Chapter 6 Labs

Labs

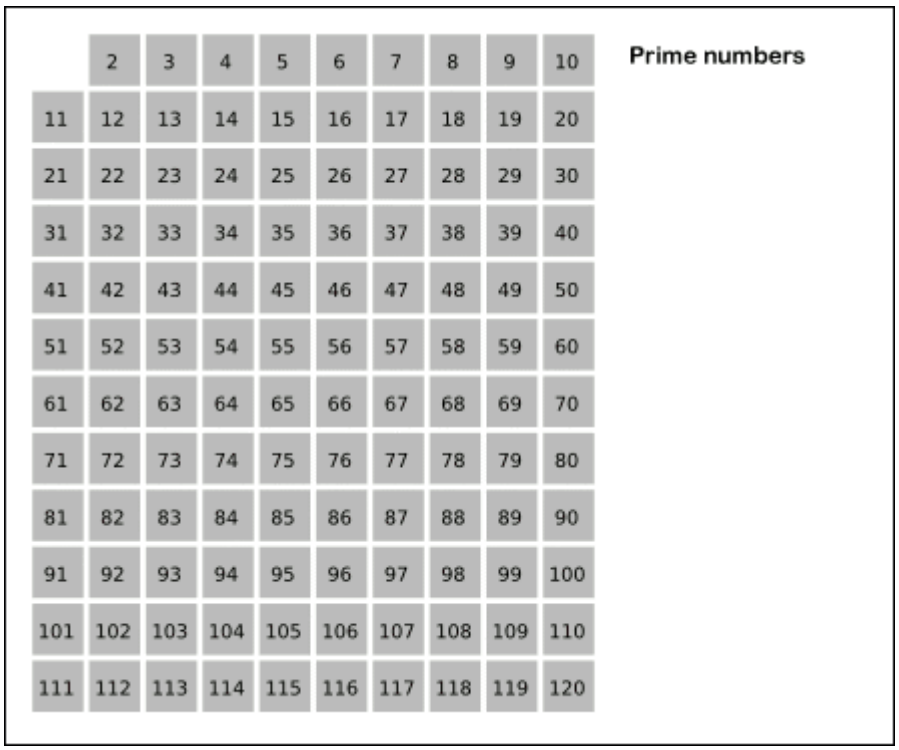
1. Sieve of Eratosthenes
2. Simple Integer List
3. Sorting Comparison
4. Game of Life

Sieve of Eratosthenes

In previous labs, you've explored the concept of primality testing using the method of trial division (i.e. for a number n, iterate of 2..sqrt(n) and check whether they divide n). In this exercise, we explore a more sophisticated algorithm called the sieve of Eratosthenes for finding all primes up to a given value. Here is a basic outline of the method:

1. Consider the numbers 2..n.
2. For each number, do the following:
    1. If the number is composite, do nothing
    2. Otherwise add the number to the list of prime numbers and mark all of its multiples as composite

This procedure is illustrated in the following animation:



Your task is to implement this algorithm in the method `findPrimes()` in `Sieve.java`. Use a `boolean` array to keep track of the numbers marked composite and an `ArrayList<;Integer>;` to store the primes.

Simple Integer List

In this exercise, you will explore the internals of `ArrayList`s by implementing your own resizable list of `int`s. In the file `SimpleIntegerList.java`, there is a basic skeleton of the class with stubbed methods. Your task is to implement all of them such that they match the behavior of the built-in `ArrayList`. It is safe to assume that all provided indices will be valid.

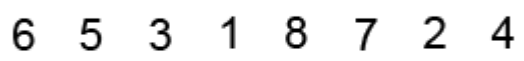
Sorting Comparison

In this chapter we've explored two common sorting methods: selection sort and insertion sort. Recall that the time complexity of both algorithms is `O(n^2)`. In this exercise, you'll compare selection sort, insertion sort, and a third algorithm on an empirical basis. Your task is two-fold:

- Implement bubble sort. This sorting algorithm is the most rudimentary of the three. Here's the basic idea:

  1. For each adjacent values in the array, swap them if they're out of order.
  2. Repeat until a single pass through the array requires no swaps (all of the values are in order).

  This animation should help clarify:

  Please write your code in the `bubbleSort()` method in `SortingComparison.java`.
- This next phase involves comparing the algorithms. When you run the program, it will test the algorithms with arrays of varying sizes and display the time taken in milliseconds (1000 milliseconds = 1 second). With these results in mind, respond to the following questions as part of your reflection:
  - Why do insertion and selection sort have different running times even though they are in the same time complexity class?
  - What do you think the time complexity of bubble sort is? Why? (It's OK if you aren't correct; I want you to think and not copy off the Internet.)

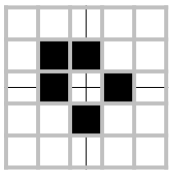
Game of Life

Introduction

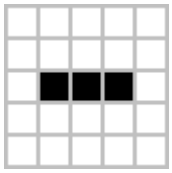
The Game of Life belongs to an interesting class of programs called cellular automatons. Similar to other automatons, the Game of Life consists of a rectangular grid of cells. Each cell can be in one of two states: "alive" (black) or "dead" (white). In each step of the "game", the following rules are applied:

1. Any live cell with fewer than two live neighbors dies
2. Any live cell with two or three live neighbors lives on to the next generation
3. Any live cell with more than three live neighbors dies
4. Any dead cell with exactly three live neighbors becomes a live cell

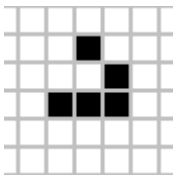
Although these rules are rather simple, surprisingly complex behavior arises from them. For example, stable configurations can occur like this boat:



Although this pattern is static, there are some stable configurations that oscillate like this blinker:



Finally, some patterns called spaceships move across the grid like this glider:



Your Task

In this exercise, all of the graphical work has been done already. Your task is to fill in the `Board` class which includes the actual logic for transitioning from one state to the next (according to the rules above). Some of the class has already been filled in. In the code, the grid is represented as a 2D array of booleans (`true` for alive, `false` for dead). Note that there are actually two arrays, `state` for the current state and `nextState` for the next state (so that the original array isn't modified during the transition).

The first step is to fill in the following helper methods. These will help with the transition logic.

- `public boolean isInside(int row, int col)`—returns `true` if the location is inside of the board and `false` otherwise.
- `public static int[][] getNeighbors(int row, int col)`—returns an 8x2 array containing the locations of the eight neighbors as (row, col) pairs

Now we're ready to fill in the `update()` method. Where applicable, please make use of the helper methods in the class including the ones above. To help guide you, here's a procedural outline:

- Loop through every cell in `state`.
    - Get the neighbors of the cell.
    - Ignoring the invalid neighbors, compute the number of alive neighbors.
    - Use this to compute the state of the corresponding cell in `nextState` according to the rules at the beginning.
- Update `state` using the `shiftState()` method.

When you're finished, run the `GameWindow.java` program. When running, the board will update 10 times per second. To pause/resume it, press the space bar. To run a single update, press the right arrow. To modify individual cells, just click on them (probably while it's paused) to toggle their state. For more fun, use the items in the Load menu to change the seed state.