SOFTWARE 1 PRACTICAL

RECURSION

Week 8 – Additional Exercises

Exercise 1:

1. Write a recursive function `print_all(numbers)` that prints all the elements of list of integers, one per line (use no while loops or for loops). The parameters `numbers` to the function is a list of int.
2. Same problem as the last one but prints out the elements in reverse order.

Exercise 2: *from "Python Programming Fundamental"- Kent L, 2nd Ed. (2014)*

There is an elegant algorithm for converting a decimal number to a binary number. You need to carry out long division by 2 to use this algorithm. If we want to convert 83_{10} to binary, then we can repeatedly perform long division by 2 on the quotient of each result until the quotient is zero. Then, the string of the remainders that were accumulated while dividing make up the binary number. For example,

```
83/2 = 41 remainder 1
41/2 = 20 remainder 1
20/2 = 10 remainder 0
10/2 = 5 remainder 0
5/2 = 2 remainder 1
2/2 = 1 remainder 0
1/2 = 0 remainder 1
```

The remainders from last to first are 1010011_2 which is 83_{10}.

a) Implement a **recursive** function `to_binary(number)` that takes a positive integer as parameter and returns a binary representation of that number as a string.

b) Implement the reverse function `to_base10(binary)` that takes a string containing a binary number and returns its representation in base 10. The function **must** be **recursive**. To test your solution once implemented `to_base10(to_binary(83))` should return 83.

Exercise 3:

We are given a set of items, each with a weight and a value and we need to determine the number of each item to include in a collection (a bag for example) so that the total weight is less than or equal to a given limit and the total value is as large as possible. In addition, the items are indivisible; we can either take an item or not. Note that there might be more than one item with same value and same weight. For example,

```
Input:
value = [20, 5, 5, 10, 40, 15, 25]
weight = [1, 2, 2,  3,  8,  7,  4]
max_weight = 10

Output: 60
value = 20 + 40 = 60
weight = 1 + 8 = 9
```

The idea is to use **recursion** to solve this problem. For each item, there are two possibilities

1. We include current item in the bag and recur for remaining items with decreased capacity of the bag. If the capacity becomes negative, do not recur or return -INFINITY.

2. We exclude current item from the bag and recur for remaining items

Finally, we return maximum value we get by including or excluding current item. The base case of the recursion would be when no items are left, or capacity becomes 0.

Naïve Implementation:

Implement a naïve implementation that try all possible solution and returns the maximum.

Memoization

The time complexity of a naïve implementation of above solution is exponential $O(2^n)$ and auxiliary space used by the program is constant $O(1)$. However, the above solution has an optimal substructure, that is optimal solution can be constructed efficiently from optimal solutions of its subproblems. It also has overlapping subproblems, that is the problem can be broken down into subproblems which are reused several times (in the same way as computing the Fibonacci sequence F_2 was used multiple times). In order to reuse subproblems solutions, we can apply **Dynamic Programming**, in which subproblem solutions are **memoized** (this is the correct spelling by the way) rather than computed over and over again.

Try to find by yourself, how subproblem can be memoized rather than looking at the solution on the web. As a hint, I can tell you that the space needed to store the information is of size $n \times w$ where n is the number of items available, and w is the maximum weight allowed.

Exercise 4: *Text on 9 keys – T9*

T9, which stands for **Text on 9 keys**, is a predictive text technology for mobile phones, specifically those that contain a 3x4 numeric keypad as shown in Figure 1. T9's objective is to make it easier to type text messages. It allows words to be entered by a single keypress for each letter, as opposed to the multi-tap approach used in conventional mobile phone text entry, in which several letters are associated with each key, and selecting one letter often requires multiple keypresses.

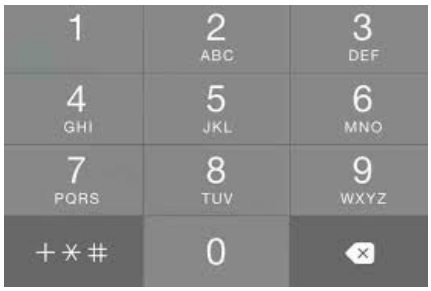

Figure 1: Example of a **3** × **4** numeric keypad used for T9

Given a sequence of numbers between [2-9], return a **set** of all possible combinations of words formed from the mobile keypad shown in Figure 1. The mobile keypad is store in a dictionary as shown below:

```
t9_keypad = {'2':['a','b','c'], '3':['d','e','f'],
             '4':['g','h','i'], '5':['j','k','l'],
             '6':['m','n','o'], '7':['p','q','r','s'],
             '8':['t','u','v'], '9':['w','x','y','z']}
```

Given the digits `'24'` the function `allwords(digits, keypad)` should return the set of 9 elements `{'AG', 'AH', 'AI', 'BG', 'BH', 'BI', 'CG', 'CH', 'CI'}`.

Exercise 3: *Coin game*

Alice and Bob are playing a game using a bunch of coins. The players pick several coins out of the bunch in turn. Each player can pick either **1**, **2** or **4** coins in each turn. Each time a player is allowed to pick coins, the player that gets the last coin is the winner. Assume that both players are very smart, and he/she will try his/her best to work out a strategy to win the game. It is known that at last, when **1**, **2** or **4** coins are left, the player with first turn will definitely pick 1, 2 or 4 coins as required and win. For example, if there are 2 coins in end and Alice is the first player to pick, she will definitely pick 2 coins and win. If there are 3 coins and Alice is still the first player to pick, no matter she picks 1 or 2 coins, Bob will get the last coin and win the game. Given the number of coins and the order of players (which means the first and the second players to pick the coins), you are required to write a program to calculate the winner of the game, and calculate how many different strategies there are for he/she to win the game.