CMSC 201 Section 40
Homework 6
Recursion

Due Date: Monday, April 18, 2022 by 11:59:59 PM
Value: 40 + 10 points
Each problem is worth 10 points, with the ability to earn up to 10 extra credit points on this assignment.

For any part that requires recursion, if you do not create a recursive function, that will result in a 0 on that part.

This assignment falls under the standard cmsc201 academic integrity policy. This means you should not discuss/show/copy/distribute your solutions, your code or your main ideas for the solutions to any other student. Also, you should not post these problems on any forum, internet solutions website, etc.

Make sure that you have a complete file header comment at the top of each file, and that all of the information is correctly filled out.

```
"""
File:     FILENAME.py
Author:   YOUR NAME
Date:     THE DATE
Section:  YOUR DISCUSSION SECTION NUMBER
E-mail:   YOUR_EMAIL@umbc.edu
Description:
  DESCRIPTION OF WHAT THE PROGRAM DOES
"""
```

Submission Details

Submit the files under the following titles:
(These are case sensitive as usual.)

submit cmsc201 HW6 rec_range.py catalan.py slice_replace.py count_gold.py ab_consecutive.py

Problem 1 - Recursive Range	rec_range.py
Problem 2 - Catalan Numbers	catalan.py
Problem 3 - Slice Replace	slice_replace.py
Problem 4 - Count The Gold	count_gold.py
Problem 5 - AB Consecutive	ab_consecutive.py

Coding Standards

Coding standards can be found here.

We will be looking for:

1. At least one inline comment per program explaining something about your code.
2. Constants above your function definitions, outside of the "if __name__ == '__main__':" block.
    a. A magic value is a string which is outside of a print or input statement, but is used to check a variable, so for instance:
        i. `print(first_creature_name, 'has died in the fight. ')` does not involve magic values.
        ii. However, `if my_string == 'EXIT':` exit is a magic value since it's being used to compare against variables within your code, so it should be:

```
EXIT_STRING = 'EXIT'
...
if my_string == EXIT_STRING:
```

    b. A number is a magic value when it is not 0, 1, and if it is not 2 being used to test parity (even/odd).
    c. A number is magic if it is a position in an array, like my_array[23], where we know that at the 23rd position, there is some special data. Instead it should be
    USERNAME_INDEX = 23
    my_array[USERNAME_INDEX]
    d. Constants in mathematical formulas can either be made into official constants or kept in a formula.
3. Previously checked coding standards involving:
    a. snake_case_variable_names
    b. CAPITAL_SNAKE_CASE_CONSTANT_NAMES
    c. Use of whitespace (2 before and after a function, 1 for readability.)

Allowed Built-ins/Methods/etc

- Declaring and assigning variables, ints, floats, bools, strings, lists, dicts.
- Using +, -, *, /, //, %, **; +=, -=, *=, /=, //=, %=, **= where appropriate
- Comparisons ==, <=, >=, >, <, !=, in
- Logical and, or, not
- if/elif/else, nested if statements
- Casting int(x), str(x), float(x), (technically bool(x))
- For loops, both *for i* and *for each* type.
- While loops
    - sentinel values, boolean flags to terminate while loops
- Lists, list(), indexing, i.e. my_list[i] or my_list[3]
    - 2d-lists if you want them/need them my_2d[i][j]
    - Append, remove
    - **list slicing**
- If you have read this section, then you know the secret word is: createous.
- String operations, concatenation +, +=, split(), strip(), join(), upper(), lower(), isupper(), islower(), rjust(), ljust()
    - **string slicing**
- Print, with string formatting, with end= or sep=:
    - '{}'.format(var), '%d' % some_int, f-strings
    - Really the point is that we don't care how you format strings in Python
    - Ord, chr, but you won't need them this time.
- Input, again with string formatting in the prompt, casting the returned value.

