10.2 UNLÖSBARE PROBLEME

EINFÜHRUNG

Mit gescheiten Computerprogrammen kann man zwar viele und immer mehr Probleme lösen. In diese Kapitel wirst du aber mit einfach zu formulierenden Fragen konfrontiert, die möglicherweise trotz de rasanten Entwicklung der Computer und enormem wissenschaftlichem Aufwand nie algorithmisc lösbar sein werden.

PROGRAMMIERKONZEPTE: *Unlösbares Problem, Teilsummenproblem, Aufzählungsverfahren, Kombinatorische Explosion, Polynomiale Ordnung, Unentscheidbares Problem*

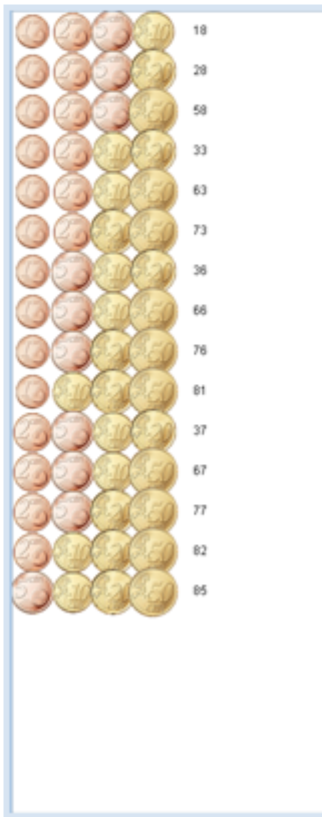
UNLÖSBARE PROBLEME

Es harren noch einige Probleme der Lösung, die einfach zu formulieren und auch für die Praxis vo grosser Bedeutung sind. Eines davon, bekannt unter dem Namen **Teilsummenproblem**, kann au folgende Problemstellung zurückgeführt werden [mehr...]:

Du hast in deinem Geldbeutel eine Anzahl von Münzen und musst damit einen bestimmten Betrag (ohne Rückgeld) bezahlen. Ist dies mit den vorhandenen Münzen möglich und wenn ja, mit welchen Münzen musst du dabei bezahlen?

In deinem ersten Programm lernst du zuerst mit den Münzen umzugehen. Die Namen der Euro-Münzen mit den Werten 1, 2, 5, 10, 20, 50 cents speicherst du in der Liste *coins*. Die Funktion *value()* liefert den Wert einer Münze zurück. Der Geldbeutel wird mit einer Liste (oder einem Tupel) *moneybag*, welche die Namen der vorhandenen Münzen enthält. Die Funktion *getSum(moneybag)* liefert den Wert aller Münzen im Geldbeutel.

