Random Numbers

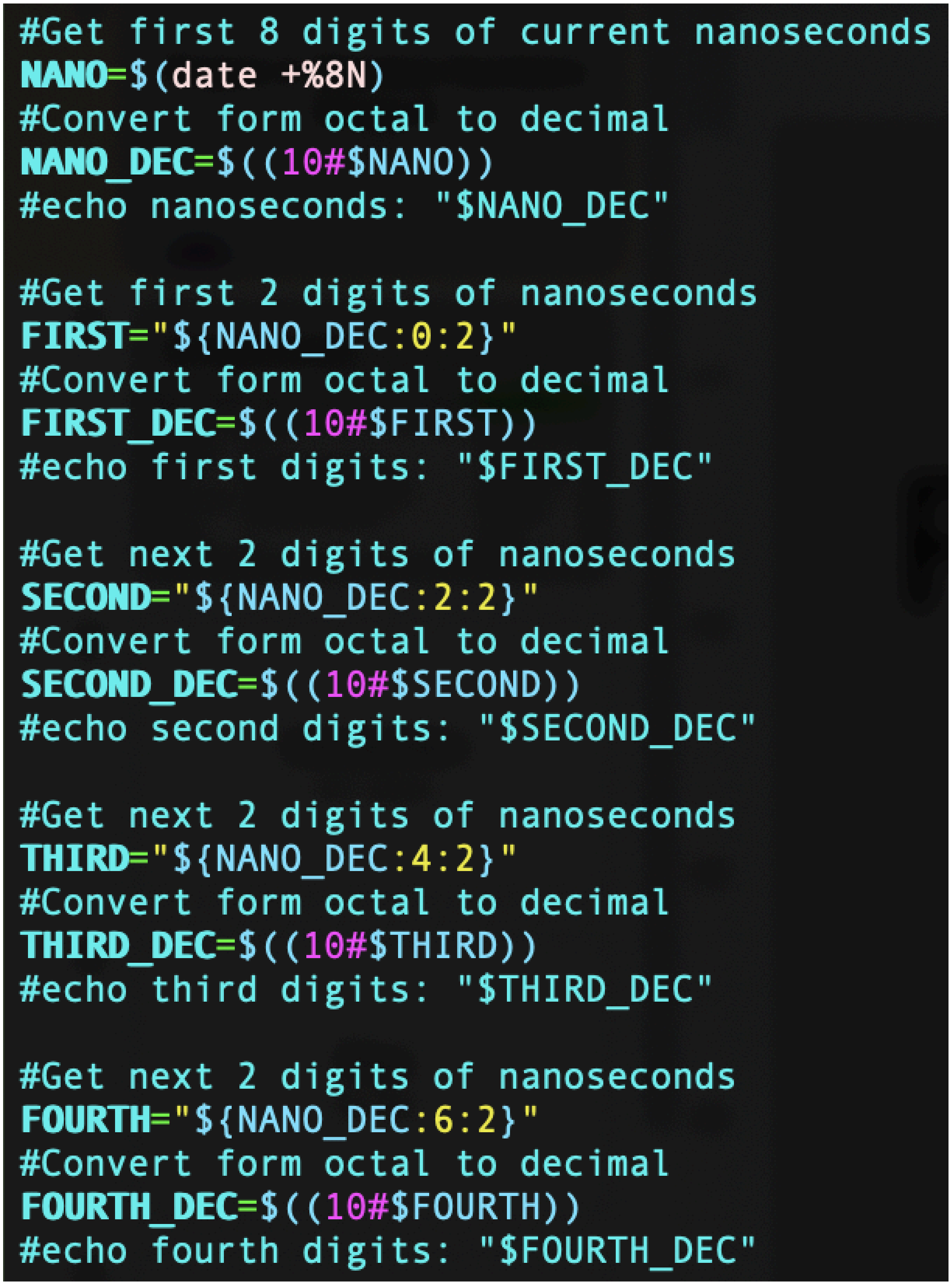
CEO Nadeen - Introduction

For this project, my group and I have decided to work on generating random numbers. This is done through creating an algorithm which implements the idea of the lingerie congruential generator (LCG). We decided to use this algorithm as it is efficient and simple, and also easy to debug. After generating the numbers, we would use them in our application. The application we have chosen to implement is similar to the idea of a lottery, where one person checks if the numbers generated are exactly the same as their own.