- **Dictionaries**
  - creation using dict(), or {}, copying using dict(other_dict)
  - .get(value, not_found_value) method
  - accessing, inserting elements, removing elements.
- Using the functions provided to you in the starter code.
- Using import with libraries and specific functions **as allowed** by the project/homework.
- **Recursion - required, so allowed for HW6**

Forbidden Built-ins/Methods/etc

This is not a complete listing, but it includes:

- break, continue
- methods outside those permitted within allowed types
    - for instance str.endswith
    - list.index, list.count, etc.
- Keywords you definitely don't need: await, as, assert, async, class, except, finally, global, lambda, nonlocal, raise, try, yield
- The *is* keyword is forbidden, not because it's necessarily bad, but because it doesn't behave as you might expect (it's not the same as ==).
- built in functions: any, all, breakpoint, callable, classmethod, compile, exec, delattr, divmod, enumerate, filter, map, max, min, isinstance, issubclass, iter, locals, oct, next, memoryview, property, repr, reversed, round, set, setattr, sorted, staticmethod, sum, super, type, vars, zip
- If you have read this section, then you know the secret word is: argumentative.
- exit() or quit()
- If something is not on the allowed list, not on this list, then it is probably forbidden.
- The forbidden list can always be overridden by a particular problem, so if a problem allows something on this list, then it is allowed for that problem.

Problem 1 - Recursive Range

Create a ***recursive*** function that simulates the range functionality in a file called rec_range.py.

However we should simplify a little bit. Instead of creating a "range object" we'll just return a list.

Because using a for loop and range would be cheating, we'll use recursion to generate the required list.

When we call rec_range, we'll return a list of the proper elements as if the range had been called.

The function should have the following parameters (you can change the names but the usage should be the same):

```
def rec_range(start, stop, step):
```

For instance rec_range(1, 5, 1) will return [1, 2, 3, 4]. We will use all three parameters, start, stop and step.

As another example rec_range(2, 17, 3) will return [2, 5, 8, 11, 14].

Given how difficult recursion is, let's try to do this more like a guided exercise.

First, let's think about the base case. What happens when start => end and step > 0 or start <= end and step < 0? Well there's nothing left to return, so we should return an empty list.

Now, we should think about the recursive case, where we call rec_range again. Basically we should call rec_range with a new start parameter. Think about it this way:

rec_range(0, 5, 1) = [0] + rec_range(1, 5, 1)
rec_range(1, 5, 1) = [1] + rec_range(2, 5, 1)
rec_range(2, 5, 1) = [2] + rec_range(3, 5, 1)
rec_range(3, 5, 1) = [3] + rec_range(4, 5, 1)
rec_range(4, 5, 1) = [4] + rec_range(5, 5, 1)
rec_range(5, 5, 1) = [] (base case)

Then we can unwind, so that:

rec_range(0, 5, 1) = [0] + [1, 2, 3, 4] = [0, 1, 2, 3, 4]
rec_range(1, 5, 1) = [1] + [2, 3, 4]
rec_range(2, 5, 1) = [2] + [3, 4]
rec_range(3, 5, 1) = [3] + [4]
rec_range(4, 5, 1) = [4] + []
rec_range(5, 5, 1) = []

Use these hints to build the rest of the function.

I used this code for the test driver:

```
if __name__ == '__main__':
    string_list = input('Enter the range start, stop, step separated by
spaces: ').split()
    int_list = []
    for s in string_list:
        int_list.append(int(s))

    start, stop, step = int_list
    print(rec_range(start, stop, step))
```

Sample Output

```
linux5[201]% python3 rec_range.py
Enter the range start, stop, step separated by
spaces: 0 5 1
[0, 1, 2, 3, 4]

linux5[202]% python3 rec_range.py
Enter the range start, stop, step separated by
spaces: 2 17 3
[2, 5, 8, 11, 14]

linux5[203]% python3 rec_range.py
Enter the range start, stop, step separated by
spaces: 10 0 -2
[10, 8, 6, 4, 2]

linux5[204]% python3 rec_range.py
Enter the range start, stop, step separated by
spaces: 1 10 -1
[]

linux5[205]% python3 rec_range.py
Enter the range start, stop, step separated by
spaces: 2 100 2
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28,
30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54,
56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80,
82, 84, 86, 88, 90, 92, 94, 96, 98]
```

Problem 2 - Catalan Numbers

Write a ***recursive*** function which calculates the sequence of Catalan numbers (in a file called catalan.py):

```
def catalan(n):
```

The recursive definition of the Catalan numbers is as follows:

$$C_{n+1} = \frac{2(2n+1)}{n+2} C_n$$

where the zeroth term $C_0 = 1$.

