PA102 – Analyzing Payoff

Prerequisites

You need a working *Prog4.c* from PA101 in order to work on PA102. Refer to the PA101 "wrapup" video for help in getting your version working.

Synopsis

We are going to start by modifying a bit our program so that we enable it to display the payoff values for a range of R1, R2 and R3 values.

Then, we will *refactor* it, that is, rewrite it differently without altering the features it implements. This first refactoring will allow us to extract a part of the program and rewrite it as a function. Decomposing a program into functions is more than an aesthetic requirement. It allows you to factorize parts of the implementation you are likely to use from multiple locations in your program.

Finally, we will apply what we learned about recursive functions.

Please note that, unlike PA101, this PA does not require you to implement every step as a separate project. Instead, you will start working with the files provided to you by your instructor, modify them to implement step #1's requirements, then implement step #2 requirements, then implement step #3 requirements.

Step #1 – Feature – What is the payoff distribution like?

Now that we know how to compute the payoff for a given triplet of values R1, R2 and R3, what about displaying the payoff for all possible triplets?

Modify your program so that, instead of asking the user for values, it uses loops to make each variable iterate over its valid values. For each possible value of R1, we want to generate each possible value of R2. For each of these, we want to generate each possible value of R3. When we are there, we compute the payoff for this specific triplet of values and display it on the screen, like so;

Example Output:

```
1     1     1     payoff is     1
1     1     2     payoff is     1
                  < Some output removed here to keep this short >
3     3     1     payoff is     4
3     3     2     payoff is     5
3     3     3     payoff is     3
```

Step #2 – Refactoring – Payoff is F(R1, R2, R3)

The main issue with our previous program, meant to determine the payoff of a few dice rolls in our homebrewed game, is that it is a monolithic block of code. Of course, we tried to add comments to make it easier to spot the various parts but it still very much looks like a big block of code when trying to understand it for the first time.

Breaking some things down into functions might help us organize this program in a better way. After all, we started by tackling the problem of computing a payoff, given the values of R1, R2 and R3. Only once this was done, we embedded that part into nested loops to vary the above-mentioned values. This suggests that the computation of the payoff might be a function that we would then invoke from within our loops.

Add a declaration and definition for the following function to your program:

Example:

```
int payoff ( int R1, int R2, int R3);
// Takes the 3 values representing dice rolls as parameters
// returns the payoff as specified in assignments.
```

When you are done, replace the code computing the payoff in your nested loops by an invocation to this function. Compare the output of this version of your program, to the previous output. They should match if your modification didn't break anything which was previously working.

Remarks – *Due to the fact we print out all possible values, we are not really defining a list of tests here. Looking at the whole output from one version to the other is enough to ensure our program is still behaving the same way. Whether you use selected tests or run your program on all possible input values, the very idea of running old tests to make sure nothing was broken by a modification is referred to as regression testing. It is important in large projects to ensure that, when you get your application to pass some 500 tests, these are still working when you decide to alter the font in the GUI.*

Interlude – Equivalence Loops / Recursive Functions

Here is an example of how an iterative loop might be translated into a recursive function. First, let us start by a simple program using a counter controlled loop;

```
int main(){
	int n = 0;
	while ( n <= 999 ){
		printf("I'm doing something with index %d\n", n);
		n++;
	};
}
```

Now let us rewrite this program so that we use a recursive function instead;

```
void myloop ( int n, int upto);

int main(){
	myloop( 0, 999);
}

void myloop (int n, int upto){
	/* takes as parameter
		start, the value of our loop index
		upto, the value up to which we iterate, inclusive.
	*/
	if (n <= upto){
		printf("I'm doing something with index %d\n", n);
		myloop( n+1 , upto);
	}
}
```

Step #3 – Refactoring – Recursive Functions vs. Loops

Use this equivalence to modify your program. Right now, we have three nested loops to iterate over all possible values of R1, R2 and R3. In pseudo-code, this would look like;

```
Loop#1 – for R1 in values [1..3]
    Loop#2 – For R2 in values [1..3]
        Loop#3 – For R3 in values [1..3]
            result = payoff(R1, R2, R3);
            …
```

Let's start by replacing the whole thing by a call to a function named *loopR1*. From the *main* point of view, you will only have to invoke a single function as follows;

```
loopR1(1, 3);  …  <  Yes, you will have three LoopR1-3 >
```

The parameters mean that we start iterating R1 at 1 and go up to 3 inclusive. Now, as we implement this recursive function, we are going to make sure that;

- It implements loops #2 and #3 as above, invoking the payoff function every time
- It then calls itself recursively by incrementing by one its first parameters
- It keeps doing so until the first parameter is greater than the second.

At this point, you have replaced one of the nested loops by a recursive function. Let's now replace the two other nested loops by calls to recursive functions. We will end up with three recursive functions;

- a recursive function named *loopR1* to replace the loop iterating over all possible values for *R1*
- a recursive function named *loopR2* to replace the loop iterating over all possible values for *R2*
- a recursive function named *loopR3* to replace the loop iterating over all possible values for *R3*

Replace one loop at a time by its corresponding recursive function and test your program extensively before you start modifying more. You are free to decide what parameters the loopR2 and loopR3 functions need to do their job.