Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

**Übungen zur Vorlesung Informatik II (D-BAUG) FS 2017**

D. Sidler, F. Friedrich

http://lec.inf.ethz.ch/baug/informatik2/2017

**Solution to exercise sheet # 4** **13.3.2017 – 21.3.2017**

## Problem 4.1. Recursive Function Analysis

For the following two recursive functions:

(a)
```
boolean f (int n) {
    if (n == 0) {
        return false;
    }
    return !f(n-1);
}
```

(b)
```
void g (int n) {
    if (n == 0) {
        System.out.printf("*");
        return;
    }
    g(n-1);
    g(n-1);
}
```

i. Formulate pre- and postconditions.

ii. Show that the function terminates (under preconditions of i.).

iii. Determine the number of functions calls as (mathematical) function of parameter $n$.

**Submission link:** https://codeboard.ethz.ch/inf2baugex04t01

## Solution of Problem 4.1.

i. (a)
```
// PRE: n >= 0
// POST: return value is false if n is even and true if n is odd
```

(b)
```
// PRE: n >= 0
// POST: 2^n stars have been written to standard output
```

ii. (a) The function `f` immediately terminates for `n == 0`. With each recursive call `n` is decremented, i.e., `f` is called with parameter `< n`, eventually reaching `n = 0`.

(b) The function `g` immediately terminates for `n == 0`. With each recursive call `n` is decremented, i.e., `g` is called with parameter `< n`, eventually reaching `n = 0`. This is true for both recursive calls of `g`.

iii. Note: The following formulas include the first (non-recursive) functional call.

(a) $Calls_f(n) = n + 1$

(b) $Calls_g(n) = \sum_{i=0}^{n} 2^i = 2^{n+1} - 1$

## Problem 4.2. Money

In how many ways can you own CHF 10? Despite its somewhat philosophical appearance, the question is a mathematical one. Given some amount of money, in how many ways can you partition it using the available denominations (bank notes and coins)? Today's denominations in CHF are 1000, 200, 100, 50, 20, 10 (banknotes), 5, 2, 1(coins). Note we ignore values below CHF 1. The amount of CHF 4, for example, can be owned in four ways: $(2, 2), (2, 1, 1), (1, 1, 1, 1)$.

Solve the problem for a given input amount, by writing the following function.

```
// PRE: billsAndCoins is array of bill and coins values, start is an index into
    this array, the array is sorted such that billsAndCoins[i] >
    billsAndCoins[i+1] > .. > 0
// POST: return value is the number of ways to partition amount
```

```
//      using billsAndCoins beginning at index start
public static int partition(int amount,
                            int[] billsAndCoins,
                            int start);
```

To allow you to focus on implementing the recursive function `partition`, we provide you with the template code that implements all functionality except the said function. The input is the amount in CHF as `int` and the expected output is the number of ways that amount can be owned as `int`.

**ETH Codeboard link:** https://codeboard.ethz.ch/inf2baugex04t02

Once you have implemented the partition function, you can test your program by un-commenting the annotation `@RunTests`. If you pass the test you can submit your program.

## Solution of Problem 4.2.

The solution below implements the following idea: Suppose you owe an amount of CHF 5. Then you can either use a CHF 5 coin to directly pay the amount (1 way), or you can pay a first part using a smaller coin, let's say CHF 2, and you still owe CHF 3. Now you have to compute (recursively) the number of ways how CHF 3 can be partitioned.

There is a small twist: if you first pay back CHF 1, you still owe CHF 4. If you now simply compute the number of ways CHF 4 can be partitioned, you will also count partitions that use coins of CHF 2. Let's say the partition (1 $\times$ CHF 1, 1 $\times$ CHF 2, 2 $\times$ CHF 1). But this partition can also be realized by *first* paying CHF 2, and then 3 $\times$ CHF 1, thus it will be counted at least twice (once when you first pay CHF 2, and when you first pay CHF 1). In order to not count a partition twice, we only count partitions of the remaining amount that *do not* use coins that are larger than the *first used coin*. In this way, we will count the partition (1 $\times$ CHF 2, 3 $\times$ CHF 1) only once, namely when we first pay CHF 2 but not when we first pay CHF 1.