PM Alfie - Plan

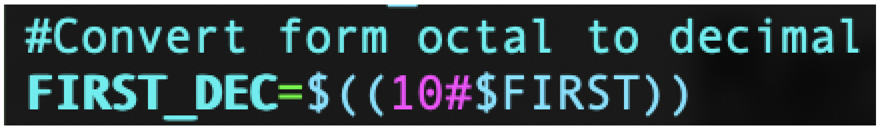
Dev0 Ben - PRNG Implementation

As Developer 0 my job was to implement the PRNG algorithm that my group chose. We chose to implement the linear congruential generator (LCG) algorithm using shell script. In order to make the outputs of this algorithm as random as possible, I decided that the best approach was to NOT have the variables of the LCG algorithm be determined by user input. Instead they would depend on the nanoseconds of the current time in which the LCG algorithm was executed. I pulled the first 8 digits of the nanoseconds of the current time using the syntax “date %8N” and split the digits into pairs of two, to create 4 two digit numbers. I then assigned those numbers into variables.



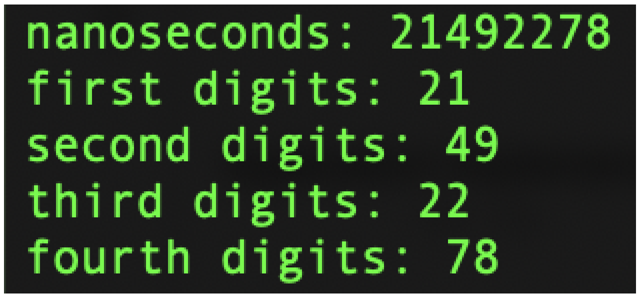
```
#Get first 8 digits of current nanoseconds
NANO=$(date +%8N)
#Convert form octal to decimal
NANO_DEC=$((10#$NANO))
#echo nanoseconds: "$NANO_DEC"

#Get first 2 digits of nanoseconds
FIRST="${NANO_DEC:0:2}"
#Convert form octal to decimal
FIRST_DEC=$((10#$FIRST))
#echo first digits: "$FIRST_DEC"

#Get next 2 digits of nanoseconds
SECOND="${NANO_DEC:2:2}"
#Convert form octal to decimal
SECOND_DEC=$((10#$SECOND))
#echo second digits: "$SECOND_DEC"

#Get next 2 digits of nanoseconds
THIRD="${NANO_DEC:4:2}"
#Convert form octal to decimal
THIRD_DEC=$((10#$THIRD))
#echo third digits: "$THIRD_DEC"

#Get next 2 digits of nanoseconds
FOURTH="${NANO_DEC:6:2}"
#Convert form octal to decimal
FOURTH_DEC=$((10#$FOURTH))
#echo fourth digits: "$FOURTH_DEC"
```

Before I could assign values to the variables, I had to change the numbers from octal to decimal. This was necessary because there were times when the two digit number would begin with a ‘0’ (for example ‘09’). When this would occur the code would treat the number as octal, making it impossible to use it for computation. I converted the number from octal to decimal using this syntax:



