Final Project
Multiprocessing

Problem

You are tasked with creating a Python program capable of deciphering text that has been encrypted using a customized version of a Vigenère cipher to obfuscate the original text. Upon receiving the encrypted string, your application will build a matrix of size L x L (where L is the length of the encrypted string) and then process this matrix through 100 iterations of a dynamic programming algorithm designed to constantly transform the values across the matrix. Using the sum of the values for each column of the 100th iteration, your code will then perform a series of rotational algorithms to determine what the correct letter originally was. Your solution must make use of multiprocessing in Python to receive full credit.

Guidance and Rules

This section outlines the guiding principles for your decryption application and suggests a sequential framework for tackling the problem. Your approach should be divided into two distinct phases: the serial computation phase and the concurrent computation phase. Each phase is broken into one or more stages and after completing each stage, it is highly recommended to rigorously test your program before proceeding to the subsequent one.

Phase 1 – Serial Computation

Stage 1.1 – Data Retrieval

Your solution should begin by parsing and verifying the command line arguments as they are described in the *Command* Line Arguments and Examples section on page 5. Once it has verified the command line arguments, your application should read in the string written into the input file given by the command line argument '-i'. This file will follow the specification as described in the *Input File Specification* section on page 6.

Stage 1.2 – Matrix Generation

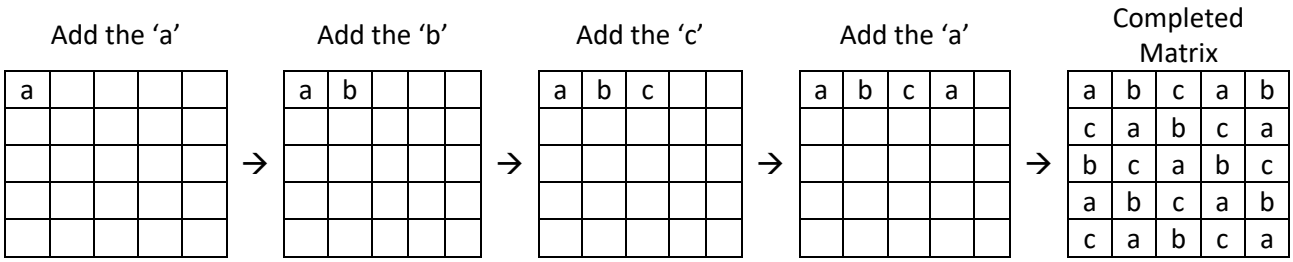
Your solution should first generate a matrix with dimensions L x L, where L corresponds to the length of the string obtained from the input file in Stage 1.1. Next, using the seed string supplied via the '-s' command-line argument, populate the matrix according to the following rules:

1. **Initialize Your Matrix**
    a. Start by creating an empty matrix with L rows and L columns, where L is the length of the string located in the input file retrieved in Stage 1.
    b. Once generated, your matrix will have a total of L^2 cells to fill.
2. **Fill the Matrix**
    a. Begin at the top-left cell of your matrix and proceed to fill the matrix with the characters from the seed string provided to your application by the '-s' command line argument.
    b. Place each character in the cells moving from left to right across the top row.

3. **Continue Filling Rows**
    a. Once you reach the end of the first row, move down to the first cell of the next row and continue filling in the same left-to-right pattern.
4. **Repeat the String**
    a. If you reach the end of seed string before filling all the cells, start again with the first character of seed string.
    b. Continue this process until every cell in the matrix is filled.
5. **Consider the Whole Matrix**
    a. Make sure you view the matrix as a continuous loop for the string, wrapping around when the end of the string is reached and resuming from its start.

Stage 1.2 Example

If you assume the input file contained the string "Cbhij" (Length L = 5) and your application received the seed string "abc", then your application should generate the following 5x5 matrix.



