CS106B Handout 22
Summer 2011 July 18th, 2011

Midterm Exam

This is an open-note, open-reader exam. You can refer to any course handouts, textbooks, handwritten lecture notes, and printouts of any code relevant to any CS106B assignment. You may not use any laptops, cell phones, or handheld devices of any sort. You will be graded on functionality—but good style helps graders understand what you were attempting. You do not need to include any libraries and you do not need to forward declare any functions. You have 2 hours. We hope this exam is an exciting journey ☺.

Last Name: _____

First Name: _____

Section Leader: _____

I accept the letter and spirit of the honor code. I've neither given nor received aid on this exam. I pledge to write more neatly than I ever have in my entire life.

(signed) _____

		Score	Grader
1. Mystery	[10]	_____	_____
2. Saveway	[15]	_____	_____
3. Who Has A Silly Password?	[15]	_____	_____
4. Mirror Temple	[15]	_____	_____
Total	**[55]**	**_____**	

Problem 1: Tracing C++ programs and big-O (10 points, Medium Difficulty)

Assume that the functions **`Mystery`** and **`Enigma`** have been defined as follows:

```
int Mystery(int n) {
   if (n == 0) {
        return 0;
   } else {
        return Mystery(n - 1) + Enigma(n/2) + Enigma(n/2);
   }
}

int Enigma(int n) {
    int index = 0;
    int sum = 0;
    while (index < n * 2) {
        sum += index;
        index++;
    }
    return sum;
}
```

(a) [3 points] What is the value of **`Enigma(2)`**?

(b) [3 points] What is the value of **`Mystery(4)`**?

(c) [2 points] What is the worst case computational complexity of the **`Enigma`** function expressed in terms of big-O notation, where *N* is the value of the argument **`n`**? In this problem, you may assume that **`n`** is always a nonnegative integer.

(d) [2 points] What would be the effect on the worst case big-O of Enigma if the line:

```
if(RandomChance(0.5)) break;
```

had been added to the beginning of the while loop in Enigma?

Problem 2: Vectors, stacks, and queues (15 points, Very Hard)

You've been hired by Saveway, a national supermarket chain, to predict the total amount of time it will take a group of customers to check out. The parameters to your simulation are the number of cashiers and a group of customers.

We are going to use the following simplified model of a grocery story:

- Each register has a line and initially there are no customers in any of the register lines.
- When a customer is ready to check out they will join the shortest register line, based on number of people, and will never leave that line until served.
- Registers are first come first serve and can only process a single customer from their line at a time.
- The simulation is finished when all customers have been checked out.

For this problem we will use the following struct to represent customers:

```
struct customerT {
    //Minutes from the start before the customer will get in line
    int timeReady;
    //Minutes it will take for a register person to check out this customer
    int timeToCheckout;
};
```

The variable **`timeReady`** represents how many minutes into the simulation the customer waits before picking a register line to join. The variable **`timeToCheckout`** stores how long it will take for the person to checkout once he or she has become the **first** person in line for his or her register. You may assume that **`timeReady`** is always greater than or equal to 0 and that **`timeToCheckout`** is always greater than or equal to 1.

Since all customer variables are integers a good strategy for working through the simulation is to update your supermarket model one minute at a time.

For each register you should have a variable that represents the line of customers and a variable that represents any progress that has been made towards helping the first customer in line.

Your job is to write the function:

```
int GetCheckoutTime(Stack<customerT> & customers, int numCashiers)
```

that takes as input a **`Stack<customerT>`**, which you can assume is **sorted** by **`timeReady`** and the amount of time it takes to serve every customer.

(space for the answer to problem 2 appears on the next page)

Answer to problem 2:

```
// space for you to define any helper structs




int GetCheckoutTime(Stack<customerT> & customers, int numCashiers) {
```

continued from previous page

Problem 3: Maps and Sets (15 points, Hard)

In any map, multiple keys can have the same value. The Most Common Value is the value in the map with the largest number of associated keys.

Your job is to write a function **`KeysForMostCommonValue`** that takes in a **`Map<string>`**, finds the Most Common Value and returns the set of keys that map to the Most Common Value.

Consider the following map which stores username password combinations:

Key	Value
"cpiech"	**"password"**
"adgress"	**"pizza"**
"zuckerberg"	**"12345"**
"haxor2000"	**"password"**
"sheen"	**"12345"**
"rebeccablack"	**"friday"**
"neo"	**"password"**

The Most Common Value in the map is **"password"** and the keys that map to the Most Common Value are **"cpiech"**, **"haxor2000"** and **"neo"**.

Hint: This problem is much easier to solve if you populate internal set(s) and/or map(s) that will help you find the Most Common Value.

(space for the answer to problem 3 appears on the next page)

Answer to problem 3:

```
Set<string> KeysForMostCommonValue (Map<string> & map) {
```