Think about what the base case is and what the recursive case will be.

However, when you're in the function you'll be trying to calculate Cn rather than C_{n+1} so we should modify the original equation to decrease the index by one.

$$C_n = \frac{2(2n-1)}{n+1} C_{n-1}$$

Another hint: be very careful about your order of operations. For instance,

`2 * (2 * n - 1) // (n + 1) * c_n_minus_one`

is not the same as:

`(2 * (2 * n - 1) * c_n_minus_one) // (n + 1)`.

Test Driver and Starter Code

The test driver code is:

```
if __name__ == "__main__":
    for i in range(0, 20):
        print(i, catalan(i))
```

Sample Output

```
linux5[109]% python3 catalan.py
0 1
1 1
2 2
3 5
4 14
5 42
6 132
7 429
8 1430
9 4862
10 16796
11 58786
12 208012
13 742900
14 2674440
15 9694845
16 35357670
17 129644790
18 477638700
19 1767263190
```

Problem 3 - Slice Replace

Create a ***recursive*** function in a file called slice_replace.py:

```
def slice_replace(big_string, find_string, replace_with):
```

We want to write a replace function which takes the big_string and replaces any time we find the find_string with the replace_string then returns it.

For instance slice_replace('hello ella', 'el', 'x') should return 'hxlo xla'.
slice_replace('abababab', 'ab', 'c') should return 'cccc'.

Hint: You can use slices to do string comparisons.

Hint: You can also slice the big_string. Either you'll be slicing the first letter off, or the len(find_string) letters.

Sample Output

```
linux5[109]% python3 slice_replace.py
Enter the string to use: recursion is fun
Enter the string to find: s
Enter the string to replace it with: v
recurvion iv fun

linux5[110]% python3 slice_replace.py
Enter the string to use: abcdababcd
Enter the string to find: ab
Enter the string to replace it with: hello
hellocdhellohellocd

linux5[111]% python3 slice_replace.py
Enter the string to use: abababab
Enter the string to find: ab
Enter the string to replace it with: c
cccc
```

Problem 4 - Gold Problem

Create a recursive function in a file called count_gold.py

Let's search a grid and count up all of the gold that we find. Not all of the gold is always accessible from the starting location. Here's an example of a map:

	*	*	G1			G8	G2
G1			G6			*	
						*	
*			*		*		G9
	G2		*	G3			
		G3			*		
	*					*	
		G7					G3

If you call create_map with a seed value of 234 and 8 and 8 for rows and columns then you will get the same map.

You will start at the position [0,0] represented in green. You must search through all of the positions using a recursive algorithm which searches in all four directions (no diagonal movement is allowed).

If you visit a position, you should add up the amount of gold at that position.

You must mark positions as visited and not return to them otherwise you'll find yourself with a RecursionError caused by the infinite

recursion. You could use a visited list instead to track positions where you have been instead of replacing the positions. Sample code for pathfinding is on the github under the recursion folder.

		G3	
*	*	G9	
G5	*	G5	
		*	G7

Note that the G5 marked in red is not accessible if you start at (0,0). Therefore the total is 3 + 9 + 5 + 7 = 24

Starter Code

The starter code with a create_map function, the main driver code, and the display_map function is available at:

/afs/umbc.edu/users/e/r/eric8/pub/cs201/spring22/count_gold_start.py

Copy it with the command:
cp /afs/umbc.edu/users/e/r/eric8/pub/cs201/spring22/count_gold_start.py count_gold.py

