Homework 1

Due Friday, Jan 29, 2021 at 8pm ET

Use the following commands to download and unpack the distribution code:

```
$ wget https://amirkamil.github.io/homework/hw1/starter-files.tar.gz
$ tar xzf starter-files.tar.gz
```

You may work alone or with a partner. Please see the syllabus for partnership rules. As a reminder, you may not share any part of your solution outside of your partnership. This includes code, test cases, and written solutions.

1. *Python practice.* Implement a simulation of Conway's Game of Life in Python3. While the original simulation is on an infinite grid, your implementation should be on a finite grid. Edge cells have fewer neighbors but should otherwise follow the same rules as any other cell.

   Place your code in a file named `life.py`. Your simulation must have the following interface:

   ```
   def simulate(rows, cols, steps, live_cells):
       """Run Conway's Game of Life on a finite (rows x cols) grid.

       Runs the simulation for the number of time steps specified by
       the steps parameter. Initializes the grid such that all cells
       are dead except those listed in live_cells, which should be a
       sequence of (row, column) pairs. Prints the grid after each
       time step.
       """
       ...
   ```

   When printing the grid, you should first print a line consisting of `cols + 2` dashes (`-`). Each row should then start and end with a pipe (`|`), each live cell should be printed as an asterisk (`*`), and each dead cell should be printed as a space. Finally, print another line consisting of `cols + 2` dashes followed by another empty line.

   We have provided a test case in `life_test.py` and the expected output in `life_test.correct`.

   You may use the NumPy library if you wish.

   Hint: It is possible to implement the algorithm without special cases for the borders. Think about storing the data in a $(rows+2) \times (cols+2)$ grid rather than in a $rows \times cols$ grid.

2. *Python classes and special methods.* Read through the section in the Python reference on special methods, particularly the subsections on basic customization and emulating container types. Also read the section in the reference on iterators. Then complete the definition of the `Range` class, which is a simplified version of Python's built-in `range` type, in `range.py`.

   The `Range` class is a container representing a fixed sequence of integers. It has the following methods:

   - `__init__`: The constructor, which takes in the start, stop, and optional step. If the step is not given, it defaults to 1. We will only handle the case where the step is a positive integer.
   - `__iter__`: Returns an iterator over the range, represented as a `RangeIter` object.
   - `__len__`: Returns the number of integers in the range.
   - `__contains__`: Returns whether or not the given integer is a member of the range.

   The `RangeIter` class implements the iterator interface and is used for iterating over a `Range`.

   Once you have completed both classes, you can run the tests in the `test()` function from the command line:

   ```
   $ python3 range.py
   ```

3. *Literals and operators.* Implement a `BitVector` type in C++17, representing a growable sequence of booleans. Your `BitVector` must support the following operations:

   - String literals that end with the `_bv` suffix construct a `BitVector` from the string with bits in order from left to right. A `0` character specifies `false` and a `1` character specifies `true`. Any string character other than `0` or `1` produces an unspecified value. Example:

     ```
     "1010"_bv  -->  [ true, false, true, false ]
     ```

   - The `size()` member function returns the size of the `BitVector`. The return type should be `size_t`.
   - The `push_back()` member function takes in a `bool` and appends it to the end of the `BitVector`.
   - The `[]` operator returns the boolean at the given position. You may return by value or reference (i.e. you do not have to support modifying a `BitVector` using the `[]` operator).
   - The bitwise operators `&`, `|`, and `^`, when applied to two `BitVectors`, should result in a `BitVector` that consists of the `AND`, `OR`, and `XOR` of the two `BitVectors`, respectively. If they differ in size, then the operators should treat the smaller as if it were padded on the right with zeros.
   - The << stream insertion operator when applied to a `BitVector` should insert each boolean as a `0` or `1`. Example:

     ```
     cout << "1010"_bv;  -->  prints 1010
     ```

   Write your code in `BitVector.h`. We have provided a test case in `BitVector_test.cpp` and expected output in `BitVector_test.correct`.

   You may use any standard libraries you like. You may find the C++ documentation on user-defined literals helpful. (Try a web search for "C++ user-defined literals", without the quotes.)

Submission

Submit `life.py`, `range.py` and `BitVector.h` to the autograder before the deadline. Be sure to register your partnership on the autograder if you are working with a partner.