CSE 3100 Systems Programming
Lab #2

Welcome to Lab 2! At the start of each lab, you will receive a document like this detailing the tasks you are to complete. Read it carefully. Try your best to finish labs during the lab session. However, labs are not due right after each lab period.

1 Exercise: FizzBuzz (15 Points)

Open `fizzbuzz.c` with your favorite text editor. In the main function, you'll need to add code that loops i between `START` and `END`:

- If i is divisible by both 3 and 5, print "FizzBuzz"
- If i is divisible by just 3, print "Fizz"
- If i is divisible by just 5, print "Buzz"
- If i is divisible by neither 3 nor 5, print i

Here's the first 20 lines of the correct output:

```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
17
Fizz
19
Buzz
```

You can compile and run it as follows:

```
$ cc -o fizzbuzz fizzbuzz.c
$ ./fizzbuzz
```

2 Exercise: Square Root (25 Points)

We're going to implement a square root approximation called the *Babylonian Method*. Given a number s, and an initial guess x_0, we can approximate $\sqrt{s}$ as follows:

1. Pick a gues $x_0 \approx \sqrt{s}$
2. Compute $x_{n+1} = \frac{1}{2}\left(x_n + \frac{s}{x_n}\right)$
3. Repeat step 2 until stopping

For example, suppose $s = 10$, and our initial guess is 4. We start with $x_0 = 4$, and then compute x_1 as:

$$x_1 = \frac{1}{2}\left(4 + \frac{10}{4}\right) = 3.25 \tag{1}$$

Then we compute x_2 as:

$$x_2 = \frac{1}{2}\left(3.25 + \frac{10}{3.25}\right) = 3.16346153846 \tag{2}$$

Take this an square it, and we get: $3.16346153846^2 = 10.00748$, which is a pretty good approximation. Open `square.c`. You need to implement the `relError` function, and finish the `main` function.

2.1 `relError`

Given s and $x \approx \sqrt{s}$, compute:

$$\frac{|x^2 - s|}{s} \tag{3}$$

This is the relative error. Basically, it compares s to x^2.

2.2 `main` function

We first need to compute a guess $x_0 \approx \sqrt{s}$. Start $x_0 = 1$, and increment x_0 until $x_0^2 \geq s$. You can use a loop to do this:

```
int i = 1;
while(i*i < s){
   i = i + 1;
}
```

Then, iterate using the formula $x_{n+1} = \frac{1}{2}\left(x_n + \frac{s}{x_n}\right)$ until the relative error (computed using `relError`) is less than 0.001.

Once you're done with the code, you can compile the code as follows:

```
$ cc -o square square.c
```

Here, I computed the square root of 8650:

```
$ ./square
Enter s to take square root of: 8650
Square root: 93.010638
```

The correct answer is approximately 93.005376.

3 Exercise: Binomial (30 Points)

Here, you will use the recursive form of the Binomial Coefficient to compute $\binom{n}{k}$:

$$\binom{n}{0} = \binom{n}{n} = 1$$
$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

Open `binomial.c`, and finish the `binomial` function. You can compile and run as follows:

```
$ cc -o binomial binomial.c
$ ./binomial
Enter n: 10
Enter k: 4
10 choose 4 = 210
```

4 Monte Carlo and `#define`

Here, we'll use random sampling to compute π and $\ln a$. This is meant to introduce a class of methods called *Monte-Carlo Methods*, and the use (and limitations) of `#define`'s in C. While using random sampling for computing π and $\ln a$ is unecessary, because there are deterministic algorithms that compute those values faster and more accurately, it is instructive.

4.1 Example: Logarithm

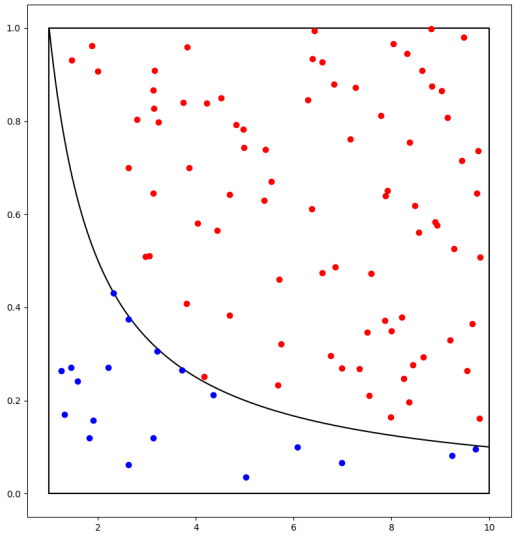
This is a worked example to help you on the following question The natural logarithm $\ln a$ can be defined in several equivalent ways. One is the area under the curve of the function $f(x) = \frac{1}{x}$ from 1 to a:

