CSE 101 Winter 2018
Programming Assignment 4

Due: Saturday, March 10th, 2018 at 11:59 PM PST

Link to starter code: https://github.com/UCSD-CSE101/W18-PA4

PLEASE READ THE FAQ BEFORE POSTING ON PIAZZA:
https://docs.google.com/document/d/1S3WpMKM-h3532sttRURVWO-sZbTSmqswqj_e62QMQRE/edit?usp=sharing

For this programming assignment, we will be simulating an interview for a software engineering job (or as close as we can get!) This means that not only that you will have more freedom in how you can implement your algorithms, but also that we will do less hand-holding and directing. You are free to use any STL libraries, structs, data structures, helper functions, etc. that you need, as long as your code compiles and runs with our Makefile and autograder.

Since we have fewer restrictions, we are providing an autograder for you to verify your implementation as you code. Note that these will not be the exact test cases you will be graded on, although the ones we use will be very similar both in scope and size.

We will also not provide a target runtime requirement for these tests (except for Question 1), but your program will be timed against large inputs, so the naive or brute-force solution will not be efficient enough. **If your algorithm is taking longer than a few seconds for any individual test case you may want to take another look at your code.** The autograder should be able to complete running on all questions within 10 seconds. We will give your code a time limit so make sure your runtime is reasonable!

Each of the following four questions will have its own autograder and tester, as well as associated test cases and reference outputs in the solutions folder. To run the tester, please refer to the instructions in each of the problems below.

Question 1: Factory Serial Numbers [20 points]

Input: A sorted list of consecutively increasing factory serial numbers (integers) in increasing order, with **one and only one** serial number printed multiple times.

Output: The serial number that is misprinted, i.e., that occurs multiple times.

You are an inspector on a toy factory line, and your job is to make sure that each toy produced has a different serial number. The machine that stamps serial numbers onto toys gives a unique serial number to each toy by stamping consecutively increasing values (1102, 1103, 1104, 1105, etc.).

Unfortunately the serial stamping machine malfunctions, and stamps the same serial number onto an unknown number of toys before returning to its normal function and continuing stamping consecutive numbers. You do not realize this mistake until you stop the machine much later.

An example list of misprinted serials might be (1102, 1103, 1103, 1103, 1103, 1104, 1105).

Given the list of toy serial numbers produced your goal is to find the **single** serial number that has been misprinted multiple times. Note that **one and only one** serial number will be misprinted multiple times.

This is the only problem where we will mention a hard bound on runtime. In order to pass the test cases, your algorithm **MUST** run asymptotically faster than O(n) time. We will verify this in the autograder by timing the execution of your algorithm.

To Make:

```
make TestSerial
```

To Test an Individual File:

```
build/TestSerial testcases/serial/input_file
```

To Run the Autograder:

```
build/TestSerial
```

Example Test File:

```
10    // The number of serial numbers
223 224 225 226 226 226 226 227 228 229 // The list of serials
```

Example output:

```
Repeated serial number: 226
```

In the example above, we return 226, since the machine has misprinted it multiple times in the list of toys.

Question 2: Tennis Lessons [20 points]

Input: A list of lesson time **Intervals**, each with a start and an end time.

Output: The minimum number of tennis courts required to host all the lessons

You are a supervisor for a tennis facility and have a list of tennis lesson intervals that have been booked at your facility. A tennis lesson interval for this problem is defined as an instance of the **Interval** class, defined in **Interval.hpp**. The **Interval** class contains an integer **start** and an integer **end**, which represent the start time of the lesson and the end time of the lesson respectively.

Your goal is to determine the minimum number of courts you need to set aside for all the lessons booked. Although you do not need to determine the specific assignment of lessons to courts, all lessons must follow these constraints:

- Every lesson needs to have a court that it can be completed at
- A lesson must start and end at the same court
- A lesson cannot start in a court until the previous lesson at that court has ended.
- **As soon as a lesson has ended, a new one can immediately start.** This means that two lessons with intervals [1,3) and [3,4) can be scheduled on the same court.

You are not guaranteed any specific ordering for the list of Intervals given to you.

