Homework Set #2

Imperative Progamming with Python

January 2012

The homework should be uploaded using the BlackBoard system, it should *not* be printed. You should upload a compressed (`.zip`) file containing:

- A text (`txt`, Word, `ps`, PDF) file with all the answers to the exercises that do not contain source code (in the answer).
- For each exercise including source code, a directory named '`exerciseN`' where `N` is the number of the exercise. This directory should include *all* the needed files and directory structure to run the program (if applicable).
- When an exercise asks you to modify or write a variant of a previous program you should include all the versions, not just the last one.

Homework not uploaded by the due date will be considered failed. The overall score is 100/100. Each section is worth 50 points, you need at least 25 in each section to pass. You may choose to do the exercises you want, but keep in mind that those marked with a star (★) are *mandatory* and the chosen exercises should add up to 50 points in each section (not more). Estimate the time for each exercise as in the first homework set. **Due date:** 23/01/2012.

1 Data structures

Exercise 1 (5 pts.)**.** Read Section 10.2 from the book and solve Exercise 10.2 ("Chop and middle").

Exercise 2 (5 pts.)**.** Do Exercise 10.3 from the book ("Sorted").

Exercise 3 (5 pts.)**.** Do Exercise 10.4 from the book ("Anagram").

Exercise 4 (15 pts. ★)**.** *Searching and sorting.*

a) Write a function `read_word_list(fn)` which, given a filename, works as the program described in Exercise 13.1 from the book ("Read list of words") but with the additional constraint that the list returned shouldn't have repeated words.

b) Write a function `linear_search(e, t)` which looks for element `e` in list `t` by sequentially checking every element. Observe that, if the length of the list is n then this function will take approximately n steps to finish. This is called the *temporal complexity* of the algorithm.

c) Write a function `sort(t)` which takes a list and sorts it in ascending order ($<$). Observe that the function it should modify the parameter and not return a new list. You can *not* use the sort method of lists nor any built-in sorting function. For this point you should choose one of the following three sorting algorithms and implement it: Bubble sort, Insertion sort, Selection sort. Check (at least) the Wikipedia pages for a description of the algorithms.

d) Write a function `binary_search(e, t)` which looks for element `e` in list `t` by using the method described in Exercise 10.8 in the book. The function should return `True` iff the element is found in the list. Observe that, as described in the Exercise, the temporal complexity of this algorithm is approximately $\log_2(n)$ where n is the length of the list.

e) Write a program called `findin.py` which, when executed, receives two arguments: a filename and a word. The program should use the functions from the last points to read the file into a list and look for the word using the linear and the binary search methods. For each method it should print whether it found it or not and how long (seconds) it took. It should be executed like: `python findin.py hamlet.txt cake`. Try it with some big files and include the results in the homework report (also include the files you used).[1]

[1] You can download free books from Project Gutenberg.

Exercise 5 (10 pts.)**.** *Memos.*

a) Read section 11.5 from the book.

b) The *spatial complexity* of an algorithm is the amount of space (memory) it needs to run, with respect to the size of the arguments. In this case, for the fibonacci function, the size of the arguments is n itself. Observe that the implementation given in section 11.5 has a spatial complexity of approximately n (it saves all the values from 0 to n). I know you can do better: Implement a memoizing version of fibonacci with spatial complexity 2. That is, you can only save 2 fibonacci values. Also, you should not use recursion.

   That's great! Do you see it? We started with a function which was slow, used a lot of memory and whose call stack was deep and, with some clever tricks, ended up with an iterative, fast version which uses a constant amount of memory no matter which number you want to calculate.

Exercise 6 (15 pts. ★)**.** *Word prediction and Markov Analysis.*

a) Read chapter 13 from the book.

b) Do exercises 13.1–13.3, 13.5–13.8 from the book.

Exercise 7 (5 pts.)**.** Do exercise 13.9 from the book ("Zipf's law in natural language"). Plotting the graph is optional but "nice to have". In case you do it, attach the text you used to get it.

2 Classes and user-defined types

Exercise 8 (5 pts.)**.** Do exercise 16.8 from the book ("Date and time").

Exercise 9 (5 pts. ★)**.** *Information hiding.*
Another way of representing colors in the RGB space is as a 3-tuple of integers (r,g,b) where $r, g, b \in \{0, \ldots, 255\}$. Take the type `Color` defined in class #6 and change its internal representation of colors to use this one. It is truly and utterly important that the interface of the class stays the same: all the methods should receive and return the same types and in the same range as before. The idea is that the change in the internal representation should go unnoticed to the outside world.

Exercise 10 (10 pts.)**.**

a) Write a user defined type `Matrix` which represents a matrix of integers. You should be able to sum, multiply and compare them using the `+`, `*` and `==` operators respectively.

b) Suppose that, because of our context, we know that sometimes we use *sparse matrixes*, that is, most of the elements are zero. Add/modify the necessary classes to take advantage of this feature and save some memory (probably at the expense of time).

Exercise 11 (15 pts.)**.** *Propositional model checking.*
A formula from propositional logic is defined as $\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi$. A valuation is a function assigning `True` or `False` for every propositional variable. We want to write a function `sat(phi, v)` which returns `True` iff the formula *phi* is satisfied by the valuation v.

a) Choose appropriate representations for the valuation and for the formulas and define the necessary classes. Think object-oriented: what do all formulas have in common? How are they constructed?[2]

b) Write the `sat` function using the types of the last point. Implement the formulas in such a way that printing them shows a nice representation of the formula, e.g: `~(p & q) & r`.

c) Suppose now we consider $\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \Rightarrow \varphi$. Modify the program of the last point to include the implication. It should be easy to do and you shouldn't need to modify any of the previous classes. If that is not the case, probably the design was not so good!

[2]Hint: consider a base class `PropositionalFormula` with a pure virtual method `sat(v)`.

Exercise 12 (15 pts. ★)**.** *Bank accounts.*
Consider the following context: We will be dealing with bank accounts. They have an associated number and owner. You can withdraw and deposit money to the accounts. We have two types of bank accounts, the "Normal" account and the "Savings" account. The savings account is free to be opened whereas the normal account costs 1 euro. It is free to deposit in both account types and it is also free to withdraw from the normal account. In the case of the savings account you can withdraw for free once a day, after that, it costs 2.5 euro each time.

You should write a program that simulates the bank: a client should be able to open a new account, check the balance of their accounts, deposit and withdraw. The program should start as a bank without any account and get filled as used by (possibly) different clients. When the program is asked to finish, the information about the accounts gets lost.

Exercise 13 (10 pts.)**.** In **R**ole **P**laying **G**ames you control a character which has a name, hit points (life) and strength. Characters can be of two different classes: Fighter, Wizard. They also have a race: Human or Orc. The characters use weapons to fight, these weapons can be of different types: blunt (e.g: staff), piercing (e.g: rapier, dagger), long range (e.g: bow). When a character attacks another one, the damage is calculated using the formula $d = \text{strength} + \text{class modifier} + \text{weapon damage}$. The class modifier is +1 for Fighter and -1 for Wizard. Each weapon has its own damage points.

Read section 18.8 of the book and draw a class diagram for the RPG described above. Also include in the diagram the list of attributes and methods that you think are important for the scenario. You don't need to write any code.

Exercise 14 (5 pts.)**.** *Serializing.*
Modify Exercise 12 to save the accounts information to a file so that it gets reloaded when the program is ran again.

Exercise 15 (5 pts.)**.** The built-in function `sum(t)` calculates the sum of a list of ints. It works with some other types but not for strings. Write a function `sumall` which works for every type that implements the `+` operator.