St Petersburg Paradox - Code Efficiency

When using R to approach a problem, it is often tempting to dive right in and start coding, but this can often result in code that is slow to run, and horribly inefficient, despite there being much simpler and quicker ways of dealing with the problem. Recently I came across a rather nice example of this, when addressing the St Petersburg Paradox (see the Wikipedia page here)

The Paradox

The paradox involves a casino game which works as follows. A (fair) coin is flipped, and if it lands on heads, the game ends here and the player wins £2. If the coin however lands on tails, the game proceeds to round two. In round two the coin is flipped again, and if it lands on heads, the game ends with a payoff of £4 to the players, and if it lands on tails the game proceeds to round three. The potential payoff to the player doubles in each round, whilst the probability of this happening halves (due to the increasingly unlikeliness of flipping many consecutive tails). In the original paradox this can continue for an infinite number of rounds.

<table>
<thead>
<tr>
<th>Round</th>
<th>Payoff</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1,]</td>
<td>1</td>
<td>0.50000</td>
</tr>
<tr>
<td>[2,]</td>
<td>2</td>
<td>0.25000</td>
</tr>
<tr>
<td>[3,]</td>
<td>3</td>
<td>0.12500</td>
</tr>
<tr>
<td>[4,]</td>
<td>4</td>
<td>0.06250</td>
</tr>
<tr>
<td>[5,]</td>
<td>5</td>
<td>0.03125</td>
</tr>
</tbody>
</table>

The paradox arises when you ask how much should you be willing to pay for a chance to enter this game? You have half a chance of winning £2, a quarter of a chance of winning £4, and eight of a chance of winning £8 and so on until infinity. £2? £20? £20000?

Expected Value

To answer this, we use the expected value of the game. To illustrate this we can use the simple example of rolling dice. Let’s say the dice is not fair, so there is half a chance of rolling a six, and a 10% chance of rolling each of the remaining five numbers. Let’s then say that the payoff for rolling a six is £20, and the payoff for rolling any other number is 0. So 50% of the time we get £20 (for rolling a six), and 50% of the time we nothing (for rolling any other number). The expected value of this game is the amount we win if each event occurs, weighted by the chance of each of these events occurring. So here, our expected value is 0.5 x £20 + 0 x £1 , which equals £10. This is how much we expect to win (on average) by playing this game.

St Petersburg Expected Value

Going back to the St Petersburg Paradox, our expected value is again the amount we win from each event, multiplied by the chance of each event occurring. So for the first few events (shown in the table) this is (2 x 0.5) + (4 x 0.25) + (8 x 0.125) + (16 x 0.0625) + (32 x 0.03125). We then notice that each of these values = 1, because the probability of winning any amount is 1 / the amount. As such, our expected value, the amount we expect to win on average, is 1 + 1 + 1 + 1 + 1. If the game goes on until infinity, we can see that the expected value is 1 + 1 + 1... all the way to infinity, so we should be willing to pay an infinite amount of money to play this game! This is clearly a ridiculous answer, but what would be a more reasonable answer? The 18th century French mathematician Comte de Buffon approached the problem by paying a child to play 2048 trials of the game, but we can do better by making a computer play the game.
The Finite game

The game we are going to play is a slight variation of the St Petersburg paradox in that it is finite meaning the casino has only a certain amount of money to give you. We will call this \( W \). This means that there is a maximum number of rounds, which we will call \( N \) which we can calculate as the rounded down (or floor) of \( \log_2 W \). For example, if \( W = 20 \), \( N = \text{floor}(\log_2 20) = 4 \). In this instance, 50 doesn’t fall on the normal payoff scale (it is between 16 and 32) so the rules of the game state that if the game reaches the round where the casino has to pay out more than the size of their bank (\( W \)) they simply pay out their entire bank.

The code

So what is the intuitive way of approaching this problem? It is to do exactly what Buffon did ~250 years ago, but with a computer simulating trials of the game, rather than paying a child to manually flip a coin. To start with we can set our two parameters, which is the number of simulations \( \text{sims} \) and the size of our casino’s bank \( W \), and calculate the maximum number of rounds \( N \).

\[
\text{sims} \leftarrow 1000000 \quad # \text{Number of simulations} \\
W \leftarrow 1000000 \quad # \text{size of bank} \\
N \leftarrow \text{floor}(\log(W, 2)) \quad # \text{maximum number of rounds} 
\]

First we can make a simple vector to store the outcome of each trial

\[
\text{output} \leftarrow \text{c()} 
\]

Now we can make the loop that will go through each round number, work out what the outcome of the flip of the coin in that round is, and either record the payoff corresponding to that round and stop (if we flip heads) or proceed straight to the next round (if we flip tails). We can use a random number between 0 and 1 to simulate a coin flip (if it is > 0.5, we call that heads) and the payoff is simply \( 2^i \) where \( i \) is the round we are currently in (i.e., if we flip heads in round 3, the payoff is \( 2^3 = 8 \)).

\[
\text{for}(i \text{ in } 1:(N+1))\{ \\
\quad \# \text{Flip coin, if heads store payoff in output matrix} \\
\quad \text{if(runif(1) } > 0.5)\{\text{output } \leftarrow 2^i\} \quad # \text{payoff } = 2^\text{round} \\
\quad \text{if}(i == (N+1))\{\text{output } \leftarrow W\} \quad # \text{Or whole bank} \\
\quad \# \text{if we reach the last round} \\
\quad \text{if(output } > = 2)\{\text{break}\} \quad # \text{if we win, stop flipping} \\
\}
\]

This will give us a payoff for one trial of the game. In order to make many attempts of the game, we need an outer loop to run through this procedure many times. For this we will need a matrix to record the output of each trial

\[
\text{output} \leftarrow \text{matrix}(0, \text{sims}, 1) 
\]