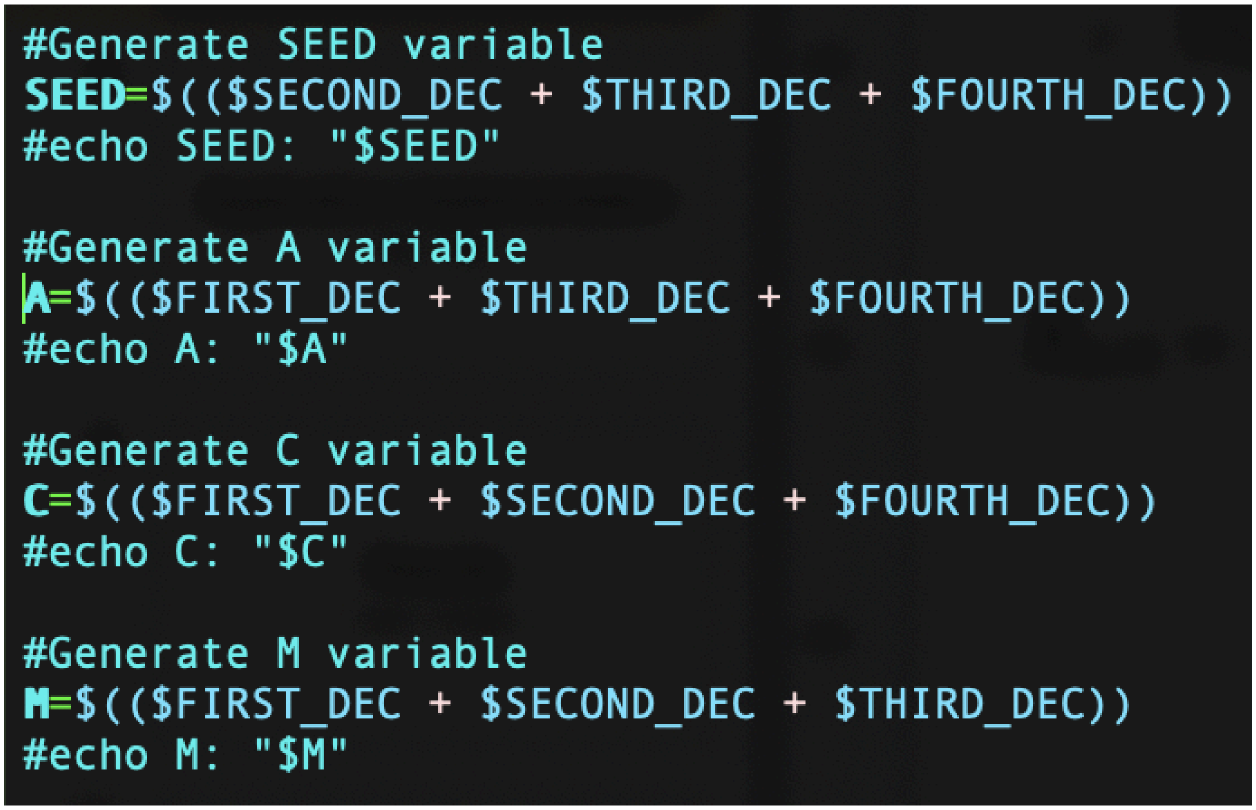
```
#Convert form octal to decimal
FIRST_DEC=$((10#$FIRST))
```

Here is an example output of the initial values that we will manipulate and use in the LCG algorithm, again all derived from the current nanoseconds:



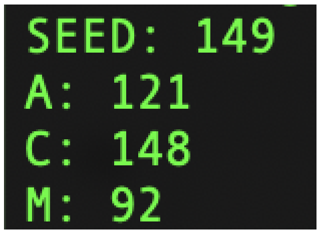
```
nanoseconds: 21492278
first digits: 21
second digits: 49
third digits: 22
fourth digits: 78
```

Next is to compute the values of the variables of the LCG algorithm. Here is my approach to this step:



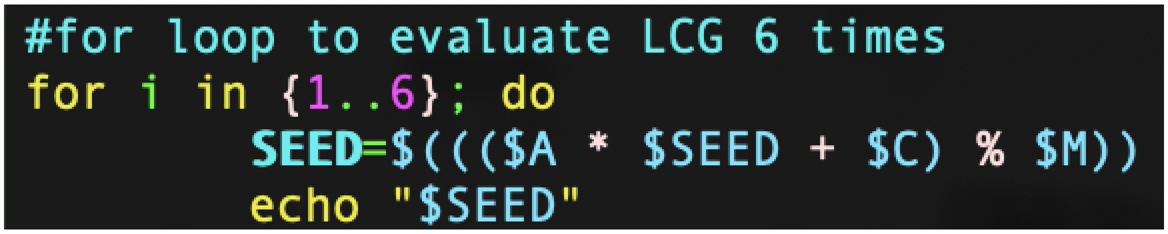
```
#Generate SEED variable
SEED=$(($SECOND_DEC + $THIRD_DEC + $FOURTH_DEC))
#echo SEED: "$SEED"

#Generate A variable
A=$(($FIRST_DEC + $THIRD_DEC + $FOURTH_DEC))
#echo A: "$A"

#Generate C variable
C=$(($FIRST_DEC + $SECOND_DEC + $FOURTH_DEC))
#echo C: "$C"

#Generate M variable
M=$(($FIRST_DEC + $SECOND_DEC + $THIRD_DEC))
#echo M: "$M"
```