The number of ways in which you can own CHF 5 is 4, and CHF 10 can be owned in 11 ways. The above program becomes very slow for larger values, since during the recursive calls, many values are computed over and over again. For CHF 50, we already have to wait "forever". We can speed things up by using dynamic programming. We don't even have to change the structure of our function, but we provide it with an additional two dimensional array to store the values that have already been computed. Whenever we need a value, we first check whether it has already been computed, and only if this is not the case, we recursively call the function.

```
/**
 * Main class of the Java program.
 *
 * For TESTING and SUBMITTING: Uncomment the @RunTests annotation
 * (Remove the two slashes at the beginning of line ~10)
 *
 */
import java.util.Scanner;

//@RunTests
public class Main {

    public static int partition(int amount, int[] billsAndCoins, int start) {
        if (amount==0) {
            return 1;
        }
        int ways = 0;
        for(int i = start; i < billsAndCoins.length; i++) {
            if (amount >= billsAndCoins[i]) {
                ways += partition(amount - billsAndCoins[i], billsAndCoins, i);
```

```
            }
        }
        return ways;
    }

        public static void main(String[] args) {
            int[] billsAndCoins = {1000, 200, 100, 50, 20, 10, 5, 2, 1};
            System.out.println("In how many ways can I own x CHF for x=?");
            Scanner input = new Scanner (System.in);
            int amount = input.nextInt();
            int ways = partition(amount, billsAndCoins, 0);
            System.out.println(ways);
        }
}
```

## Problem 4.3. Throwing Eggs.

Suppose you are in a skyscraper with 100 floors and you want to find the lowest floor $f$ such that an egg breaks when you throw it out the window. So you get some eggs to conduct your experiments. If you throw an egg and it doesn't break, you may re-use it. However, if it breaks, it can't be thrown again. You make the reasonable assumption that if you throw an egg from floor $i$ and it breaks, an egg thrown from a higher floor $j > i$ will also break. So if an egg doesn't break when thrown from floor $i$, it also doesn't break when thrown from a lower floor $k < i$. Your task is to find a strategy to find the lowest floor from which a thrown egg breaks such that you need as few attempts as possible (you don't care about the number of eggs you break). However, when you run out of eggs, you should be able to name the requested floor.

(a) What would be your strategy if you would have an arbitrary number of eggs?

(b) What would you do if you only had one egg?

(c) What would be your strategy if you only had two eggs?

**Submission link:** https://codeboard.ethz.ch/inf2baugex04t03

## Solution of Problem 4.3.

(a) You can use binary search. Throw an egg from floor 50, if the egg breaks continue on floor 25, and on floor 75 otherwise. In the worst case, you need $\log_2(100)$ attempts.

(b) Clearly, you will have to start with the lowest floor and work your way upwards from one floor to the next until the egg breaks.

(c) If you have two eggs, you can use the first egg to find an interval of floors in which the requested floor lies. Within this interval, you use the same strategy as if you would have only one egg (as you have only one egg left). Let us say you are aiming to use no more than $s$ attempts to find the floor $f$. If you throw the egg from floor $s$ and it breaks, you need at most $s-1$ additional attempts to find the floor. If it does not break, you can continue with floor $s+(s-1)$. If the egg breaks there, the floor $f$ is between floor $(s+1)$ and floor $s+(s-1)-1$, so you need at most $s-2$ additional attempts to find the floor $f$ with the second egg. So if we choose $s$ wisely, we need not more than $s$ attempts in total. To get to the top of the skyscraper, we have to choose $s$ such that

$$s+(s-1)+(s-2)+\cdots+2+1=\sum_{i=1}^{n} i=\frac{s(s+1)}{2}\geq 100.$$

This is the case for $s=14$. The first egg is thus thrown out first of floor 14, then of floor $14+13=27$ and so on until it breaks. If it breaks after $i$ attempts, we only need at most $s-i$

additional attempts with the second egg. In total, we never need more than $s = 14$ attempts to find floor $f$ with only two eggs. In general, for a building with $n$ floors and only two eggs, you need $\mathcal{O}(\sqrt{n})$ attempts.