Der Geldbeutel soll zuerst alle Münzen genau einmal enthalten und du bildest alle möglichen Münzkombinationen mit 1, 2, 3, 4, 5 und 6 Münzen, die du in einem *JGameGrid*-Fenster darstellst. Dazu machst du in *showMoneybag(moneybag, y)* aus jeder Münze des *moneybags* einen Actor und stellst diese im Spielfenster in der Zeile y dar.




```
from gamegrid import *
import itertools

coins = ["one", "two", "five", "ten", "twenty", "fifty"]

def value(coin):
    if coin == "one":
        return 1
    if coin == "two":
        return 2
    if coin == "five":
```

```
        return 5
    if coin == "ten":
        return 10
    if coin == "twenty":
        return 20
    if coin == "fifty":
        return 50
    return 0

def getSum(moneybag):
    count = 0
    for coin in moneybag:
        count += value(coin)
    return count

def showMoneybag(moneybag, y):
    x = 0
    for coin in moneybag:
        loc = Location(x, y)
        removeActor(getOneActorAt(loc))
        coinActor = Actor("sprites/" + coin + "cent.png")
        addActor(coinActor, loc)
        x += 1
    addActor(TextActor(str(getSum(moneybag))), Location(x, y))

makeGameGrid(8, 20, 40, False)
setBgColor(Color.white)
show()

n = 6
k = 1
while k <= n:
    combinations = list(itertools.combinations(coins, k))
    setTitle("(n, k) = (" + str(n) + ", " + str(k) + ") nb = "
    + str(len(combinations)))
    y = 0
    for moneybag in combinations:
        showMoneybag(moneybag, y)
        y += 1
    getKeyCodeWait()
    removeAllActors()
    k += 1
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Die Kombinationen von k Elementen, die du aus einer Liste von n Elementen bilden kannst, lassen sic elegant mit der Funktion *combinations()* aus dem Modul *itertools* herausholen. Du musst de Rückgabewert in eine Liste umwandeln, in der sich die gefundenen Kombinationen dann (als Tupels herausholen lassen.

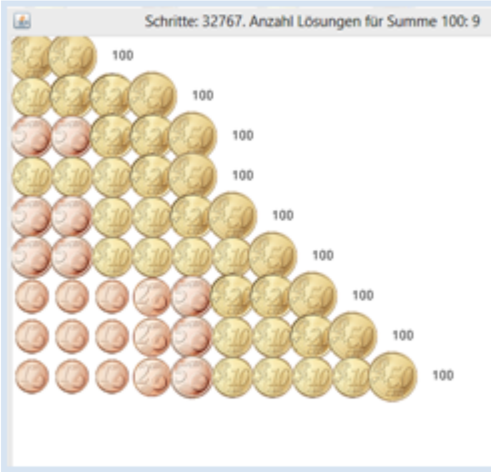
Wie du siehst, sind die damit erhaltenen Kombinationen so geordnet, wie du es vernünftigerweise auc von Hand machen würdest. Die Anzahl der Kombinationen von n Elementen zur Ordnung k kannst d bekanntlich wie folgt berechnen:

$$c = \binom{n}{k} = \frac{n!}{k!*(n-k)!}$$

wo n! die Fakultät, also das Produkt aller Zahlen von 1 bis n bedeutet. Für unserem Fall mit n = ergeben sich 6, 15, 20, 15, 6, 1, also insgesamt 63 Kombinationen.

Du kannst jetzt das Teilsummenproblem beim Geldbeutel wie folgt lösen: Du bestimmst alle Kombinationen der vorhandenen Münzen und untersuchst sie einzeln, ob ihre Summe den gewünschten Wert ergibt.

Dieses **Aufzählungsverfahren** ist wohl nicht das beste, ist aber sicher korrekt und liefert alle möglichen Lösungen. Für eine Geldbörse mit 3 Eincent, 1 Zweicent, 2 Fünfcent, 4 Zehncent, 2 Zwanzigcent und 3 Fünfzigcent-Münzen, also insgesamt 15 Münzen wäre es bereits schwierig, die Lösungen von Hand zu finden. Du schreibst nur unterschiedliche Münzzusammenstellungen aus, die zusammen 1 Euro ergeben.




```
from gamegrid import *
import itertools

coins = ["one", "one", "one", "two", "five", "five",
         "ten", "ten", "ten", "ten", "twenty", "twenty",
         "fifty", "fifty", "fifty"]

def value(coin):
    if coin == "one":
        return 1
    if coin == "two":
        return 2
    if coin == "five":
        return 5
    if coin == "ten":
        return 10
    if coin == "twenty":
        return 20
    if coin == "fifty":
        return 50
    return 0

def getSum(moneybag):
    count = 0
    for coin in moneybag:
        count += value(coin)
    return count

def showMoneybag(moneybag, y):
    x = 0
    for coin in moneybag:
        loc = Location(x, y)
        removeActor(getOneActorAt(loc))
        coinActor = Actor("sprites/" + coin + "cent.png")
        addActor(coinActor, loc)
        x += 1
    addActor(TextActor(str(getSum(moneybag))), Location(x, y))

makeGameGrid(15, 20, 40, False)
setBgColor(Color.white)
show()

target = 100

k = 1
result = []
count = 0
```

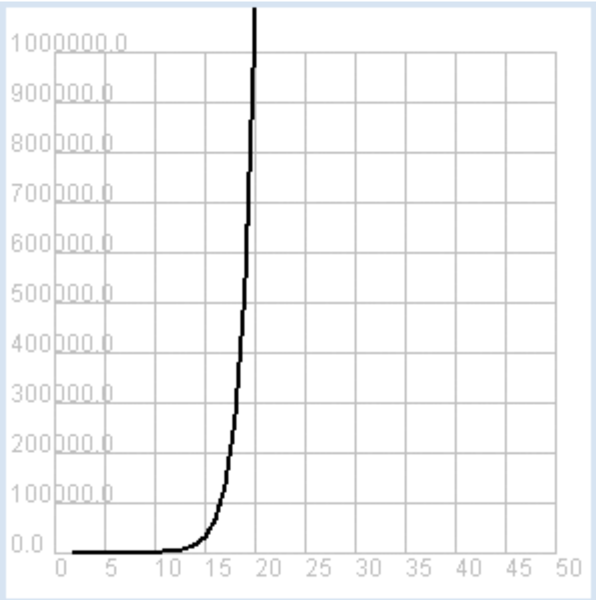
```
while k <= len(coins):
    combinations = tuple(itertools.combinations(coins, k))
    nb = len(combinations)
    for moneybag in combinations:
        count += 1
        count = getSum(moneybag)
        if count == target:
            if not moneybag in result:
               result.append(moneybag)
    k += 1