The addneds for the 'seed' value, the 'a' value, the 'c' value, and the 'm' value were all chosen at random by me. The addends of these variables can be in any combination, this is just what I chose. If we use the example output I had shown previously and we apply this next step, then we would get this output:



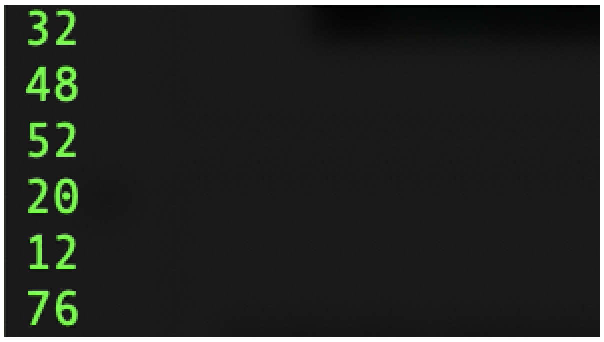
```
SEED: 149
A: 121
C: 148
M: 92
```

Next we move on to the final step of the algorithm which is the recursive computation. The way the LCG algorithm works is that the 'a' value, the 'c' value, and the 'm' value all stay the same, but the input 'seed' value changes each iteration. Basically, whatever the algorithm outputs is treated as the new 'seed' value for the next iteration.



```
#for loop to evaluate LCG 6 times
for i in {1..6}; do
        SEED=$((($A * $SEED + $C) % $M))
        echo "$SEED"
```

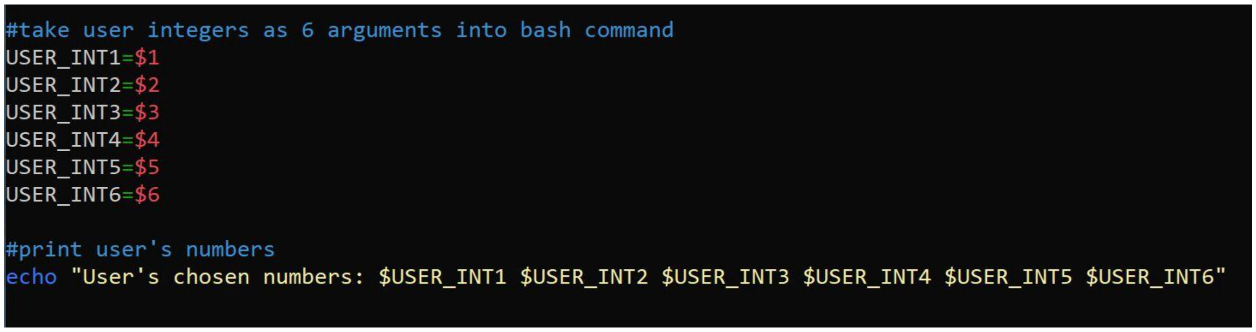
We loop through this process 6 times and get this output of pseudorandom numbers based on the example input we have been using (referenced above):



```
32
48
52
20
12
76
```

Dev1 Tyler - Application Implementation

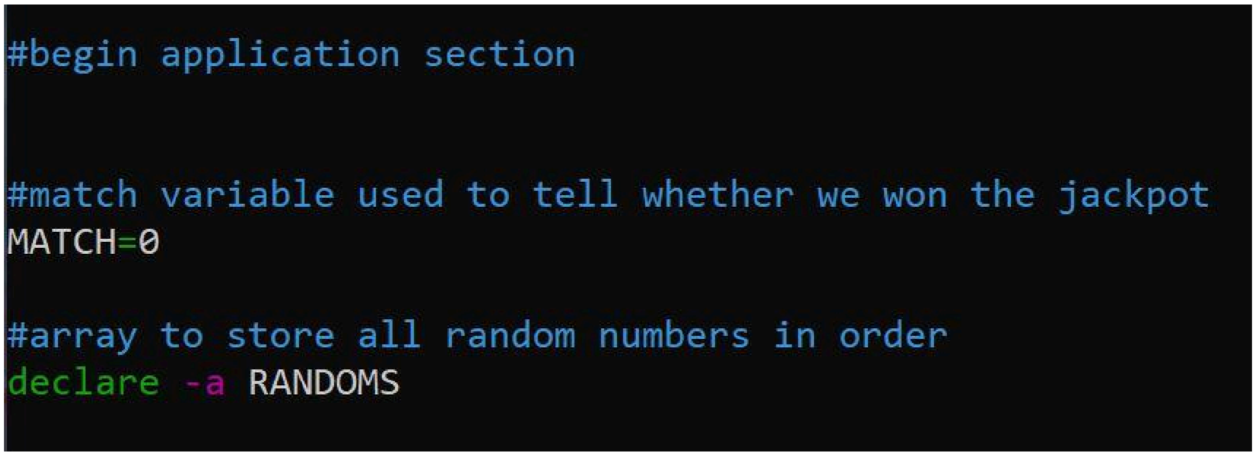
For the application implementation, I started with taking user input. To implement our lottery simulator, we needed to take 6 integers from the user.



```
#take user integers as 6 arguments into bash command
USER_INT1=$1
USER_INT2=$2
USER_INT3=$3
USER_INT4=$4
USER_INT5=$5
USER_INT6=$6

#print user's numbers
echo "User's chosen numbers: $USER_INT1 $USER_INT2 $USER_INT3 $USER_INT4 $USER_INT5 $USER_INT6"
```

