DIT027 – Assignment 2

The exercises are supposed to be done individually. However, it is allowed to discuss the problems with other students and share ideas with other students. There are 14 problems to solve. Solving at least 11 of them is enough for passing the assignment.

Don't forget to write comments in your code to make your solutions clearer!

The submission deadline is **12 September 2016** at 12:00 (noon).

A test suite will be provided to test the module that you create. It is required to test your solution with the test suite before submitting, and to include the output from running the test suite on your code.

The test suite will also be used to evaluate your code. Passing the test suite is required, but not sufficient to pass the assignment.

You should also not use library functions, but define your own.

Overcomplicated solutions will not be accepted.

Make sure that you follow ALL the instructions closely!

GOOD LUCK!

1. Download the module skeleton **assignment2.erl** from GUL (from Documents). You should be able to compile the module without modifications

2. **Refactoring –** In this problem you are supposed to modify Erlang functions without changing their functionality.

   A. Modify the function **price/1**, into a function with multiple clauses. The function should not use case- or if-expressions.

   Ex: price({orange, 2}) → 32

   B. Modify the function **stretch_if_square/1**, so that it uses a case-expression. The function should not use if-expressions.

   Ex: stretch_if_square({rect, 2, 2}) → {rect, 2, 4}

   C. Rewrite the function **convert/1** from Assignment 1, so that it uses

a case-expression.

D. Implement the function **rect_overlap/2** from Assignment 1, so that it uses a case-expression and makes use of **range_overlap/2**.

3. Basic recursion

These questions require writing simple recursive functions. Solution for **A** is provided. You may need to define auxiliary functions.

A. Function **print_0_n/1** when given a non-negative integer **N** prints numbers from **0** to **N** separated by newlines. The function returns the atom **ok**.

Ex: print_0_n(3) should return **ok** and print:
0
1
2
3

B. Write a function **print_n_0/1**, which given a non-negative integer **N** prints numbers from **N** to **0** separated by newlines. The function should return the atom **ok**. Implement the solution using an auxiliary recursive function that starts the recursion from **0** and goes up.

Ex: print_n_0(3) should return **ok** and print:
3
2
1
0

C. Write a function **print2_n_0/1**, which given a non-negative integer **N** prints numbers from **N** to **0** separated by newlines. The function should return the atom **ok**. Implement the solution by counting down from **N**.

Ex: print2_n_0(3) should return **ok** and print:
3
2
1
0

D. Write a function **print_sum_0_n/1**, which given a non-negative integer **N** prints numbers from 0 to **N** separated by newlines, and also computes the sum of numbers from 0 to **N**. The function should return the sum of the numbers. The function could be implemented easily by running two separate recursive functions, but this task requires that all work is implemented as one recursive function (and possibly more non-recursive functions).

Ex: print_sum_0_n(3) should return **6** and print:
0
1
2
3

4. Recursion on bits

Integers are represented in binary form on most CPU architectures. As we consider only non-negative integers, they can be represented as sequences of digits 0 and 1. For example, the decimal number 10 is represented as binary 1010 (10 = 2^3 + 2^1), whereas the decimal number 7 is represented as binary 111 (7 = 2^2 + 2^1 + 2^0).

Solutions for **A** and **B** are provided.

A. The function **lg/1** counts the length of the binary representation of a non-negative number. The examples below show binary representations of numbers as <XXXX>

Ex:
lg(1) → 1	1	= <1>
lg(10) → 4	10	= <1010>
lg(7) → 3	7	= <111>

A solution is presented here.

```
lg(0) -> 0;
lg(N) -> 1 + lg(N div 2).
```

Here is an example run of the function.

lg (6) →	6 = <110>
1 + lg(3) →	3 = <11>
1 + 1 + lg(1) →	1 = <1>
1 + 1 + 1 + lg(0) →	0 = <>
1 + 1 + 1 + 0→	
1 + 1 + 1 →	
1 + 2 →	
3	

B. The function **count_one_bits/1** counts the 1s in the binary representation of a non-negative number.

Ex:
count_one_bits(3) → 2	3	= <11>
count_one_bits(11) → 3	11	= <1011>
count_one_bits(5) → 2	5	= <101>

A solution is presented here.

```
count_one_bits(0) -> 0;
count_one_bits(N) ->
  (N rem 2) + count_one_bits(N div 2).
```

Here is an example run of the function.

count_one_bits(5) → 5 = <101>
1 + count_one_bits(2) → 2 = <10>
1 + 0 + count_one_bits(1) → 1 = <1>
1 + 0 + 1 + count_one_bits(0) → 0 = <>
1 + 0 + 1 + 0→
1 + 0 + 1 →
1 + 1 →
2

C. Write a function **print_bits/1**, which given a non-negative integer prints its binary representation, each digit in a separate line. The function should return **ok**.