y = 0
for moneybag in result:
    showMoneybag(moneybag, y)
    y += 1
setTitle("Step: " + str(count) + ". number of solutions for the count  "
          + str(target) + ": " + str(len(result)))
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Bereits mit nur 15 Münzen sind 32767 Schritte nötig, um das Teilsummenproblem mit dem Aufzählungsverfahren zu lösen.

Deine Freude an der Computerlösung wird leider getrübt, wenn du versuchst, einen etwas grösseren Geldsack, sagen wir mit 50 oder 100 Münzen, zu verwenden. Zählst du nämlich für einen Geldbeutel mit n Münzen die nötigen Schritte zusammen und trägst sie in einer Grafik auf, so gibt es bei n = 2 eine regelrechte **kombinatorische Explosion** und du stösst an eine Grenze des Machbaren [mehr...].




```
from gpanel import *
from math import factorial

z = 100

def nbCombi(n, k):
    return factorial(n) / factorial(k) / factorial(n - k)

makeGPanel(-5, 55, -1e5, 1.1e6)
drawGrid(0, 50, 0, 1e6, "gray")
setColor("black")
lineWidth(2)
```

```
for n in range(2, z + 1):
    count = 0
    for k in range(1, n):
        count += nbCombi(n, k)
    print "n =", n, ", nb =", count
    if n == 2:
        move(n, count)
    else:
        draw(n, count)
print "Runtime with 10^9 operations per second:", count / 3.142e16, "years"
print "or:", int(count / 2e20), "times the age of the universe"
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Das Teilsummenproblem ist bereits bei relativ kleiner Anzahl von Elementen mit de Aufzählungsverfahren **unlösbar**, obschon das Lösungsverfahren bekannt ist. Es fragt sich daher, ob e **wesentlich bessere Algorithmen** zu seiner Lösung gibt, wenn möglich solche wie beim Sortiere deren Schrittzahl oder Komplexität eine Potenz von n, also **polynomial** ist. Leider ist es bis heute nic gelungen, einen solchen Algorithmus für das Teilsummenproblem zu finden, und man vermutet, dass e keinen solchen gibt. Allerdings gibt es auch keinen theoretischen Beweis für diese Vermutung.

Immerhin weiss man heute, dass es eine Vielzahl **ähnlich schwieriger Probleme** gibt, und dass ma damit rechnen kann, auf einen Schlag alle diese Probleme mit polynomialer Komplexität zu lösen, wen man für eines davon eine solche Lösung findet [mehr...].

UNENTSCHEIDBARE PROBLEME

Die Grenzen des menschlichen Verstands und der Computertechnik werden noch in einem andere Zusammenhang als bei der Komplexität sichtbar. Der Mathematiker und Zahlentheoretiker Loth Collatz hat bestimmte Zahlenfolgen natürlicher Zahlen untersucht und 1939 folgende Frage formuliert:

Gehe von irgend einer Startzahl n aus und bilde die Nachfolgezahlen nach folgender Regel:

- Ist n gerade, so teile n durch 2 (wieder eine natürliche Zahl)
- Ist n ungerade, so bilde die Nachfolgezahl 3n +1 (eine gerade Zahl)

Frage: Erreicht diese Folge mit jeder möglichen Startzahl n immer die Zahl 1?

Collatz und viele andere Zahlentheoretiker und Computerwissenschafter haben sich um eine Lösun bemüht, denn selbst mit den grössten und schnellsten Computern ergeben sich immer Folgen, die bei landen. (Die Folge konvergiert nicht, denn fährt man weiter, so wird endlos die Sequenz 4,2 durchlaufen).

Darum liegt die **Vermutung** nahe, dass der folgende Satz gilt:

Die 3n+1-Folge erreicht für alle natürlichen Startzahlen n nach endlich vielen Schritten die Zahl 1.

Mit einem Computerprogramm kannst du für eine beliebig vorgebbare Startzahl die 3n+1-Folg durchlaufen.

