Archery Target Practice Simulation

Assignment 4

Due date: Sunday, 17 Nov (11:59 pm)

1 Overview

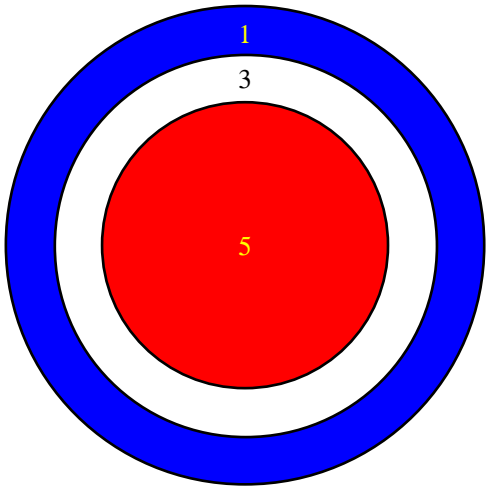
In this assignment, students will develop and implement a program that allows the user to "practice their archery target shooting".

The primary objectives of this assignment is for students gain experience in using:

- function decomposition to simplify implementing a program; and
- random number generation with the library functions `rand`, `srand`, and `time`.

2 The Archery Simulation

The goal of the simulation is to shoot an arrow at a target consisting of three concentric circles. Hitting the inner target (bullseye) is worth 5 points; hitting the middle target is worth 3 points; and hitting the outer target is worth 1 point (0 points for missing both targets). The targets are illustrated below.



The program user is allowed to continue the simulation for any number of rounds. Each round consists of shooting a total of 12 arrows; specifically, *four* arrows are shot at each of *three* different distances: 20 meters, 30 meters, and 40 meters.

Note that the player does **not** get to aim the arrows. Instead, the program uses random number generation to determine the where the arrows land; this is what is meant by the word *simulation*.

After each arrow is shot, the program prints a message describing the result. After each round is completed, the *total* score for that round is printed, along with a message asking the user if he/she would like to continue the simulation.

3 Program Structure

One of the most important reasons for implementing large programs as a group of small functions to to reduce program complexity. For this reason, students are **required** to implement their programs by defining the following four functions:

- `main`
- `playOneRound`
- `shootOneArrow`
- `randomProbability`

3.1 The `main` Function

The heart of the `main` function consists of a sentinel based loop. In each repetition of this loop, it calls the `playOneRound` function, and then interacts with the program user (meaning that a prompt is printed, such as `"Want to play another round?"`, the reply is scanned in, and either the loop is broken or continued).

Before the loop is entered, the `main` function should seed the random number generator by calling the library functions `time` and `srand` as described in section 5.9 of the textbook, so that a different simulation is performed each time the program is executed.

It is important to realize what the `main` function is *not* responsible for. It does not generate the random numbers; it does not do any of the scoring; and it does not print any other output aside from the prompt necessary for user interaction. Other function in the program will perform these operations.

3.2 The `playOneRound` Function

The `playOneRound` function **must** have the following prototype:

```
int playOneRound();
```

This function is responsible for calculating, printing, *and returning* the *total* score for a single round of the simulation[1]. It does not (directly) generate the random numbers, nor does it concern itself with the details of determining and printing the results of individual shots. Instead, it repeatedly calls the `shootOneArrow` function to accomplish these actions.

3.3 The `shootOneArrow` Function

The `shootOneArrow` function **must** have the following prototype:

```
int shootOneArrow(int distance);
```

The parameter `distance` to this function is the distance (20, 30, or 40) from which the shot is taken. As we will soon see, the farther the distance, the harder it is to hit the targets.

This function calls the `randomProbability` function to generate a random *double floating point* number between 0 and 1. The result of the shot is determined by both the parameter `distance` and the value returned by `randomProbability` according to the following table (p is assumed to be the value returned by `randomProbability`).

	Result			
`distance`	inner target	middle target	outer target	miss
20	$p \leq .5$	$.5 < p \leq .7$	$.7 < p \leq .9$	$.9 < p$
30	$p \leq .3$	$.3 < p \leq .6$	$.6 < p \leq .8$	$.8 < p$
40	$p \leq .1$	$.1 < p \leq .2$	$.2 < p \leq .7$	$.7 < p$

The function `shootOneArrow` is responsible for determining and printing both the shot result and the shot score. In addition, it *must* return this score back to its caller.

3.4 The `randomProbability` Function

The `randomProbability` function **must** have the following prototype:

```
double randomProbability();
```

This is the simplest function to implement: it calls the library function `rand` to generate a random integer between 0 and the symbolic constant `RAND_MAX`, and then returns the quotient obtained by dividing the value returned by the `rand` function by `RAND_MAX`. Note that in order to work correctly, the division must be the *double floating* point variety.

[1]The value returned by the `playOneRound` function is not actually going to be used by the `main` function in this assignment; nevertheless, the function should return a value, and the only logical value is to return the total score.

4 Sample Output

```
20 yard distance

1 points -- outer target
0 points -- you missed
5 points -- bullseye!
5 points -- bullseye!

30 yard distance

3 points -- middle target
1 points -- outer target
1 points -- outer target
3 points -- middle target

40 yard distance

0 points -- you missed
1 points -- outer target
5 points -- bullseye!
0 points -- you missed

TOTAL SCORE: 25

Enter 0 to exit, any other integer to play again: 1

20 yard distance

5 points -- bullseye!
1 points -- outer target
5 points -- bullseye!
5 points -- bullseye!

30 yard distance

5 points -- bullseye!
3 points -- middle target
0 points -- you missed
5 points -- bullseye!

40 yard distance

1 points -- outer target
1 points -- outer target
1 points -- outer target
0 points -- you missed

TOTAL SCORE: 32

Enter 0 to exit, any other integer to play again: 0
```

5 Testing the Program

Since the input data to this program is randomly generated, it is virtually impossible to duplicate the sample output shown above, and difficult to adequately test the program. Fortunately, we do know (via simple probability analysis) that the *expected* value of a single round is approximately 29. This allows us to do a reasonably good job by running the program several times, computing (by hand, or within the program) the sample average of the round scores, and verifying that the average is near 29.