To Make:

```
make TestTennis
```

To Test an Individual File:

```
build/TestTennis testcases/tennis/input_file
```

To Run Autograder:

```
build/TestTennis
```

Example Test File:

```
5          // The number of lessons
1 3        // this lesson runs from time [1 - 3)
3 4        // this lesson runs from time [3 - 4)
7 9
2 6
8 10
```

Example Output:

```
Min courts required: 2
```

In the example above, we cannot schedule all the lessons on 1 court, since lessons [1, 3) and [2, 6) would overlap. Instead, we can schedule lessons [1, 3), [3, 4), and [7, 9) on court 1, and lessons [2, 6) and [8, 10) on court 2. Thus we have a minimum of 2 courts necessary to hold all the lessons.

Question 3: Number of Buildings in a City [30 points]

Input: M x N image of a city represented using a matrix of 1's and 0's

Output: The number of buildings in the grid

You are a building surveyor, and have been given a satellite image of a city. This image is represented by a grid of 1's and 0's, where a 0 represents a street or sidewalk, and a group of contiguous 1's represents a building. A contiguous group of 1's is defined as a group of 1's such that each 1 is **adjacent** to at least one other 1 in any direction, **except diagonally**.

For example, in the following grid, there are 3 different buildings present, each highlighted with a different color:

```
1 1 0 1 0
0 1 0 0 1
1 1 0 1 1
```

Your goal is to determine the number of buildings present in the image. Note that you are free to change the values in the TwoD_Array given to you.

To Make:

```
make TestBuildings
```

To Test an Individual File:

```
build/TestBuildings testcases/buildings/input_file
```

To Run Autograder:

```
build/TestBuildings
```

Example Test File:

```
3            // Number of rows in the grid
5            // Number of columns in the grid
1 1 0 1 0    // The image, represented as a grid of 0's and 1's
0 1 0 0 1
1 1 0 1 1
```

Example Output:

```
Total buildings: 3
```

Question 4: Delivering Ice [30 points]

Input: M x N matrix of temperatures for rooms in a warehouse

Output: Minimum amount of ice needed to start with in the top left corner to make it to the bottom right with at least one unit of ice left

You are a worker in an ice packing warehouse, and need to deliver some ice to your boss. The warehouse is made up of M x N rooms, laid out in a 2-d grid.

You start with an initial amount of ice, **K**, then **enter the top left room**. Unfortunately, the thermostat in the warehouse has malfunctioned and caused the temperatures in the different rooms to vary wildly.

- If a room's temperature is negative, it causes your ice to grow by the same amount. (If you enter a room with a temperature of -2, you will now have K+2 units of ice)
- If a room's temperature is positive, it causes your ice to shrink by the same amount. (If you enter a room with a temperature of 5, you will now have size K-5 units of ice)
- If a room's temperature is 0, the amount of ice you have will not change.
- If your ice block melts entirely (reaches 0) you fail and cannot create more ice by entering a room with a negative temperature.

Your goal is to reach your boss's room, which is accessible only through the **final bottom right room**, with at least one unit of ice left. In order to reach your boss's room as quickly as possible, you decide to move **only downward or rightward**.

(S)
|
V

2	3	-3
5	10	-1
-10	-30	5

|
V
(E)

In the example warehouse given above, we need a minimum of 7 units of ice to make it through successfully. We start at (S) and enter the warehouse at the top left room.

- The temperature in this top left room is 2, so our ice block shrinks to 5 units.
- We move right and the temperature is now 3, so our ice block shrinks again to 2 units.
- We move right once more and the temperature is now -3, so our ice block grows to 5 units.
- We move down and the temperature is now -1, so our ice block grows to 6 units.
- We move down and the temperature is now 5, so our ice block shrinks to 1 unit.
- We exit the warehouse and deliver the 1 unit of ice to our boss at (E).

You will always enter in the top-left most room and exit from the bottom-right most room.

To Make:

```
make TestIce
```

To Test an Individual File:

```
build/TestIce testcases/ice/input_file
```

To Run Autograder:

```
build/TestIce
```

Example Test File:

```
3          // First line denotes number of rows
3          // Second line denotes number of cols
2 3 -3     // Each subsequent line denotes the temperatures in
5 10 -1    // each room of the M x N warehouse
-10 -30 5
```

Example Output:

```
Min ice required: 7
```