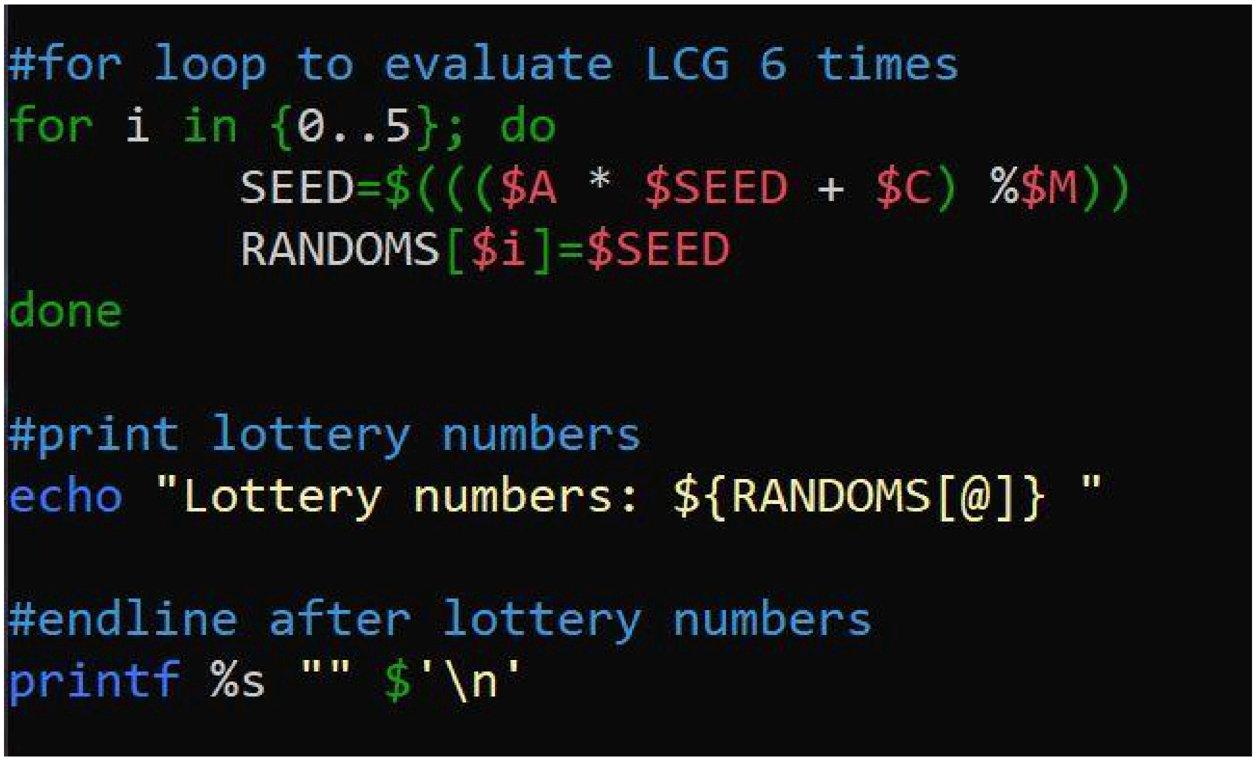
I stored the first six command line arguments into variables and printed them to the standard output.