We can then make our loop. Note that we have altered the inner loop slightly to record in a certain cell of the matrix, as we will be generating a new payoff for each trial. For a more detailed explanations of using loops, see here.

\[
\text{for}(j \text{ in } 1:\text{sims})\{ \quad # \text{Outer loop } j \text{ for many trials} \\
\text{for}(i \text{ in } 1:(N+1))\{ \quad # \text{Inner loop } i \text{ for each round} \\
\quad \text{if(runif(1) } > 0.5)\{\text{output}[j, 1] } \leftarrow 2^i\} \quad # \text{payoff } = 2^\text{round} \\
\quad \text{if}(i == (N+1))\{\text{output}[j, 1] } \leftarrow W\} \quad # \text{Or whole bank} \\
\quad \# \text{if we reach final round} 
\} 
\]
if(output[j, 1] >=2){break}  # if we win, stop flippinh
}
mean(output[, 1])  # output the mean payoff of all trials

## 20.09033

We can time how long this takes using the tictoc package which is really simple to use. First use the following code to install (if required) and load the package:

```r
list.of.packages <- c("tictoc")
new.packages <- list.of.packages[!(list.of.packages %in% installed.packages())[,"Package"]]
if(length(new.packages)) install.packages(new.packages)
library(tictoc)
```

To use it simply write tic() where you wish to start timing, and toc() at the end. Now we can see how long it takes to run 1 million trials of the game, when the maximum payoff is £1000000.

```r
tic()  # start timing here
output <- matrix(0, sims, 1)
for(j in 1:sims){  # Outer loop j for many trials
  for(i in 1:(N+1)){  # Inner loop i for each round
    if(runif(1) >= 0.5) {output[j, 1] <- 2^i}  # payoff = 2^round
    if(i == (N+1)) {output[j, 1] <- W}  # Or whole bank
      # if we reach final round
    if(output[j, 1] >=2){break}  # if we win, stop flippinh
  }
}
mean(output[, 1])  # output the mean payoff of all trials
toc()  # stop timing here
```

Probably ~15 seconds, not bad! But we can do so much better. We have approached the problem like a human, not a computer. We’ve logically programmed the steps that need to happen, in the order that they need to happen, but much of this is unnecessary. We are wastefully creating many random numbers to decide whether we should progress to the next round, and using a loop to go through every round of the game. A computer can do things far more effectively. Let’s start again by redefining our starting parameters

```r
sims <- 1000000  # Number of simulations
W <- 1000000  # size of bank
N <- floor(log(W, 2))  # maximum number of rounds
```

Now let’s ask; what are the potential payoffs of this game? well we know from earlier that the formula for the payoff of each round is simply $2^i$ where $i$ is the round we are currently in. We can easily generate this list of possible payoffs, $2^\text{seq}(1:N)$ will generate this. We must also remember that the final possible payoff is simply the size of the casino’s bank, $W$, which we can add to our list by using a vector

```r
payoffs <- c(2^\text{seq}(1:N),W)
```

We can now sample from these payoffs to determine the payoff of each trial, without needing to simulate up to $N$ random coin flips. We could do this using the sample function, for example like this

```r
winnings <- \text{sample}(payoffs, sims, replace = \text{TRUE})
mean(winnings)
```

## 102423.8

Here we are generating a sample of the potential payoffs, doing so for $sims$ number of trials, and replacing
back each payoff into the available pool once we have sampled it.

The problem of course is that it is not equally likely that we win each of these payoffs, so we can’t just randomly sample them. However, each payoff has an associated probability, which we can use to make a biased sample of the payoffs that is exactly equivalent to playing the game many times, without having to waste time generating random coin flips; we can simply generate a single number, the payoff, for each round.

The biased sample is generated by simply attaching probabilities to our sample, a simple example could be as follows:

```r
payoffs <- c(0, 1) # vector of possible payoffs
probs <- c(0.2, 0.8) # vector of associated probabilities
# we are making the probability of winning 0 = 0.2
# and the probability of winning 1 = 0.8
sample(payoffs, 10, prob = probs, replace = TRUE) # 10 samples
```

```
## [1] 1 1 1 1 1 1 1 1 1 0
```

We can use this for the St Petersburg paradox; we already know what the vector of possible payoffs is `payoffs <- c(2^seq(1:N),W)` and the probabilities are simply 1 / each payoff `probabilities <- c(1/(2^seq(1:(N+1))))`. Putting these into the sample function, we can then make our biased sample, with the size of the sample equal to the number of trials of the game we want to run, and then simply calculate the mean of that sample. We will time this code with `tictoc` to see how quick it is compared to the previous method.

```r
tic()
payoffs <- c(2^seq(1:N),W)
probabilities <- c(1/(2^seq(1:(N+1))))
winnings <- sample(payoffs, sims, prob = probabilities, replace = TRUE)
mean(winnings)
toc()
```

It is several hundred times quicker! And we can even improve a little on that, by running the code on a single line

```r
 tic()
 mean(sample(c(2^seq(1:N),W), sims, prob = c(1/(2^seq(1:(N+1))))), replace = TRUE))
toc()
```

This approach is so much faster, with the key being to think about how you can recreate a ‘real’ process like flipping a coin many times, with a mathematical equivalency, like taking a sample of possible outcomes, biased by their likelihood. Although this approach would be silly if we were to manually play the game; it would be like the French mathematician taking a million pieces of parchment, and writing the payoffs on a proportion of them equal to the likelihood of them occurring, it works so much faster when coding. If you try bumping up the number of trials you will see that the time difference between the two approaches is even greater.