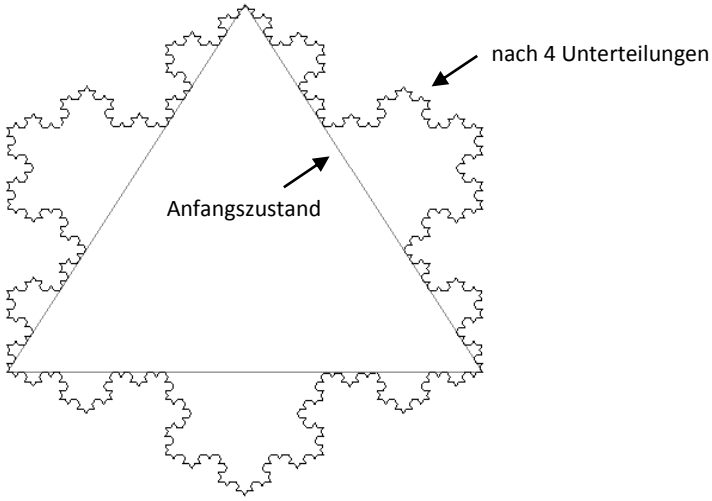
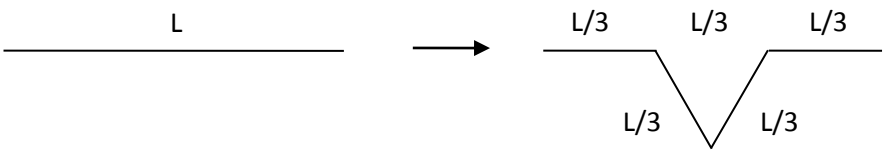
Übung 8 Abgabe 21.6.2012

Aufgabe 1 – Koch-Schneeflocke 14 Punkte

Entwickeln Sie ein Programm zur Berechnung der Koch-Schneeflocke:



Die Koch-Schneeflocke ist rekursiv definiert: Beginne mit dem gleichseitigen Dreieck $(0,0), (1,0), \left(\frac{1}{2}, \frac{\sqrt{3}}{2}\right)$. Teile jede vorhandene Strecke der Länge L in vier neue Strecken der Länge L/3 gemäß untenstehender Skizze (die Spitze bildet wieder ein gleichseitiges Dreieck):



Die entstehenden Strecken werden analog weiter unterteilt. Theoretisch kann dies unendlich oft wiederholt werden, praktisch wird man nach etwa 4 bis 8 Unterteilungen aufhören.

a) Geben Sie die formale Unterteilungsregel an: Wie werden aus gegebenen Endpunkten p_1, p_2 einer Strecke die neuen Punkte berechnet? Hinweis: Der auf einem gegebenen Vektor $\vec{u} = \begin{pmatrix} s \\ t \end{pmatrix}$ senkrecht stehenden Vektoren sind $\vec{v} = \begin{pmatrix} t \\ -s \end{pmatrix}$ und $\vec{v}' = \begin{pmatrix} -t \\ s \end{pmatrix}$. *6 Punkte*

b) Implementieren Sie die Funktion `points=kochSnowflake(level)`. Die Anzahl der Unterteilungen wird durch den Parameter `level` bestimmt, die resultierende Punktliste als Python-Array zurückgegeben. Speichern Sie die Punktliste in einem Textfile `"snowflake.txt"` (ein Punkt pro Zeile, x und y durch Leerzeichen getrennt), zeichnen Sie die Schneeflocke mit Gnuplot (oder einem anderen Programm) und exportieren Sie das Bild in eine Datei. Geben Sie die Bilddatei und das File `"snowflake.py"` ab. Ein geeignetes Gnuplot-Skript wäre z.B. *8 Punkte*

```
set term postscript color portrait # für Postscript-Ausgabe
set out "snowflake.eps"
set size square
set xrange [-0.2,1.2]
set yrange [-0.4:1.0]
plot "snowflake.txt" with lines
unset out
```

Aufgabe 2 – Fibonacci-Zahlen 10 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) Die effizienteste 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. (Zur Erinnerung: 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 *6 Punkte*

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

Dies führt zu einem schnellen Algorithmus, weil man Potenzen mit dem Verfahren der "binären Exponentiation" in $O(\log_2 N)$ berechnen kann, während `fib5(N)` (der beste Algorithmus aus Teilaufgabe a) lineare Zeit $O(N)$ benötigt. Für eine Matrix *X* und nicht-negatives *N* ist die binäre Exponentiation rekursiv durch folgende Regeln definiert:

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

wobei '*' die Matrixmultiplikation ist. Das logarithmische Verhalten kommt zustande, weil man den Exponenten *N* bei jedem rekursiven Aufruf halbiert. Implementieren Sie diese Methode zur Berechnung der Fibonaccizahlen als `fib6(N)` im File "`fibonacci.py`". Testen Sie, dass `fib6(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 3 – Binäre Suche 16 Punkte

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

```
def binarySearch(a, key, start, end):
    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.[1] Alternativ kann man das in den rekursiven Aufrufen benötigte Teilarray per Wert übergeben, indem man den Algorithmus folgendermaßen ändert:

```
def binarySearch2(a, key):
    if len(a) == 0:
        return None
    center = len(a) / 2
    if key == a[center]:
        return center
    elif key <  a[center]:
        return binarySearch2(a[:center], key)
    else:
        res = binarySearch2(a[center+1:], 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 Wert in $O(N)$ ist, wobei N die Anzahl der kopierten Arrayelemente darstellt. Hinweis: Die Komplexität ist für beide Varianten verschieden. *5 Punkte*

b) Überprüfen Sie mittels `timeit`, dass Ihre theoretische Berechnung korrekt ist. Geben Sie die gemessenen Laufzeiten als Tabelle oder Diagramm ab. *5 Punkte*

c) Die Funktion `binarySearch()` ist endrekursiv und kann daher leicht in eine iterative Form transformiert werden. Implementieren Sie die iterative Version sowie geeignete Unit Tests und geben Sie diese im File `binarySearch.py` ab. *6 Punkte*

[1] Zur Wiederholung von Referenz- und Wertsemantik siehe Übung 1.