Ex: print_bits(6) should return ok and print:
1
1
0

D. Write a function **print_bits_rev/1**, which given a non-negative integer prints its binary representation reversed, each digit in a separate line. The function should return **ok**.

Ex: print_bits(6) should return ok and print:
0
1
1

5. **List comprehensions –** In this problem you are supposed to implement the functions using list comprehensions.

A. Write a function **expand_circles/2**, which takes as arguments a positive integer **N**, and a list of pairs each consisting of the atom **circle** and a positive integer. The function returns a list of pairs of the same form as the list given as the second argument. The function's input represents circles of different radii. The function's output should be a list of equal length as the input list, where each element is a circle from the input list whose radius has been enlarged **N** times.

Ex: expand_circles(2, [{circle, 4}, {circle, 2}, {circle, 5}])
→ [{circle, 8}, {circle, 4}, {circle, 10}]

B. Write a function **print_circles/1**, which takes as argument a list of pairs each consisting of the atom **circle** and a positive integer. The function should print a line "Circle N" for each circle in the list, where N is the radius of the circle. Lines should be printed in the same order as the corresponding elements are in the list, and a trailing newline should also be printed. The function should return the atom **ok** after printing all lines.

Ex: print_circles([{circle, 3}, {circle, 2}, {circle, 4}])
→ ok (printing ”Circle 3~nCircle 2~nCircle 4~n”)

C. Write a function **even_fruit/1**, which takes as argument a list of pairs each consisting of an atom and a positive integer, and returns a list of atoms. The function’s input represents different kinds of fruit and their quantities (numbers). The function’s output should be a list of atoms representing fruit, with an atom for each fruit that has an even quantity in the input list. Fruit appearing many times in the input list should be treated as separate. You should use your **even_odd/1** function from assignment 1.

Ex: even_fruit([{orange, 3}, {apple, 2}, {banana, 4},
{orange, 2}, {leechee, 5}, {apple, 6}])
→ [apple, banana, orange, apple]

D. You are operating a small ferry that can carry two vehicles. There is a number of vehicles to be transported. However, due to weight restrictions not all pairs may be transported together. Write a function **ferry_vehicles/2**, which takes as arguments an integer **N** and a list of pairs of the form **{Vehicle, Weight}**, where **Vehicle** describes the kind of the vehicle and **Weight** is a positive integer describing its weight. The vehicle descriptions in the list are all unique. The function’s output should be a list of triples **{Vehicle1, Vehicle2, M}**, where **Vehicle1** and **Vehicle2** are vehicle descriptions from the input for two (different) vehicles, and **M** is their combined weight, which must be most **N**. All valid pairs of vehicles should be present in the output twice (also as **{Vehicle2, Vehicle1, M}**).

Ex: ferry_vehicles(3200, [{compact, 1500}, {crossover, 1900},
{mini, 1200}, {keicar, 800}, {minibus, 2200}])
→ [{compact,mini,2700}, {compact,keicar,2300},
{crossover,mini,3100}, {crossover,keicar,2700},
{mini,compact,2700}, {mini,crossover,3100},
{mini,keicar,2000}, {keicar,compact,2300},
{keicar,crossover,2700}, {keicar,mini,2000},
{keicar,minibus,3000}, {minibus,keicar,3000}]

(This is only one of possible results)

E. Write a function **ferry_vehicles2/2**, which takes the same arguments as **ferry_vehicles/2**, but returns a list of valid vehicle pairs (together with combined weights) without duplicate pairs.

Ex: ferry_vehicles(3200, [{compact, 1500}, {crossover, 1900},
{mini, 1200}, {keicar, 800}, {minibus, 2200}])
→ [{mini,compact,2700}, {mini,crossover,3100},
{mini,keicar,2000}, {keicar,compact,2300},
{keicar,crossover,2700}, {minibus,keicar,3000}]

(This is only one of possible results)