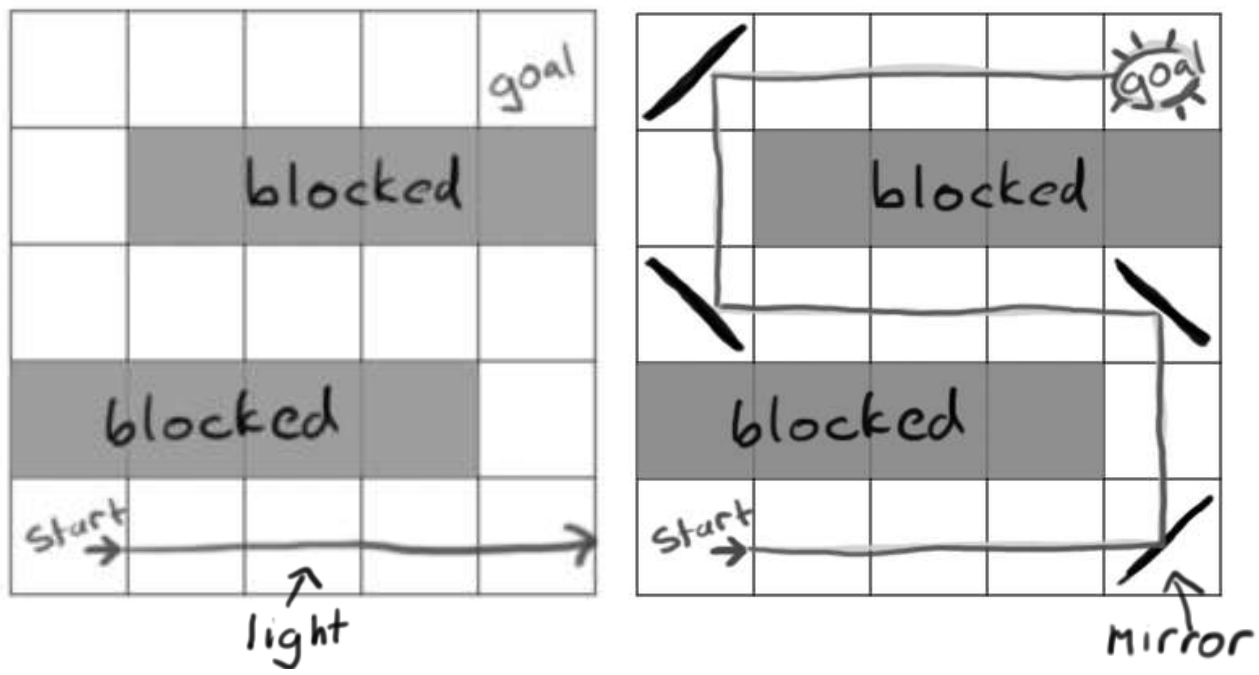
Problem 4: Recursive Exploration (15 points, Hard)

It has been thought that Ancient Egyptians used systems of mirrors to bring light into the inner chambers of their temples! Light normally travels in a straight line (think of a laser) but you can turn the path of the light using reflective surfaces.

Your job is to write a predicate function `IsSolvable` that returns whether or not there is a system of mirrors for a given temple such that light traveling in a specific direction from a starting position can be directed to a goal position using a *limited* number of mirrors.

This problem is constrained so that light is always traveling north, south, east or west and mirrors turn light clockwise (to the right) or counterclockwise (to the left). The temple is stored as a `Grid<bool>` where `true` means that the location is blocked—light cannot travel through this location—and `false` means that the location is open—light can travel through this location.

As an example: in the temple depicted in the image below-on-the-left, light starts in the bottom left corner traveling to the east and the goal is to get the light to the upper right corner. In this example there is no solution with three or fewer mirrors. However using four mirrors the system in the image below-on-the-right will direct light to the goal:



Hint: You should think of this problem step by step from the perspective of the light. At each square you can: move straight, turn left or turn right. It costs nothing to move straight—in the direction you were already facing—but it costs a mirror to turn left or to turn right..

The problem uses the following struct to represent a position in the grid and an enum to represent light direction:

```
// Directions that light can travel
enum dirT = {NORTH, EAST, SOUTH, WEST};

// Position in the blocks Grid
struct posT = {
     int row,
     int col
};
```

To help you with the updating of positions and directions, we provide you with the following functions (you **don't** have to write them):

```
posT Move(posT curPos, dirT curDir);
dirT TurnLeft(dirT curDir);
dirT TurnRight(dirT curDir);
```

`Move` returns the new position obtained by taking one step from the passed-in-position oriented in the passed-in-direction. **`TurnLeft`** and **`TurnRight`** return the orientation obtained from rotating left and right respectively from the passed-in-direction.

Using these predefined functions write a recursive predicate **`IsSolvable`** that takes a grid, a number of mirrors, a start configuration and a goal position as arguments and returns **`true`** if there is a system of mirrors to direct light to the goal and **`false`** otherwise.

(space for the answer to problem 4 appears on the next page)

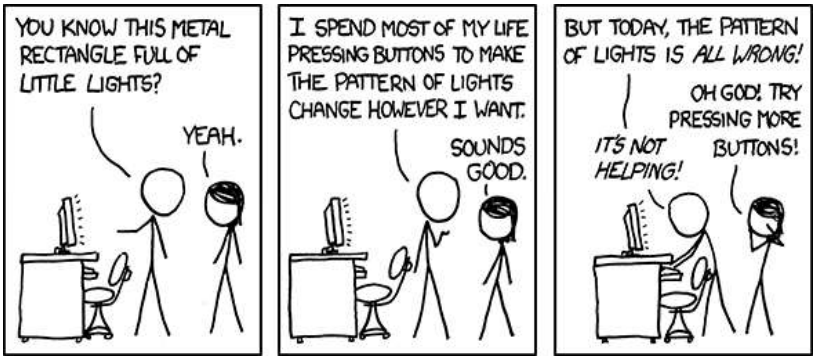
Answer to problem 4:

```
// predefined helper functions
posT Move(posT curPos, dirT curDir);
dirT TurnLeft(dirT curDir);
dirT TurnRight(dirT curDir);


bool IsSolvable(Grid<bool> & templeBlocks, int numMirrors, posT startPos,
                  dirT startDir, posT goalPos) {
```

Problem 5: Optional Fun Problem (0 points, Easy)

Make your own paneled style stick figure comic pontificating or commenting about CS, computers, life, etc. Here's one for reference:



This problem isn't required and there are no points associated with doing any work here, but any great gems will be very much appreciated by the course staff :).