Stage 1.3 – Matrix Processing

Using the matrix you generated in stage 1.2, your solution should then perform the next 100 steps of a simulation that performs the following during each step:

- For each cell of the matrix, sum up the neighboring cells using the following rules:
    - Neighboring cells containing an 'a' are equal to 0. 'a' = 0
    - Neighboring cells containing an 'b' are equal to 1. 'b' = 1
    - Neighboring cells containing an 'c' are equal to 2. 'c' = 2
- For each cell of the matrix, update the next iteration of the matrix using the following rules:
    - If the current cell contains an 'a'
        - If the sum of the values is a prime number, then the cell remains as an 'a'
        - If the sum of the values is an even number, then the cell becomes a 'b'.
        - If the sum of the values is an odd number, then the cell becomes a 'c'.
    - If the current cell contains a 'b'
        - If the sum of the values is a prime number, then the cell remains as a 'b'
        - If the sum of the values is an even number, then the cell becomes a 'c'.
        - If the sum of the values is an odd number, then the cell becomes an 'a'.
    - If the current cell contains a 'c'
        - If the sum of the values is a prime number, then the cell remains as a 'c'
        - If the sum of the values is an even number, then the cell becomes an 'a'.
        - If the sum of the values is an odd number, then the cell becomes a 'b'.

For this program, a neighbor is defined as any cell that touches the current cell, meaning each current cell has between 3 and 8 neighbors depending on its position. The two examples below show the neighbors (**N**) for a given cell (**C**) with example #1 showing a cell in the middle of a matrix and example #2 showing a cell on an edge.

Neighbor Example #1

	0	1	2	3	4	5
0						
1		N	N	N		
2		N	C	N		
3		N	N	N		
4						
5						

Neighbor Example #2

	0	1	2	3	4	5
0						
1						
2						
3						
4					N	N
5					N	C

Your solution will generate the starting matrix (Time Step 0) as described in Stage 1.2 and then simulate steps 1 through 100 using the rules listed in this stage (Stage 1.3). The final matrix (**Time Step 100**) will then be utilized in the next stage.

Stage 1.3 Example

If you assume the input file contained the string "Cbhij" (Length L = 5) and your application received the seed string "abc", then your application should generate the following 5x5 matrix as Time Step 0 and then produce the following 4 timesteps by constantly re-applying the rules for Stage 1.3.

Time Step 0

a	b	c	a	b
c	a	b	c	a
b	c	a	b	c
a	b	c	a	b
c	a	b	c	a

→ Time Step 1

a	b	a	b	b
a	c	c	c	b
b	c	c	c	a
b	c	c	c	b
b	b	b	a	a

→ Time Step 2

a	c	b	c	c
b	a	a	b	c
b	a	a	a	b
b	c	a	a	c
c	b	b	b	a

→ Time Step 3

a	c	b	a	c
b	a	b	c	a
c	a	a	b	b
c	a	a	a	c
a	c	c	b	a

→ Time Step 4

a	c	b	b	c
c	a	c	a	b
c	b	b	c	b
a	b	b	a	c
b	a	c	c	a

Stage 1.4 – Decryption

Using the Time Step 100 matrix, your solution will then need to begin the process of decrypting the string using the data within this final matrix. The decryption process is done using the following steps:

1. **Column Summation**
    a. Starting with the 0th column, sum together all the cells in this column using the following values:
        i. Cells containing an 'a' add +0.
        ii. Cells containing a 'b' add +1.
        iii. Cells containing a 'c' add +2.
2. **Decrypt the Letter**
    a. Using the provided python function "decryptLetter", pass the 0th character of the encrypted string and the sum of the 0th column.
    b. This function will then rotate the character to its decrypted character and return that character as a length 1 string.

3. **Continue Decryption**
    a. Once you have decrypted the 0^{th} column, perform the same two steps on all remaining columns.
    b. The resulting string created by concatenating all of these decrypted letters together is the decrypted string.
4. **Write to Output File**
    a. Write the decrypted string to the output file given by the command line argument '-o'.

Phase 2 – Concurrent Computation

Stage 2.1 – Concurrency Using Multiprocessing

Once Phase 1 has been completed, the next task is to re-write Stage 1.3 (Matrix Processing) to now make use of currency via the *multiprocessing* module. As such, your solution is required to effectively utilize the Python *multiprocessing* module to enhance its performance. It should be capable of initializing a number of processes that corresponds to the number specified by the user through the `-p` option. Typically, when not using the *multiprocessing* module, your program executes serially—that is, it operates using a single thread or process. However, for the purpose of this project, you must adapt your program to run in parallel using multiple processes. To comply with this requirement, ensure that your solution:

- Parses the `-p` option to determine the number of processes the user intends to create.
- Implements the `multiprocessing` module to spawn the exact number of processes requested by the user.
- Orchestrates these processes to work concurrently on the task at hand, which should demonstrate a clear understanding and application of parallel processing concepts.

Proper implementation of the *multiprocessing* module is crucial. If your solution does not conform to these multiprocessing standards, it will be deemed incorrect. Make sure to test and verify that each process is performing its intended task and that all processes are running in parallel as expected.

