Scientific programming in Python: final assignment

Solve any four questions. Place all the deliverables in a zip file with your name on it and submit it electronically by April 11th, 2024. You may answer all five questions and the lowest mark will replace your midterm assignment, if higher. **This is a hard deadline**: its distance is for your scheduling convenience; the actual assignment should take only a few hours.

Question 1: Excel Column Indexing

In Microsoft Excel spreadsheets, the rows are indexed with numbers, but the columns are indexed with letters—in a peculiar way. First, all the letters are used, in alphabetical order, then the cycle starts over, but with the letter `A` in front. Next cycle, `B` is in front, etcetera. Here are some excerpts from the sequence of column labels:

```
A B C ... Z AA AB ... AZ BA BB ... BZ ... ZA ZB ... ZZ AAA AAB ... AAZ ABA ABB ...
```

You can open up your copy of Excel to see the column labeling in action, if it is still not clear. You must code a module, `excelindex.py`, that exposes two functions:

`find_num(col_string)`: given a string, `col_string`, containing a column label such as `'AAZ'`, returns an integer representing the equivalent numeric column index, starting from 1. For clarity: `'A'` should return 1, `'B'` should return 2, `'Z'` should return 26, `'AA'` should return 27, etc.

`find_string(col_num)`: given a numeric column index, `col_num`, returns a string containing the equivalent column label. Thus, 1 should return `'A'`, 2 should return `'B'`, 26 should return `'Z'`, 27 should return `'AA'`, etc.

I will import the functions from your module into a script placed in the same directory as `excelindex.py`, that imports your functions after importing them the invocation `from excelindex import find_num, find_string`.

Question 2: Run-Length Encoding

Imagine you are presented with a sequence of digits. You may create a new sequence by reading aloud the previous sequence and using that reading as the next sequence, summarizing any run of x identical digits d as "x d". For example, 3 1 1 is read as "one three, two ones", which becomes 1 3 2 1 (1 3, 2 1s). You may continue generating sequences iteratively, using the previous value as input for the next step. For each step, take the previous value, and replace each run of identical digits (like 4 4 4) with the number of digits (e.g., 3) followed by the digit itself (4).

For example:

```
1 becomes 1 1 (1 copy of digit 1).
1 1 becomes 2 1 (2 copies of digit 1).
2 1 becomes 1 2 1 1 (one 2 followed by one 1).
1 2 1 1 becomes 1 1 1 2 2 1 (one 1, one 2, and two 1s).
1 1 1 2 2 1 becomes 3 1 2 2 1 1 (three 1s, two 2s, and one 1).
```

Create a module `sequenceprop.py` containing a function `sequence_prop_30()` that takes an integer which represents your original sequence of digits (i.e., the integer 522 represents the sequence "5 2 2") and determines how many digits are in the final result after this iterative replacement process has been repeated 30 times.

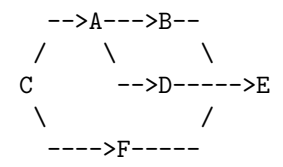
I will import the function from your module into a script placed in the same directory as `sequenceprop.py`, that imports your functions after importing them the invocation `from sequenceprop import sequence_prop_30`.

Question 3: Task Ordering

Suppose you have a project consisting of a series of steps, and you have requirements about which steps must be finished before others can begin. Each step is designated by a single capital letter. For example, suppose you have the following instructions:

```
Step C must be finished before step A can begin.
Step C must be finished before step F can begin.
Step A must be finished before step B can begin.
Step A must be finished before step D can begin.
Step B must be finished before step E can begin.
Step D must be finished before step E can begin.
Step F must be finished before step E can begin.
```

Visually, these requirements look like this:



These requirements can be represented by a list of two-item lists, with each two-item list's first element representing a task that must precede a task listed as the second element. For example, the network described above could be represented in Python as:

```
rule_list = [ ['C','A'], ['C','F'], ['A','B'], ['A','D'], ['B','E'], ['D','E'], ['F','E'] ]
```

Note that the order of the two-item lists within the outer list of lists is not important.

Your task is to create a Python module called `sorted.py` containing a function called `sorted_steps()` that takes a list of lists formatted as above that define the step ordering rules as the sole user-provided argument. The function must return a string consisting of the letters corresponding to each step, in an order that satisfies all the conditions. It must contain each step mentioned in a rule, and each of these exactly once.

For example, consider the following code:

```
from sorted import sorted_steps

rule_list = [ ['C','A'], ['C','F'], ['A','B'], ['A','D'], ['B','E'], ['D','E'], ['F','E'] ]
step_string = sorted_steps(rule_list)
print(step_string)
```

This could legally print `CABDFE` or `CFADBE` or `CAFDBE`. It could not, for example, legally print `CDFABE` because this violates the rule `step A must be finished before step D can begin`.