Next, I needed to use some variables to implement the functionality of the program. First, I created a MATCH variable which operates as a boolean variable that is 0 when we haven't had a mismatch and 1 when we have at least 1 mismatch. I also created a RANDOMS array to store our six random numbers we generate.



```
#begin application section


#match variable used to tell whether we won the jackpot
MATCH=0

#array to store all random numbers in order
declare -a RANDOMS
```

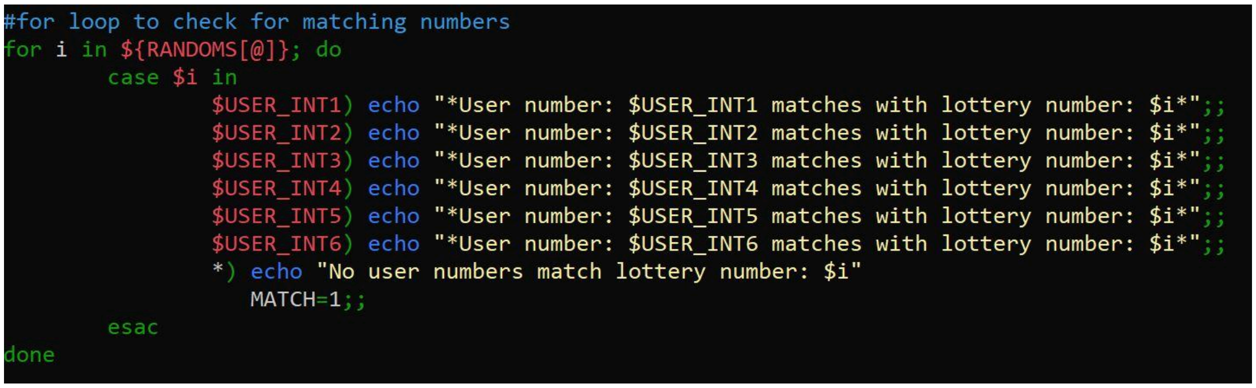
Now, we generate six random numbers using our LCG algorithm.



```
#for loop to evaluate LCG 6 times
for i in {0..5}; do
        SEED=$((($A * $SEED + $C) %$M))
        RANDOMS[$i]=$SEED
done

#print lottery numbers
echo "Lottery numbers: ${RANDOMS[@]} "

#endline after lottery numbers
printf %s "" $'\n'
```

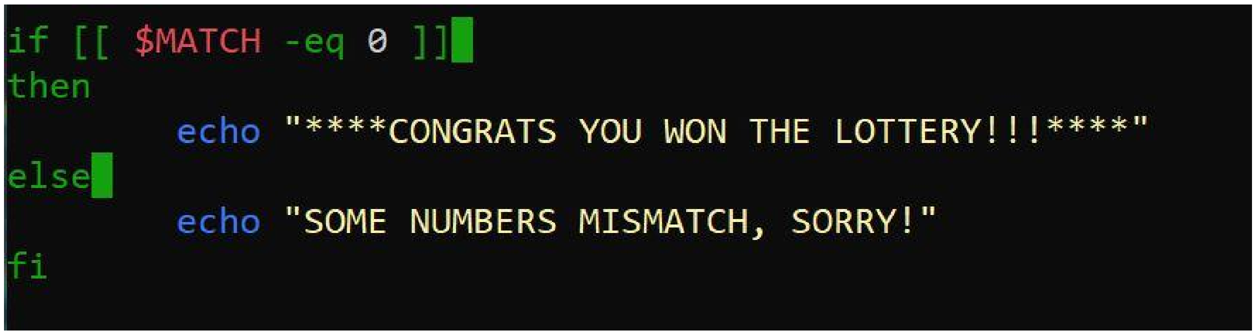
SEED is our random number generated by our LCG algorithm and we store it in the ith index of the RANDOMS array. We then use the “${RANDOMS[@]}” syntax to print each element in the array with a space in between.

