1 Purpose

In this lab you will practice state-based reasoning by counting coin flips. Also, you will practice using two-dimensional vectors.

If you have not already done so, create a sub-directory `Lab11` within your `Labs` directory. This new directory is where you will work for this lab.

2 Finite State Machines

A **finite state machine** is a model of a computation that consists of **states** and **transitions** between states. States represent the current state of the computation; transitions represent ways in which the state of the program can change and actions that can occur as a result of a state change. Transitions are associated with states: for a given input, we look at the current state, and choose the transition for that state corresponding to the input.

As an example, consider text processing. We can read the characters from a stream as our input. Our states might be "reading whitespace" and "reading non-whitespace". Every time the type of character read changes, we change state. For example, if we're reading whitespace and we encounter a non-whitespace character, we change state to reading words. If we consider a word to be a sequence of non-whitespace characters, then we can count the number of words by incrementing a counter every time we transition from reading whitespace to reading non-whitespace.

2.1 Exercise 1: Counting Coin Tosses

In this part of the lab you will use the finite state machine approach to write a C++ program that determines and prints the longest run of heads in a row and the longest run of tails in a row.

We can use a finite state machine to count the number of heads in a row when tossing a coin. The two states are "saw a head (HEAD)" and "saw a tail (TAIL)." If we just saw a head and toss another head, we stay in the same state otherwise we transition to the other state. We start the finite state machine depending on the first toss. For example, if the first toss is a head, we start in state HEAD.

Draw a **state diagram** for tossing a coin. A state diagram has circles as states and arrows for transitions. Label the states and transitions. Remember to label how the machine starts.

Add actions on your state diagram to keep track of the longest run of heads in a row and the longest run of tails in a row.

Ask the user of the program to enter the number of coin tosses desired.

Use the `getRandom(2)` function each time you want to toss a coin.

```
flip = getRandom(2);
```

which returns the integers 0 or 1. Assume we represent a tail as 0 and head as 1. Remember to seed the random number generator by calling the `initRandom()` method once. To use these two methods, include the file:

```
/home/accounts/COURSES/cs203/include/random2.cc
```

From your finite state diagram, implement the C++ program. Use a C++ `enum` statement to label the two states and a second `enum` statement to label the results of a coin toss, i.e., `H` or `T`.

3 Exercise 2: Two Dimensional Vectors

Many problems especially in science and engineering use two-dimensional matrices. In math we would write the 3 by 4 matrix A as the following:

$$\begin{matrix} a_{1,1}a_{1,2}a_{1,3}a_{1,4} \\ a_{2,1}a_{2,2}a_{2,3}a_{2,4} \\ a_{3,1}a_{3,2}a_{3,3}a_{3,4} \end{matrix}$$

Since C++ starts the indices at zero, we would show a two-dimensional vector a with three rows and four columns as the following:

```
a[0][0]  a[0][1] a[0][2] a[0][3]
a[1][0]  a[1][1] a[1][2] a[1][3]
a[2][0]  a[2][1] a[2][2] a[2][3]
```

In C++ we would define the two-dimensional vector `a` as a vector of vectors as follows:

```
vector<vector<int> > a(3, 4);
```

Notice that the space after `<int>` is *very* important since "`>>`" is a token and is incorrect in this context.

We can access an element of a by using the notation `a[r][c]` where r is the row and c is the column. A common error of students is to reverse the order of the two indices. Remember **RC** for Rows-Columns by associating with RC Cola or Roman Catholic.

Copy the file `vectors2D.cc` from `~cs203/Labs/Lab11` into your `Lab11` directory. Study the program carefully.

You are to add two functions to the program. One to find the average grade for each student and a method that returns the maximum grade by any student on any exam. The program currently compiles and can be run.

4 What to Hand In

For exercise 1, hand in a copy of your working program, three sample runs with each 100 coin tosses, a neatly drawn state diagram including the additional actions that reflects the program you are handing in.

For exercise 2, hand in a copy of your working program and test runs that demonstrate that your program works.