Übung 8 Abgabe 20.06.2019

Aufgabe 1 – Fibonacci-Zahlen 13 Punkte

a) Implementieren Sie die Berechnung von Fibonaccizahlen mittels Baumrekursion (Funktion `fib1(N)` aus der Vorlesung), mittels Course-of-Values-Rekursion (`fib3(N)` aus der Vorlesung) und mit Iteration (`fib5(N)` aus der Vorlesung). Bestimmen Sie für alle drei Versionen die größte Zahl *N*, für die Python die zugehörige Fibonaccizahl berechnen kann, bzw. die größte Zahl *N*, für die die Berechnung weniger als etwa 10 Sekunden benötigt (falls ersteres zu lange dauert). Geben Sie Ihre Lösung im File "`fibonacci.py`" ab. Erklären Sie die Unterschiede, die Sie beobachten! *4 Punkte*

b) Eine elegante Methode zur Berechnung der *N*-ten Fibonaccizahl ist die Verwendung von Matrixpotenzen. Sei $F_1 = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ die (symmetrische) "Fibonacci-Matrix" und $F_N = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^N$ deren *N*-te Potenz. Matrixpotenzen sind genauso definiert wie gewöhnliche Potenzen, außer dass man statt der gewöhnlichen Multiplikation die Matrixmultiplikation verwendet. Für *N*=0 erhält man die Einheitsmatrix I der entsprechenden Größe, d.h. in unserem Fall (2x2 Matrizen) gilt $F_0 = I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$. Die *N*-te Fibonaccizahl f_N findet man stets in der Nebendiagonale von F_N, es gilt also *4 Punkte*

$$f_N = (F_N)_{12} = (F_N)_{21}$$

Implementieren Sie die Matrixmultiplikation für 2x2 Matrizen als Funktion `mul2x2()`, wobei Matrizen als Arrays der Länge 4 repräsentiert werden. Benutzen Sie dies, um eine weitere Funktion `fib6(N)` zu schreiben, die die Matrixpotenz in einer Schleife berechnet. Bestimmen Sie wiederum die größte Fibonaccizahl, die in weniger als 10 Sekunden berechnet werden kann.

c) Wir können die Effizienz weiter verbessern, wenn wir die Potenzen mit dem Verfahren der "binären Exponentiation" in logarithmischer Zeit $O(\log_2 N)$ berechnen. Im Gegensatz dazu hat `fib1(N)` exponentielle Komplexität $O(2^N)$, `fib5(N)` und `fib6(N)` (die besten Algorithmen bisher) benötigen lineare Zeit $O(N)$. Für eine Matrix *X* und nicht-negatives *N* ist die binäre Exponentiation rekursiv durch folgende Regeln definiert: *5 Punkte*

$$X^N = \begin{cases} I & \text{wenn } N = 0 \\ X & \text{wenn } N = 1 \\ (X * X)^{\frac{N}{2}} & \text{wenn } N \text{ gerade ist} \\ X * (X * X)^{\frac{N-1}{2}} & \text{wenn } N \text{ ungerade ist} \end{cases}$$

wobei '*' die Matrixmultiplikation ist. (Die Definition gilt natürlich ebenso, wenn *X* eine Zahl und '*' die gewöhnliche Multiplikation ist. Dies kann beim Testen des Codes hilfreich sein.) Das logarithmische Verhalten kommt zustande, weil man den Exponenten *N* bei jedem rekursiven Aufruf halbiert. Implementieren Sie diese Methode zur Berechnung der Fibonaccizahlen als `fib7(N)` im File "`fibonacci.py`". Testen Sie, dass `fib7(N)` die gleichen Ergebnisse liefert wie `fib5(N)` und finden Sie heraus, bei welchem *N* der neue Algorithmus etwa 10 Sekunden benötigt. Wie viele Dezimalstellen hat die resultierende Fibonaccizahl?

