Data Structures & Algorithms 2020/2021

Take-Home Exam

The goal of this take-home exam is to test your mastery of the topics taught in the first-year Basic Programming (Introduction to Programming) course. If you took this prerequisite course but did not pass it, yet you do pass this take-home exam, you are still allowed to take part in the Data Structures & Algorithms course. We will also use the results for this take-home exam as a guideline for the speed with which we provide new material.

This exam consists of 5 exercises, each worth 2 points, for a total of 10 points. During this test, you are allowed to consult the reading material for the course, such as the course book and the standard Python reference. You are not allowed to consult other sources, including sources you find on the internet or other people. The exam is meant to test to what extent *you* can write code. If you have questions on the exercises, you can ask them to the instructors, as long as it is about the descriptions of the exercises. The instructors will not help you create solutions.

Your code can make use of all the standard modules delivered with Python. You are not allowed to make use of modules that are not standard for Python, such as `numpy`.

Each question needs you to write a program. You get a template program file with the question. The template contains a function which you need to complete. You can test the function by simply running the code as normal. At the bottom of the file some test calls to the function are given. You are allowed (and encouraged) to change these to test your code even further. If you add any statements to test your code, you must add them to the `main()` function which is in the Python file. Do not add any code outside the `main()` function or any other functions, as we will load your code as a module to test it.

Make sure that you do not put any `print()` statements in the functions. You can do so during testing, but not in the final version that you submit. Such statements would invalidate our automated testing.

Make sure that every file that you submit contains your name and student number at the top of the file. If you add comments into your submissions, please write them in English.

You must upload your answers to the corresponding assignment on Canvas.

In principle, you should need less than 2.5 hours to solve all five exercises. If you need more than 3 hours for them, you should practice more, as you will need to get faster to be able to pass the final exam for the course.

Exercise 1: Powersum

File: `powersum.py`

Consider the following sum:

$S(n) = 1 + 2 + 2^2 + 2^3 + ... + 2^n$

where *n* is a non-negative integer.

Write a function `powersum()` that gets *n* as parameter and returns the value of *S*(*n*). You may assume that the parameter *n* is passed to the function as an integer. If a negative *n* is passed, return -1 to indicate that there is a mistake in the parameter.

Examples:

`powersum(3)` returns 15
`powersum(0)` returns 1
`powersum(-3)` returns -1

Exercise 2: Mastermind

File: `mastermind.py`

Master Mind is a game in which a player has to guess the secret code of another player. With each attempt at a guess, a hint is given. This hint is called the score of an attempt.

In this exercise, a secret code consists of four digits, which are integers between 0 and 9 (both 0 and 9 included). The digits are all different. An attempt to guess the code also consists of four digits in the same range.

The score of an attempt is a number of black pegs (represented by an X) and white pegs (represented by an O). If a number is correct and in the correct position, a black peg is scored. If a number is correct but not in the correct position, a white peg is scored.

Write a function `mastermind()` which scores an attempt. The function gets two parameters: the secret code, and the attempt. Both are strings with length 4, consisting of only digits (you do not need to check for this). You may assume that the digits in the secret code are all different, and the digits in the attempt are also all different. The function returns the score as a string consisting of X's and O's, with the X's to the left of the O's. The score will be at most length 4.

Examples:

Secret code: 1234
Attempt: 4321
Score: OOOO

Secret code: 1234
Attempt: 5678
Score:

Secret code: 1234
Attempt: 3524
Score: XOO

Secret code: 1234
Attempt: 7254
Score: XX

Exercise 3: Hotel

Files: `hotel.py`, `Registry1.txt`, `Registry2.txt`

A hotel keeps a registry of the guests that have booked a room. The registry is in the form of a text file. Each line in the text file contains the following items:

- Guest name (a string of letters, which may be capitals or lower case, and zero or more spaces)
- Number of the day that the guest arrives (an integer which is zero or higher)
- Length of stay (an integer which is 1 or higher)

The three items are separated by commas. There are no spaces next to the commas. For example, the contents of the file may look like this:

```
Frodo Baggins,7,6
Gandalf the Grey,9,10
Bilbo Baggins,12,3
```

Note that it would have been more logical in practice to store the day of arrival in the form of a date, but that would increase the complexity of the exercise considerably. Therefore a day is simply stored as an integer.

