = int(input('n = '))

y = 1
p = a

if n == 0: p = 1
else:
  while y <= n-1:
    p = p * a
    y = y + 1

print()
print (a,'^',n,' =',p)
```

b)



*Bemerkung: Die Korrektheit des Algorithmus b) erschließt sich nicht auf den ersten Blick; hier läßt sich aber eine Schleifeninvariante angeben, deren Korrektheit man beweisen kann.*

Arbeitsaufträge:

- Schreibe und teste ein Python-Programm gemäß Struktogramm b).
- Schreibe und teste ein funktional (insbesondere rekursiv) formuliertes Python-Programm.

Vereinbarung: Die folgenden Algorithmen zu Sorting and Searching operieren jeweils auf einem aus **n** Komponenten bestehenden array **a** (in Python als Liste realisiert) mit den Komponenten a[0], . . . , a[n-1].

## II. SelectionSort (Sortieren durch direkte Auswahl)

Die auf der Teilliste x mit den Komponenten a[j], . . , a[n-1] , j = 0, . . . , n-2, operierende Funktion **`min(x,j)`** bestimmt das kleinste Element und weist dieses der Komponente a[j] zu, der ursprüngliche Inhalt von a[j] wird nach Zwischenspeicherung in der Variablen temp derjenigen Komponente zugewiesen, der das Minimum entnommen wurde; nachdem **`min(x,j)`** für alle Werte von j ausgeführt wurde, ist die Gesamtliste sortiert.

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

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

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

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

Arbeitsaufträge:

- Markiere den Schleifenrumpf **A** innerhalb der Funktion **`min`**.
- Ermittle anhand folgender Tabelle die Gesamtzahl der Abarbeitungen des Schleifenrumpfs **A**; dabei gebe die Variable z(j) die Anzahl der Durchläufe für jeden Index j an.

| **Index j** | **Index i** | **z(j)** |
|---|---|---|
| j = 0 | ≤ i ≤ | |
| j = 1 | ≤ i ≤ | |
| j = 2 | ≤ i ≤ | |
| j = 3 | ≤ i ≤ | |
| .... | .... | .... |
| j = n-3 | ≤ i ≤ | |
| j = n-2 | ≤ i ≤ | |

Bemerkung:

Als imperativ formulierter Algorithmus bedient sich SelectionSort wesentlich der Iteration (hier: zwei ineinander verschachtelte Schleifen).

Dagegen ist MergeSort ein typischer Vertreter des funktionalen Ansatzes.

## III. **MergeSort** (Sortieren durch Mischen)

Arbeitsaufträge:

- Formuliere den Algorithmus MergeSort in Worten.
- Untersuche den Algorithmus hinsichtlich des Aufwands zum Sortieren von n Datenelementen.
- Zeige, daß der Speicherbedarf zur Laufzeit im Vergleich zum Gesamtaufwand zu vernachlässigen ist.

*Bemerkung:*
*Der zeitliche Aufwand A(n) zum Sortieren von n Datenelementen wächst linear-logarithmisch bei MergeSort:*

$$A(n) \sim n \cdot \log_2(n)$$

*Damit ist MergeSort wesentlich effizienter als SelectionSort mit quadratischer Komplexität; allgemein läßt sich beweisen, daß es kein Sortierverfahren gibt, dessen Aufwand zur Laufzeit schwächer als linear-logarithmisch wächst.*

## IV. **BinarySearch** (Binäre Suche)