Compact Course Python

Exercise 3

1 Prime numbers

1.1 Prime numbers with `for`

In the first lecture, we introduced an algorithm that finds all primes in a certain range of numbers (e.g. from 2 to 100) using a `while`-loop.

Write a function that computes prime numbers using a `for`-loop. The function shall take the limits of the range as its parameters. The return value shall be a list containing all the prime numbers within that range.

```
>>> primes(2,50)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

1.2 Prime numbers with list comprehension

Write a function that computes prime numbers between two given numbers using list comprehension. The return value should be defined using as few list comprehensions as possible.

2 Quicksort

A recursive sorting algorithm for lists is *Quicksort*:

- Chose a random element `e` out of the list (this is called a "pivot element")
- Create two lists: the first list shall contain every element which is smaller than `e`, the second lest shall contain every element that is greater or equal to `e`. [1]
- Sort the two lists (with the same procedure).
- Return a list which is the concatenation of the first list, `e` and, at last, the second list.

Implement a Quicksort function that takes a list of numbers as parameter and returns a sorted list. (*Don't* use Python's `sorted` function!)

```
>>> quicksort([5, 1, 23, 934, 42, 3234, 432, 234, 561, 451, 4, 5, 1, 123, 54])
[1, 1, 4, 5, 5, 23, 42, 54, 123, 234, 432, 451, 561, 934, 3234]
```

[1] `e` is in none of the two lists. The standard algorithm requires all elements that are equal to `e` to be distributed over both lists at random. We simplify this step here.

3 Fibonacci numbers

3.1 Computing them iteratively

On the slides of the second lecture, we show a recursive algorithm that computes the Fibonacci numbers. Implement a *non-recursive* function that computes the Fibonacci numbers using a loop.

```
>>> fibonacci_iter(5)
5
```

3.2 Memorization

Although it is easier to read, the recursive definition of the fibonacci numbers is slowed down by repeated calculation of the same values. A dictionary can be used to store and re-use known values. A useful approach can look as follows:

```
def fibonacci_cache(n,  cache={}):
  if n in cache: return n
  else:
    # compute the value
    # store it in cache
    # return it
```

Implement the rest of this function and compare its performance against the recursive and the iterative definition.

4 Calendar

Implement two functions that provide a (very simplistic) organizer function. The first function `addEvent(year,month,day,what,cal)` shall add an event to this calendar. The parameters should be year, month, day (as numbers), and a String containig the event name.

The second function `getEvents(year,month,day,cal)` shall extract all appointments for the given date (format as above) and return it as a (readable) string.

Chose appropriate data structures for the calendar. A possible program run could look like this:

```
>>> cal = ... % an empty calendar, initialized with your structure
>>> addEvent(2009,11,12,"Python Course", cal)
>>> addEvent(2009,11,12, "CoLi Party", cal)
>>> addEvent(2009,11,13, "Hangover", cal)
>>> addEvent(2009,12,24, "Christmas",cal)
>>> print(getEvent(2009,11,12,cal))
['Python Course', 'CoLi Party']
```

Optional extra task (more tricky): Implement two additional methods `getEventsForMonth(year,month,cal)` and `getEventsForYear(year,cal)` that return all events for the given month or the given year resp.

```
>>> print(getEventsForMonth(2009,11,cal))
12.: ['Python Course', 'CoLi Party']
13.: ['Hangover']
>>> print(getEventsForYear(2009,cal))
12.11.: ['Python Course', 'CoLi Party']
13.11.: ['Hangover']
24.12.: ['Christmas']
```

5 Combining Weights

Assume you have a set of weights $w_1 ... w_n$ and you would like to find a subset of them that gets as close as possible to a certain total weight. For n weights, there are 2^n possible subsets, and there is no known algorithm that would solve this problem efficiently.

Write a function `findSums(weights,minSum,maxSum)` that takes three arguments:

`weights` - a list of weights
`minSum` - a minimum total weight
`maxSum` - a maximum total weight

The function should return a list of all subsets of the weights for which the sum falls into the allowed range.

E.g. a call

```
»> findSums([1,2,4,8],10,13)
```

should return

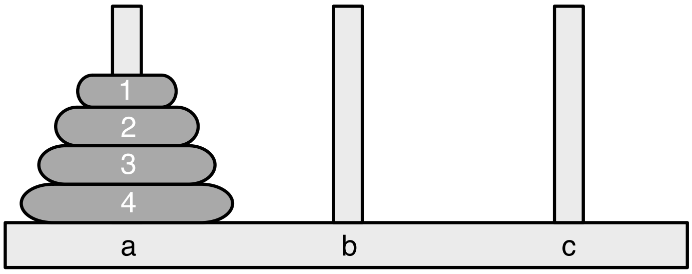
```
[[2,8],[1,2,8],[4,8],[1,4,8]]
```

(or a permutation thereof)

Hint: Use list comprehensions to generate all subsets of the given weights. Use another list comprehension to extract only the good solutions from the solution candidates.

6 Generating Passwords

Passwords should be easy to remember but hard to guess. Write a function that generates all passwords of a given length where vowels alternate with consonants. Use list comprehensions for your implementation. Provide an optional argument for specifying a set of known words that are not allowed as result.



7 The Towers of Hanoi (Extra task)

The Towers of Hanoi are a famous example for problems that have elegant recursive solutions. The game setup containts three bars (a, b, c in the picture) and n discs (n=4 in the picture). The goal is to bring the disc tower from bar a to another bar (e.g. c). You may move only one disc at a time, and a disc may only be put on a larger disc or on an empty bar.

Figure out and implement the algorithm that solves the Towers of Hanoi for n discs. One possibility to implement this is to print the moves in the correct order (something like *Move disc x from a to c.*)