Command Line Arguments and Examples

Command Line Arguments:

Develop a program capable of accepting the following command line arguments:

- -i <path_to_input_file>
  - Purpose: This option retrieves the file path to a file containing the encrypted input string.
  - Input Type: String
  - Validation: Entire file path must exist, otherwise error.
  - Required: Yes
- -s <seed_string>
  - Purpose: This option retrieves a string that sets the seed string used to compute the starting matrix.
  - Input Type: String
  - Validation: The string may only consist of 1 to many lowercase letters a, b, and c. Language: $[abc]^+$
  - Required: Yes
- -o <path_to_output_file>
  - Purpose: This option retrieves the file path for the final output file.
  - Input Type: String
  - Validation: The directories in the file path must exist, otherwise error.
  - Required: Yes
- -p <int>
  - Purpose: This option retrieves the number of processes to spawn.
  - Input Type: Unsigned Integer
  - Validation: Must be a positive integer > 0, otherwise error.
  - Required: No
  - Default Value: 1

Example executions:

- *python3 Eric_Rees_R123456_final_project.py -i inputFile.txt -s abc -o decrypted.txt -p 36*
  - Sets input file to “inputFile.txt”
  - Sets the seed value to the string “abc”
  - Sets output file to “decrypted.txt”
  - Sets process count to 36

- *python3 Eric_Rees_R123456_final_project.py -s bacca -o myOutput.txt -i myInput.txt*
  - Sets input file to “myInput.txt”
  - Sets the seed value to the string “bacca”
  - Sets output file to “myOutput.txt”
  - Sets process count to 1 (default when not specified)

File Specifications

Input File Specification

The input file must be an ASCII text file containing a single string that abides by the following specifications:

- Encoding: ASCII
- File Structure
  - The file contains a single string.
  - Preceding and trailing whitespaces should be ignored and removed.
  - Internal spaces within the string must be preserved.

Example

A file that contains the following string

```
“       This is a sample string with  preserved   internal   spaces.   \n\n”
```

file would be stored by your program simply as

```
“This is a sample string with  preserved   internal   spaces.“
```

Output File Specification

The output file must be an ASCII text file containing a single string that abides by the following specifications:

- Encoding: ASCII
- File Structure
  - The file contains a single string.
  - Preceding and trailing whitespaces should be ignored and removed.
  - Internal spaces within the string must be preserved.

Additional Rules and Hints

Program Output

Your program may print as much information as you wish to the screen during execution. The first line of output printed to the screen during execution **must** be “Project :: R#” where R# is your specific TTU R#. That is the only **required** item it must print and is the only line the grading script cares about. Your program correctness grade will be determined by the output found in the file specified by the -o argument. Your output file should **only** contain the decrypted string with no other extraneous information.

Advice

Leave room for testing – this application is meant to show you how intense large-scale computations can be. As such, testing a poorly optimized solution could take up to 48 hours to process. Make sure you leave at least 7 days for testing. Waiting to test your program a few days before it is due is a guaranteed way of not completing this project on time.

Python Information

1) Warning about compatibility:
    a. Your solution must be written to be compatible with Python version 3.x with only the base Python3 modules.
        i. Specifically, the grading script will be running a fresh install of Python 3.9.12.
        ii. Keep in mind, this means you cannot use Numpy, Scipy or other possibly useful matrix modules.
    b. Python version 2.x was ubiquitous and heavily documented online for over two decades.
        i. Python version 2.x code is often not compatible with Python version 3.x.
2) Python 3.x Tutorial / Refresher Links:
    a. https://www.w3schools.com/python/default.asp
    b. https://www.tutorialspoint.com/python3/index.htm
3) Python 3.x Tutorial for handling command line arguments:
    a. https://docs.python.org/3.8/howto/argparse.html
    b. https://pymotw.com/3/argparse/
4) Additional information and tutorials for multiprocessing:
    a. The link below covers the multiprocessing module in considerable detail:
        i. Multiprocessing - http://zetcode.com/python/multiprocessing/

What to turn in

A zip archive (.zip) whose name does not matter which contains your source code files such that:

- The primary python file for your project is named <FirstName>_<LastName>_<R#>_final_project.py
    - Example: Eric_Rees_R123456_final_project.py
- Any additional files necessary for your program to execute should be included.

Keep in mind the grading program will require that your python file be named appropriately.