After, we need to compare the user supplied integers with the lottery numbers we generated randomly. To do this, I used a case statement on each value in the RANDOMS array. Each random number is stored in the i variable and compared with the user integers in the case branches. If none match, we print that none match and we set MATCH to 1.



```
#for loop to check for matching numbers
for i in ${RANDOMS[@]}; do
        case $i in
                $USER_INT1) echo "*User number: $USER_INT1 matches with lottery number: $i*";;
                $USER_INT2) echo "*User number: $USER_INT2 matches with lottery number: $i*";;
                $USER_INT3) echo "*User number: $USER_INT3 matches with lottery number: $i*";;
                $USER_INT4) echo "*User number: $USER_INT4 matches with lottery number: $i*";;
                $USER_INT5) echo "*User number: $USER_INT5 matches with lottery number: $i*";;
                $USER_INT6) echo "*User number: $USER_INT6 matches with lottery number: $i*";;
                *) echo "No user numbers match lottery number: $i"
                   MATCH=1;;
        esac
done
```

Finally, we need to print the results of our matching.

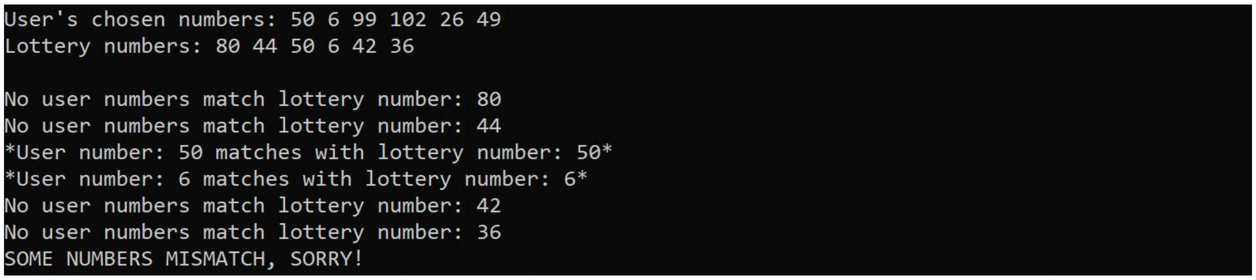


```
if [[ $MATCH -eq 0 ]]
then
        echo "****CONGRATS YOU WON THE LOTTERY!!!****"
else
        echo "SOME NUMBERS MISMATCH, SORRY!"
fi
```

We print congrats if there are no mismatches and sorry if you have mismatches and lose the lottery.

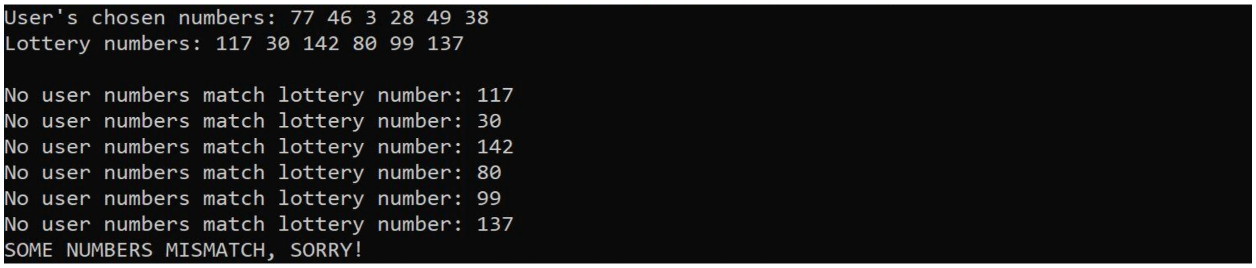
Tester Christine - Results

For this project, I was tasked with testing the program and giving feedback on our results. As you can see in this first example, two of our inputted numbers matched two of the randomized lottery numbers, and the output states exactly that. Even though two are the same, the final line of code still states that some numbers mismatch, and the user did not win the lottery.