BONUS: whenever there are multiple steps that could legally be yielded (i.e., whose prerequisite steps have been completed), yield the step that is first in alphabetical order. If `SortedSteps` obeys this additional alphabetization constraint, the above code would generate `CABDFE`.

Question 4: Conway's Game of Life

Mathematician John Conway devised the "Game of Life", in which simple rules lead to surprisingly complex behaviour. The Game of Life is "played" on a two-dimensional orthogonal grid of square cells (imagine a piece of graph paper), each of which is in one of two possible states, **live** or **dead**. Every cell interacts with its eight neighbours, which are the cells that are horizontally, vertically, or diagonally adjacent. All cells beyond the grid may be assumed dead.

At discrete points in time, called "generations", births and deaths occur simultaneously, according to the following rules:

1. Any live cell with two or three live neighbours stays live.
2. Any dead cell with three live neighbours becomes a live cell.
3. All other live cells become dead in the next generation. Similarly, all other dead cells stay dead.

These rules continue to be applied repeatedly to create further generations. The initial pattern of live and dead cells entirely determines the future evolution of the system, as the configuration of live and dead cells in each generation is totally determined by the configuration in the preceding one.

Using Numpy and Matplotlib, simulate Conway's game on a 500 by 500 grid. Represent the grid with a Numpy array in which 1 represents a live cell and 0 represents a dead cell. Initialize the grid with an initially random assortment of ones and zeros, and then evolve the grid for 100 generations, using Conway's three rules. Use Matplotlib to create a 100-step movie showing the evolution of the grid as it goes through each generation. Your deliverable for this exercise will be this movie, saved as a `.gif`.

This exercise is straightforward, but will require you to do a little research into additional capabilities in the libraries we looked at in class. Google and Stack Overflow will be your friends here. Some functions you may want to consider include `matplotlib.pyplot.imshow`, the contents of `matplotlib.animation`, and `scipy.ndimage.convolve`. Remember that is is more efficient to use bulk (vector) operations than to use loops on large matrices.

Question 5: Trouble Sort

Consider a new variant of the bubble sort algorithm. As we discussed, the basic operation of the standard bubble sort algorithm is to examine a pair of adjacent numbers, and reverse that pair if the left number is larger than the right number. But the new algorithm examines a group of **three** adjacent numbers, and if the leftmost number is larger than the rightmost number, it reverses that entire group. The algorithm is a "triplet bubble sort", named Trouble Sort for short. Here is Python code for the Trouble Sort algorithm:

```
def trouble_sort(arr):
    done = False
    while done != True:
        done = True
        for i in range(len(arr)-2):
            if arr[i] > arr[i+2]:
                done = False
                arr[i],arr[i+2] = arr[i+2],arr[i]
    return arr
```

For example, given the input list **5 6 6 4 3**, Trouble Sort would take three passes through the list, proceeding as follows:

1. inspect **5 6 6**, do nothing: **5 6 6 4 3**
   inspect **6 6 4**, see that $6 > 4$, reverse the triplet: **5 4 6 6 3**
   inspect **6 6 3**, see that $6 > 3$, reverse the triplet: **5 4 3 6 6**

2. inspect **5 4 3**, see that $5 > 3$, reverse the triplet: **3 4 5 6 6**
   inspect **4 5 6**, do nothing: **3 4 5 6 6**
   inspect **5 6 6**, do nothing: **3 4 5 6 6**
3. Inspect the three triplets and do nothing; subsequently the algorithm terminates.

Unfortunately, it is possible that Trouble Sort does not correctly sort the input list. Consider the list **8 9 7**, for example.

Your deliverable for this exercise is a module `testsort.py` that contains a function `test_trouble`. The function `test_trouble` will take a Numpy array of integers as its sole argument. It must return **-1** if Trouble Sort will successfully sort the list into non-decreasing order. Otherwise, it must return the index (counting starting from 0) of the first sorting error after the algorithm has finished: that is, the first value that is smaller than the value that comes directly before it when Trouble Sort completes.

Of course, you could simply implement Trouble Sort in Python, and then check the final result to see if it is non-decreasing. However, this will work too slowly for very large inputs. Instead: use vectorized Numpy routines, including array slicing, vectorized arithmetic, reshaping, and sorting (Numpy's `sort` function implements very fast sorting algorithms that operate on Numpy arrays). You'll need to consider how Trouble Sort operates, and how that can be expressed in terms of sorting algorithms that work properly, such as the ones included with Numpy!

Your implementation of `test_trouble` will be tested by passing it lists containing 10000 integers, each between 0 and 10000, inclusive. **To receive full marks, your implementation must operate on average at least 1000x as fast as a naive "implement Trouble Sort directly using Python loops and check how it did" approach (this should be easy if you always use native Numpy functions and vectorized operations).**