Sample Output

```
linux5[109]% python3 spider_web.py
Enter seed rows cols separated by spaces: 1234 10 10
  _  _  _ G9 G2  _  *  _  *  _
  _  *  _  _  _  _  _  _  _  _
 G6  _  _  _  _  _  _  _  _  *
  _  _  _ G5  _  _  _  _  _  _
  _  _  _  _  _  _  _ G8  * G2
  _  _  _  _  *  _  _ G5  _  *
  _  _  _  _  *  _  *  _  * G2
  _  *  *  _  _  _  _  _  _  *
  _  *  *  *  _  *  *  _ G7  _
  _  *  *  *  _  _  _  _  _  _
Counting the gold...
  V  V  V  V  V  V  *  V  *  V
  V  *  V  V  V  V  V  V  V  V
  V  V  V  V  V  V  V  V  V  *
  V  V  V  V  V  V  V  V  V  V
  V  V  V  V  V  V  V  V  *  V
  V  V  V  V  *  V  V  V  V  *
  V  V  V  V  *  V  *  V  * G2
  V  *  *  V  V  V  V  V  V  *
  V  *  *  *  V  *  *  V  V  V
  V  *  *  *  V  V  V  V  V  V
The total value of gold we can reach is:  44



linux5[110]% python3 spider_web.py
Enter seed rows cols separated by spaces: 159 8 3
  _  *  _
  *  _  _
  *  _  _
  *  _  _
     *  _
  _  _  _
  _  *  _
  _  _  _
  *  _  _
```

```
Counting the gold...
  V  *  _
  *     _  _
  *     _  _
  *     _  _
        *  _
  _        _
  _  _     _
     *     _
  _        _
  *     _  _
The total value of gold we can reach is:  0

linux5[111]% python3 spider_web.py
Enter seed rows cols separated by spaces: 456 7 10
    G9    _  *  *  _  *  * G3    _
 G1       _  _  _  _  _  _  _    *
  * G1    _  * G1  *  _  _  _ G1
  *  *  *     *     _  *  _  *
     _  G5  *  *  *  *  _  *  _ G5
  _                    * G3    _  *
  _  _  _  _  _           _  _
 G4  *     _  _  *  _  _  _  *    _
Counting the gold...
  V  V  V  *  *  V  *  *  V  V
  V  V  V  V  V  V  V  V  V  *
  *  V  V  *  V  *  V  V  V  V
  *  *  *     *  V  V  *  V  *
     _
  _  G5  *  *  *  *  V  *  V  V
                    *  V  V  V  *
  _  _  _  _  _
 G4  *     _  _  *  V  V  V  *    _
The total value of gold we can reach is:  24
```

Problem 5 - AB Consecutive

Create a ***recursive*** function in a file called ab_consecutive.py:

```
def ab_consecutive(n, allowed_cons, current_cons, current):
```

Print out all of the strings of a's and b's of length n so that there are not more than allowed_cons consecutive a's or b's.

For instance, if we call ab_consecutive(5, 2, 0, '') then we will see aabba printed out but aaabb is not allowed since there are three a's in a row. Similarly, abbba is not allowed either for the same reason, but ababa is allowed.

For this problem, start by looking at the a's and b's code on the github. You should consider the following cases:

1) The base case when the length n is 0
2) What happens when we are at the start of the string, so there is no previous letter?
3) When there is a previous letter, i.e. current[-1] exists, what happens when the previous letter is 'a', and then what happens when it's 'b'?

You can print out the strings, you don't have to return anything from your recursion.

current_cons should always start from 0, and current should be the empty string when you call the function from the main block.

I used the driver code:

```
if __name__ == '__main__':
    n, max_allowed = [int(x) for x in input('Enter n and
max allowed consecutives separated by spaces:
').split()]
    ab_consecutive(n, max_allowed, 0, '')
```

Sample Output

```
linux5[281]% python3 ab_consecutive.py
Enter n and max allowed consecutives separated by spaces:
5 2
aabba
aabaa
aabab
abbaa
abbab
abaab
ababb
ababa
bbaab
bbabb
bbaba
baabb
baaba
babba
babaa
babab

linux5[282]% python3 ab_consecutive.py
Enter n and max allowed consecutives separated by spaces:
3 1
aba
bab
```

```
linux5[283]% python3 ab_consecutive.py
Enter n and max allowed consecutives separated by spaces:
4 3
aaab
aabb
aaba
abbb
abba
abaa
abab
bbba
bbaa
bbab
baaa
baab
babb
baba
```