Assignment 0

CSCI E-118 Introduction to Blockchain and Bitcoin

Directions

What follows is a set of exercises utilizing the Python programming language. Complete problem 1 and 2. Then complete any **one** of problems 3 through 5. Note that Problem 1 has 3 parts. All excercises should be completed using Python 3. Create a separate `.py` file for each problem. In other words, you should submit exactly 3 `.py` files. Name each `.py` file as `problem_n.py` , where n is the problem number.

Each `.py` file should have:

```
if __name__ == '__main__':
    main()
```

Where `main()` is a function that demonstrates the functionality of the program.

Problem 1

1A

In the following example, we compute the factorial function recursively:

```
def factorial(n):

    if n < 2:
        return 1

    return n * factorial(n - 1)
```

The **Fibonacci** sequence consists of 1's as the first and second elements of the sequence. Every other element is the sum of the previous two elements in the sequence: thus the two 1's are followed by $1 + 1 = 2$ followed by $1 + 2 = 3$, and so on.

Therefore the first 9 elements of the sequence are:

```
1, 1, 2, 3, 5, 8, 13, 21, 34
```

Similar to the factorial function defined above, write a recursive function which returns the `n` th Fibonacci number.

1B

We can use a rudimentary test to ensure the `factorial` function behaves as expected:

```
first_factorials = [1, 1, 2, 6, 24, 120, 720, 5040]

def test_factorials(first_factorials, factorial):

    for index, ele in enumerate(first_factorials):
        assert( first_factorials[index] == factorial(index) )

    print ("All tests passed with no errors")

test_factorials(first_factorials, factorial)
```

Write a similar test checking the first few Fibonacci numbers.

1C

The following caches, or *memoizes*, a few values of the `factorial` function:

```
class FactorialMemo:

    def __init__(self):
        self.memo_dict = {0:1, 1:1}

    def factorial(self, n):
        if n in self.memo_dict:
            return self.memo_dict[n]

        self.memo_dict[n] = n * self.factorial(n - 1)

        return self.memo_dict[n]
```

Note that *memoizing* values of the factorial function results in the computer taking less time (about half) to compute `factorial(50)` the **second** time the function is run:

```
import time

factorial_memo = FactorialMemo()

# Timing the factorial function

# First run
first_run_start = time.time()
factorial_memo.factorial(50)
first_run_end = time.time()
time_delta_1 = first_run_end - first_run_start

# Second run
second_run_start = time.time()
factorial_memo.factorial(50)
second_run_end = time.time()
time_delta_2 = second_run_end - second_run_start

# Comparing the time it took to run the factorial function the first time
```

Problem 2

Write a class called `AverageStream` which, when instantiated, accumilates numbers and outputs an average of all the numbers ever input into it. Thus:

```
avg_inst = AverageStream()
avg_inst.add_element(5)
avg_inst.get_average()
```

would return 5. Then:

```
avg_inst.add_element(2)
avg_inst.get_average()
```

would return 3.5. And so on...

Write the class such that it does not keep a very large list stored within the object. In other words, if you call `add_element()` 10 times, there **should not** be a list of size 10 containing each of the input numbers.

Problem 3

Write a function (or class) that computes the powerset of the first n integers. For instance, the `powerset(n)` function returns the set of all sets of integers 1 to n.

For example `powerset(4)` should return the set of all sets of the set `{1,2,3,4}`:

```
{}, {1}, {2}, {3}, {4}, {1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4}, {3, 4}, {1, 2, 3}, {2, 3, 4}, {1, 3, 4}, {1, 2, 4}, {1, 2, 3, 4}
```

The `powerset(n)` function should return the 2^n elements in the powerset of the set `{1,2,...n}`.

Note that the null set, `{}` must be included.

Problem 4

In a text file of your creation, count the frequency of each word in the file.

In the output, include the line number each occurrence of the word appeared.

The text file should have multiple lines, where one or more lines are empty, and the rest contain one or more words per line. The text file may be created such that it contains no punctuation or special characters. Ignore casing, either by having the code treat upper and lower case letters as identical, or by only using lower case words in the text file.

Problem 5

Check if a string is a palindrome: which means it reads the same forwards and backwards.

For example the string `'abcdcba'` is palindromic.

Write a program to check if a two-dimensional array of characters reads the same in any direction.

For example: `aaaa\n abba\n abba\n aaaa\n` reads the same forwards or backwards from left-to-right, up-to-down.

Note, the array rows are separated by `'\n'`.

The program should output `True` only if every row and every column is palindromic.

The two-dimensional array need not be symmetric. In other words column 1 does not have to match row 1, and so on.

For simplicity, the sizes of the rows should all match, and the sizes of the columns should all match. But the row size does not need to equal the column size

Create two text files. Both text files should have n rows and m columns. Keep n and m reasonably sized so that a human can easily tell at a glance whether the rows and columns are palindromic. The first text file should be palindromic. The second should not be palindromic.