Aufgabe 2 – Binäre Suche 14 Punkte

Wir haben in der Vorlesung folgende Implementation von `binarySearch()` angegeben (a ist dabei ein sortiertes Array):

```
def binarySearch(a, key1, start, end):
    key = int(key1)
    size = end-start
    if size <= 0:
        return None
    center = (start + end) // 2
    if key == a[center]:
        return center
    elif key < a[center]:
        return binarySearch(a, key, start, center)
    else:
        return binarySearch(a, key, center+1, end)
```

In dieser Implementation wird das Array bei den rekursiven Aufrufen per Referenz übergeben, d.h. das Array wird niemals kopiert. Alternativ kann man das in den rekursiven Aufrufen benötigte Teilarray per Wert (also als Kopie) übergeben, indem man den Algorithmus folgendermaßen ändert:

```
def binarySearch2(a, key1):
    key = int(key1)
    if len(a) == 0:
        return None
    center = len(a) // 2
    if key == a[center]:
        return center
    elif key < a[center]:
        b=[]
        for t in a[:center]:
            b.append(t)
        return binarySearch2(b, key)
    else:
        b=[]
        for t in a[center+1:]:
            b.append(t)
        res = binarySearch2(b, key)
    if res is None:
        return None
    else:
        return res + center + 1
```

a[:center] `und` a[center+1:] erzeugen hier eine Kopie der linken bzw. rechten Hälfte des Arrays a.

a) Berechnen Sie mit dem Master-Theorem die Komplexität der beiden Varianten unter der Annahme, dass die Parameterübergabe per Referenz eine Zeit in $O(1)$ benötigt, während die Übergabe per Kopie $O(N)$ erfordert, wobei N die Anzahl der kopierten Arrayelemente darstellt. Hinweis: Die Komplexität ist für beide Varianten verschieden. *6 Punkte*

b) Überprüfen Sie mittels `timeit`, dass Ihre theoretische Berechnung korrekt ist. Messen Sie die Laufzeit für verschiedene N und plotten Sie die als Funktin von N. Das kann z.B. mit dem folgendem Code gemacht werden: *4 Punkte*

```
import matplotlib.pyplot as plt

xdata1 = []
```

```
    ydata1 = []
    xdata2 = []
    ydata2 = []
    for N in range(10000,300000,1000):
        t2  =  Timer("binarySearch2(a,str(randint(0,"+str(N)+")))","from  random  import
randint\nfrom __main__ import binarySearch2\na=range("+str(N)+")")
        time2 = t2.timeit(100)/100
        xdata2.append(N)
        ydata2.append(time2)
    for N in range(2,10000,100):
        t1  =  Timer("binarySearch(a,str(randint(0,"+str(N)+")),0,len(a))","from  random
import randint\nfrom __main__ import binarySearch\na=range("+str(N)+")")
        time1 = t1.timeit(100)/100
        xdata1.append(N)
        ydata1.append(time1)


    fig1 = plt.figure(1)
    ax1 = fig1.add_subplot(1, 1, 1)
    ax1.plot(xdata1, ydata1, color='tab:blue')
    ax1.set_title('BinarySearch plot')
    plt.xlabel('N')
    plt.ylabel('time (s)')

    fig2 = plt.figure(2)
    ax2 = fig2.add_subplot(1, 1, 1)
    ax2.plot(xdata2, ydata2, color='tab:orange')
    plt.xlabel('N')
    plt.ylabel('time (s)')
    ax2.set_title('BinarySearch2 plot')

    plt.show()
```

c) Die Funktion `binarySearch()` ist endrekursiv und kann daher leicht in eine iterative Form transformiert werden. Implementieren Sie die iterative Version `binarySearchI()` sowie geeignete Unit Tests und geben Sie diese ebenfalls im File `binarySearch.py` ab. *4 Punkte*

Bitte laden Sie Ihre Lösung bis zum 20.6.2019 um 12:00 Uhr auf Moodle hoch.