The function `occupied()` gets two parameters. The first is the name of a file, which contains the registry. The second is an integer, which indicates the number of a day. The function returns the number of guests which stay at the hotel at that particular day. If the file that is specified does not exist, you should return -1. You may assume that if the file exists, its contents are as specified above (i.e., each line is `<guest>,<arrival>,<length>`).

For example, if the function is called with the example registry given above and day number 10, then it should return 2, as at day number 10 both Frodo and Gandalf stay at the hotel, but Bilbo has not arrived yet. At day number 12 there will be three guests. At day number 19 there will be no guests.

Exercise 4: Stopwatches

File: `stopwatch.py`

You have a number of stopwatches. When you start a stopwatch, it counts seconds, starting at zero. Each of the stopwatches has a maximum number of seconds to which it can count, which may differ between the stopwatches. Instead of showing the maximum, a stopwatch will restart at zero when it reaches the maximum. For instance, a stopwatch with maximum 100 will count from zero to 99, but after 100 seconds it is back at zero again, and continues counting from there.

In this exercise you have a series of stopwatches with different maxima. The question is: if you start all of them at the same time, after how many seconds is the first time that all of them are showing zero again? For instance, if you have three stopwatches with maxima 6, 10, and 15, all of them will be showing zero again after 30 seconds (because 30 divided by each of these maxima gives zero).

The function `stopwatch()` gets as parameter a list of integers. The list contains at least one element. All integers are 1 or higher. Each of these integers represents the maximum of a stopwatch. The function returns the number of seconds after which all the stopwatches show zero again for the first time, if they would all be started at the same time.

For instance, `stopwatch([10,12,15])` returns `60`.

Note: For this exercise it makes a lot of sense to use the modulo operator (%).

Notes on algorithms: The simplest algorithm will simulate the counting of the stopwatches until all of them show zero again. This is easy to implement but may be slow. It is perfectly fine as a solution, though. There are two more approaches which you may consider (you may skip the rest of this description if you are thinking “thanks but no thanks, I will stick with the simple algorithm”):

The first is to turn each maximum into a list of its prime factors (this is called “factorization”). The prime factors are the prime numbers which can be multiplied to get the original number, e.g., the prime factors of 228 are 2, 2, 3, and 19, as 228 = 2 * 2 * 3 * 19. Then create a list which contains all the prime factors which the maxima have in common. For instance, if the maxima are 10, 12, and 15, the prime factors are, respectively [2,5], [2,2,3], and [3,5]. The list of common prime factors is therefore [2,2,3,5]. If you then multiply the prime factors of this common list, you get the requested answer.

The second is even a lot faster and also a lot easier to implement, but it requires recursive programming. You have to realize that the Least Common Multiple (LCM) of three numbers is the same as the LCM of the third number and the LCM of the first two numbers. Moreover, you can calculate the LCM of two numbers by multiplying them and dividing them by their Highest Common Factor (HCF), i.e., `lcm(x,y) = x*y/hcf(x,y)`. You can calculate the HCF using Euclid’s algorithm, which says that the HCF of two numbers `x` and `y` whereby `x <= y` is calculated as follows: if `y%x` is zero then it is `x`, otherwise it is the HCF of `y%x` and `x`.

Exercise 5: Highest column

File: `highestcolumn.py`

The function `highestcolumn()` gets as parameter a two-dimensional list, representing a matrix of *M* rows and *N* columns (*M* and *N* being positive integers). The matrix contains only positive integers. For instance, a matrix of 3 rows and 5 columns, of which all the cells contain the number 7, would be represented by the following list:

```
[ [7,7,7,7,7], [7,7,7,7,7], [7,7,7,7,7] ]
```

You may assume that the matrix with which the function is called meets the specifications given above. Add up the highest value from each of the columns of the matrix, and return the resulting sum.

For example, if the matrix is:

```
1 5 8 2 5
3 6 9 1 4
1 5 1 4 3
9 2 6 3 1
```

it is represented by the list:

```
[ [1,5,8,2,5], [3,6,9,1,4], [1,5,1,4,3], [9,2,6,3,1] ]
```

and the function should return 9+6+9+4+5 = **33**