```
from gpanel import *

def collatz(n):
    while n != 1:
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
```

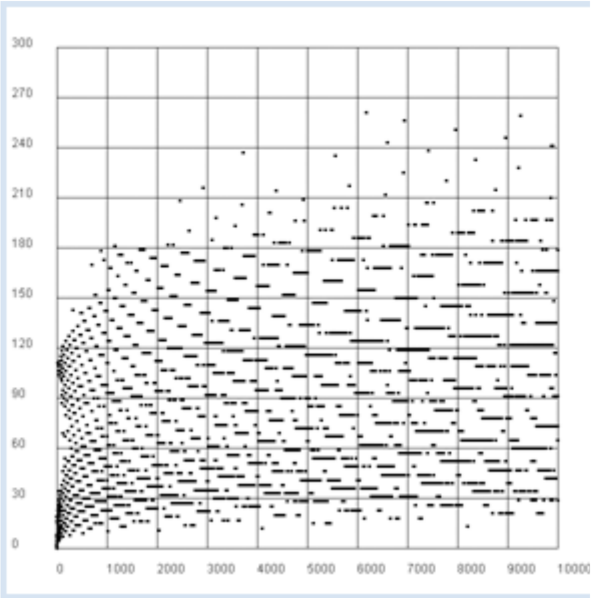
```
        print n,
    print "Result 1"
while True:
    n = inputInt("Enter a start number:")
    collatz(n)
```

[Programmcode markieren](#) (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

In Python kannst du sogar mit grossen Startzahlen die 3n+1-Folge durchlaufen und feststellen, dass d immer bei 1 landest. Damit hast du aber natürlich die Vermutung nicht bewiesen.

Interessant und ästhetisch ansprechend ist es, die Länge der 3n+1-Folge in Abhängigkeit von der Startzahl aufzutragen. Diese schwankt nämlich beträchtlich. Dazu entfernst du in der Funktion *collatz()* das Ausschreiben der Folgeglieder und gibst lediglich die Anzahl Schritte zurück.




```
from gpanel import *

def collatz(n):
    nb = 0
    while n != 1:
        nb += 1
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
    return nb

z = 10000 # max n
yval = [0] * (z + 1)
for n in range(1, z + 1):
    yval[n] = collatz(n)
ymax = (max(yval) // 100  + 1) * 100

makeGPanel(-0.1 * z, 1.1 * z, -0.1 * ymax, 1.1 * ymax)
title("Collatz Assumption")
drawGrid(0, z, 0, ymax, "gray")

for x in range(1, z + 1):
    move(x, yval[x])
    fillCircle(z / 200)
```

[Programmcode markieren](#) (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Die Vermutung von Collatz ist ein hartnäckiges Problem. Falls die Vermutung stimmt, so lässt sie sic nicht beweisen, in dem man Computertests mit immer grösseren Startzahlen durchführt. Es könnt allerdings sogar sein, dass die Vermutung zwar richtig ist, aber nie einen Beweis dafür gefunden wir denn 1931 hat der Mathematiker Kurt Gödel mit dem **Unvollständigkeitssatz** gezeigt, dass es in eine Theorie durchaus richtige Sätze geben kann, deren Korrektheit aber nicht bewiesen werden kann.

Die Vermutung von Collatz lässt sich auch als **Entscheidungsproblem** formulieren:

Stoppt ein Algorithmus, der die Glieder der 3n+1-Folge berechnet und bei 1 anhält, mit Sicherheit fü beliebig vorgebbare Anfangswerte?

Man könnte versuchen, diese Frage mit einem Computer zu lösen. Leider könnte auch die hoffnungslos sein, denn der grosse Mathematiker und Informatiker Alan Turing hat mit de **Halteproblem** bewiesen, dass es nie einen Algorithmus geben wird, mit dem man für alle Programm entscheiden kann, ob sie anhalten.

Die 3n+1-Vermutung von Collatz könnte also zwar stimmen, aber ein **unentscheidbares Problem** sein