Handout for Mastery II — Intro to Programming (T. III/17–18)

Directions:

- This mastery exam is scheduled for **Thursday July 19, 2018**.
- The exam will have a total of 4 problems. Two of them will come from problems in this handout. That is to say, only two out of these six problems below will be chosen.
- During the exam, no collaboration of any kind whatsoever is permitted. The exam is closed books, closed notes, etc. You cannot use the Internet. But we'll hand out a "Python Quick Reference" cheat sheet.
- You can work together with other students to prepare for the exam. However, the course staff will only help to clarify the problems but will not offer any help to solve them.

Problem 1: Inside Zeros

As you already know, trailing zeros are zeros at the end of a number, in the right-most digits. The number 102400, for example, has a total of 3 zeros—1 inside and 2 trailing.

Remember also that for $N \geqslant 0$, $N! = N \times (N-1)!$ where $0! = 1$. That is why $5! = 120$.

In this problem, you are to implement a function `zeroInsideFac(n)` that takes a non-negative integer n and returns an **int** representing the number of zeros inside the value of $n!$.

Here are some examples:

- `zeroInsideFac(5)` should return 0 because $5! = 120$ contains no inside zeros.
- `zeroInsideFac(16)` should return 1 because $16! = 20922789888000$ has *one* inside zero.
- `zeroInsideFac(37)` should return 4.

Problem 2: Secret Code

A secret code is hidden inside a long string. Your task is to figure out what it is! The secret code is a sequence of numeric characters such as `'01728'`, `'812552'`. The secret code is hidden in another string which contain only valid ascii characters without any white spaces.

Luckily, we know how to recover the code. The code can be recovered by identifying all of *the valid code characters*. Each valid code character must have all of the the following properties:

- Must be a numeric character i.e. `0 - 9`
- Must be preceeded by EXACTLY 2 uppercase letters i.e. `A - Z`
- Must be followed by EXACTLY 2 lowercase letters i.e. `a - z`

By identifying all code characters, we can obtain the complete secret code by simply concatenating them! For example, given the long string `'TT4za7GH5xpNHD1tyu8XQ9vg'`, we find that the secret code is `'459'`. Note that `'1'` is not a valid code character because it has more than 2 uppercase letters right in front.

In this problem, you are to implement a function `secretCode(st)` that takes a string `st` and returns the secret code `string`.

Here are some examples.

- `secretCode('TT4za7GH5xpNHD1tyu8XQ9vg')` should return `'459'`
- `secretCode('AAAAAAA4aaaaaBB5bb')` should return `'5'`
- `secretCode('XX4xx6kk')` should return `'4'`
- `secretCode('MMM555MMM')` should return `''`

Problem 3: Follow Me Not

In a prototype for your Twitter-killer startup, you're implementing a class called `User` that represents a user in your system. The `User` class can be created without any parameter, e.g., `user1 = User()`.

The main goal of the `User` class is to maintain a user's followers. A user can follow many users but not herself. A user can be followed by multiple users but not herself. If A follows B, it does *not* mean that B necessarily follows A. To keep track of the follow relationship, if `u` is an instance of `User`, it supports three class functions:

- `u.followedBy(that_user)` registers the fact that `that_user` is following `u`. Keep in mind, `that_user` is also an instance of `User`. If `that_user` is already following `u`, this call has no effects. In any case, this class function returns nothing.
- `u.unfollowedBy(that_user)` registers the fact that `that_user` has stopped following `u`. Keep in mind, `that_user` is also an instance of `User`. If `that_user` isn't following `u`, this call has no effects. In any case, this class function returns nothing.
- `u.isFollowedBy(that_user)` returns a Boolean value indicating whether `that_user` is following `u`.

This is best illustrated with a program snippet:

```
userA = User()
userB = User()
userC = User()
assert(userA.isFollowedBy(userB)==False)
userA.followedBy(userB)
userC.followedBy(userB)
assert(userA.isFollowedBy(userB)==True)
assert(userC.isFollowedBy(userB)==True)
assert(userC.isFollowedBy(userA)==False)
userA.unfollowedBy(userB)
assert(userA.isFollowedBy(userB)==False)
```

Problem 4: Ham's Apple Tree

Given a NxN board. Ham, represented by `'H'`, is located somewhere on this board. Ham's apple tree, represented by `'T'`, is also located somewhere on the board. All other cells on the board are marked empty by `'O'`. Ham wants to get to his tree, but he wants to save as much energy as possible. Basically, he wants to *minimize* the number of moves in order to get to the tree. In each move, he can only get to an adjacent cell. For example, if he is located at $(1,1)$, a single move can get him to either $(1,0)$, $(0,1)$, $(2,1)$, or $(1,2)$.

Your task is to write a function `computeMove(board)` to find out the number of moves Ham has to make to get to his apple tree.

```
>> computeMove([
  ['O','O','H'],
  ['T','O','O'],
  ['O','O','O']
  ])
3
>> computeMove([
  ['H','O','O','O'],
  ['O','O','O','O'],
  ['O','O','O','O'],
  ['O','O','O','T']
  ])
6
```

Problem 5: Abba

Miss Abba is obsessed with two characters `'A'` and `'B'`. She has spent a lot of time writing out strings that consist of only `'A'` and `'B'`. These are called *AB strings*. Here are some excerpts:

```
'AABB'
'ABBA'
'BAABB'
```

As she experiments with longer strings, it quickly dawns on her that she cannot write all of them out by hand. So she asks for your help. Your task is to write a function `generateAB(n)` that takes an integer $n \geqslant 1$

and returns a list of all AB strings of length precisely n. Your function must return the list *lexicographically sorted*—i.e., it should pass the following test: **sorted**(generateAB(n))== generateAB(n).

Examples:

```
>> generateAB(3)
['AAA', 'AAB', 'ABA', 'ABB', 'BAA', 'BAB', 'BBA', 'BBB']
>> generateAB(2)
['AA', 'AB', 'BA', 'BB']
>> generateAB(1)
['A', 'B']
```

Problem 6: Anagram With A Twist

As you already know, an *anagram* of a word w is a word formed by rearranging the letters of the word w, such as `iceman` formed from `cinema`. Similarly, `angel` is an anagram of `glean`. In this problem, you will implement a function `isAnagram(word1, word2, intab, outtab)` that checks if two given words are anagrams of each other after performing the following substitution:

Unlike our previous anagram problem, in addition to the two input words, you will also be given a substitution table. This will be given in the form of two strings `intab` and `outtab`, mapping each character in the `intab` string into the character at the same position in the `outtab` string. For example, if we use `intab = "aeiou"` and `outtab = "12345"`, you will substitute every occurrence of `'a'` in the input to a `'1'`, every `'e'` to a `'2'`, and so on. Hence, with this pair of `intab` and `outtab`, the string `'mooshu'` becomes `'m44sh5'`.

As another example (and importantly), if we use `intab = "aeiou"` but `outtab = "uaeio"`, then the string `'mooshu'` becomes `'miisho'`.

Therefore, your `isAnagram(word1, word2, intab, outtab)` function will perform the substitution on `word1` and the same substitution on `word2`, and check if the resulting words are anagrams of each other. It returns a Boolean value. Keep in mind that even though `word1` and `word2` may not be anagrams initially, they may end up being anagrams after substitution.

For example, `isAnagram('live', 'eval', 'aw', 'il')` will first make substitution on `'live'` (which gives the same string) and make substitution on `'eval'` (which yields `'evil'`). After that, we test if `'live'` and `'evil'` are anagrams. Because they are, we return `True`.