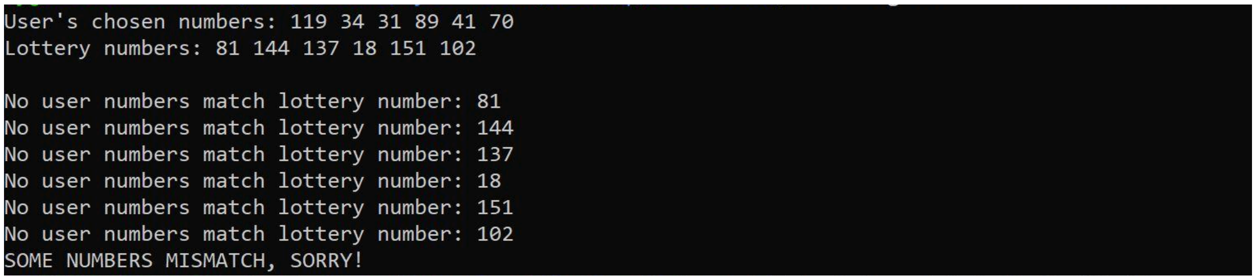


```
User's chosen numbers: 50 6 99 102 26 49
Lottery numbers: 80 44 50 6 42 36

No user numbers match lottery number: 80
No user numbers match lottery number: 44
*User number: 50 matches with lottery number: 50*
*User number: 6 matches with lottery number: 6*
No user numbers match lottery number: 42
No user numbers match lottery number: 36
SOME NUMBERS MISMATCH, SORRY!
```

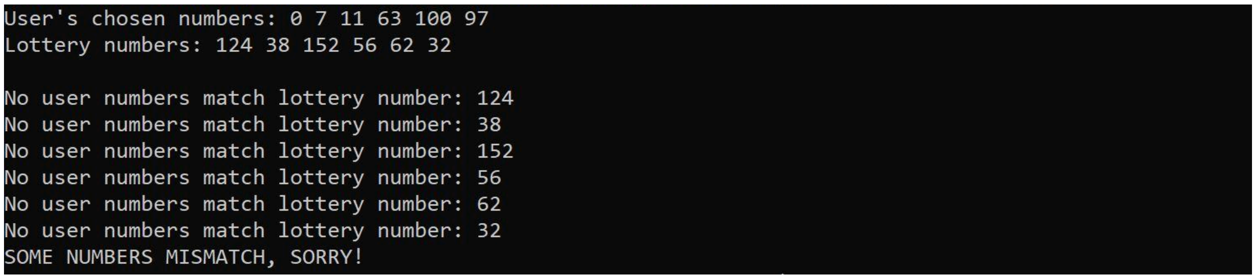
The next three examples are a more common example that I typically ran into, in which none of the inputted numbers matched the lottery numbers.



```
User's chosen numbers: 77 46 3 28 49 38
Lottery numbers: 117 30 142 80 99 137

No user numbers match lottery number: 117
No user numbers match lottery number: 30
No user numbers match lottery number: 142
No user numbers match lottery number: 80
No user numbers match lottery number: 99
No user numbers match lottery number: 137
SOME NUMBERS MISMATCH, SORRY!
```



```
User's chosen numbers: 119 34 31 89 41 70
Lottery numbers: 81 144 137 18 151 102

No user numbers match lottery number: 81
No user numbers match lottery number: 144
No user numbers match lottery number: 137
No user numbers match lottery number: 18
No user numbers match lottery number: 151
No user numbers match lottery number: 102
SOME NUMBERS MISMATCH, SORRY!
```



```
User's chosen numbers: 0 7 11 63 100 97
Lottery numbers: 124 38 152 56 62 32

No user numbers match lottery number: 124
No user numbers match lottery number: 38
No user numbers match lottery number: 152
No user numbers match lottery number: 56
No user numbers match lottery number: 62
No user numbers match lottery number: 32
SOME NUMBERS MISMATCH, SORRY!
```

In regard to the question, “Are the lottery numbers truly randomized” our response would be that since the values of the variables for the algorithm were derived from nanoseconds - the outputs are basically random at that point in time. Because of this, it is humanly impossible to execute the algorithm to get a desired output, or win the lottery.
However, since this is such a simple and recursive algorithm, there’s a possibility that there will be repeated outcomes which contradicts the randomized effect.

CEO Nadeen - Conclusion

In conclusion, we were able to get the algorithm to work just as we wanted it to, and the lottery application worked perfectly. Therefore, we were successful in reaching our goal. One challenge we faced was when running the algorithm many times, we noticed that a few times the same numbers were generated (this is one downfall for using the LCG since it is simple and does not eliminate the possibility of having duplicate numbers). Since this wasn’t technically an error, and specifically since our application of the lottery in many times allows for duplicates, we did not change any of our algorithm however we did think of alternate solutions for the future if we ever needed to use this application. One solution would be implementing case statements, to check if one number matched another before printing out the six random numbers.