$$\ln a = \int_1^a \frac{1}{x} dx \tag{4}$$

$\frac{1}{x}$ is bounded by the box $\{(1,0),(1,1),(a,0),(a,1)\}$. We pick a set of points such that $1 \leq x < a$ and $0 \leq y < 1$. We check if each point (x,y) lies under the curve $\frac{1}{x}$; that is, we check if $y \leq \frac{1}{x}$. If we generated N points, and m points fall under the curve, then the area under the curve (and thus $\ln a$) is approximately:

$$\ln a \approx (a-1)\frac{m}{N} \tag{5}$$

Here is an illustration of this for $a = 10$. There are 100 points, and 81 fall above $\frac{1}{x}$:



The code to do this is in `logarithm.c`. You'll see a bunch of `#define`s near the top of the file. Here's a summary of what each of these do:

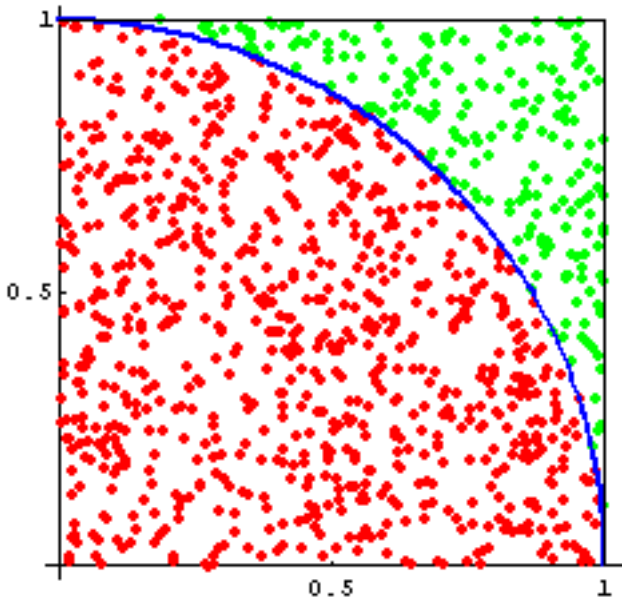
- `RANDPOINT(a, b)` This generates a random `double` x such that $a \leq x < b$
- `RANDX(k)` This uses `RANDPOINT` to generate a `double` between 1 and k.
- `RANDY` This uses `RANDPOINT` to generate a `double` between 0 and 1.
- `SAMPLE(k)` This uses `RANDX(k)` and `RANDY` to generate x and y, and check if $y \leq \frac{1}{x}$

You can compile and run it as follows:

```
$ cc -o logarithm logarithm.c
$ ./logarithm
Enter number to take ln of: 10.0
Approximate logarithm: 2.328300
```

4.2 Exercise: Throwing Darts (30 Points)

Imagine the upper right hand quarter of a unit circle (a circle centered at $(0, 0)$ with radius 1) being wrapped in a square with the corners $\{(0, 0), (1, 0), (1, 1), (0, 1)\}$. This box has area of 1. If we "throw darts" at the box, and count the fraction of "darts" that land within the circle, we get an estimate of the circle's area. Since we know that the circle has radius 1, we can estimate the value of π. Here's an image depicting this:



By throw darts, I mean we generate points (x, y) randomly within the square. That is, both $0 \leq x \leq 1$ and $0 \leq y \leq 1$. If we generate N points, and n fall within the upper quadrant of the circle, we get:

$$\pi \approx \frac{4n}{N} \tag{6}$$

Open up `darts.c`, and you'll see four defines at the top. This is what I *intended* each define to do:

- **N** is the number of points the program will sample.
- **RANDPOINT**. This calls the `rand` function, which generates a random integer between 0 and **MAX**, which is a very large number. Then, it uses the mod function % to get the remainder of dividing the number by **MAX**, which puts it in the range 0 to **MAX - 1**. Then, it divides this by **MAX**. In probability terms, it (approximately) samples from the uniform distribution $U(0, 1)$. The sampling isn't completely uniform, but it's close enough for our purposes.
- **DISTSQUARED** takes in two numbers x and y, and computes $x^2 + y^2$.
- **SAMPLE** "calls" **RANDPOINT** twice to generate x and y, and "calls" **DISTSQUARED** to compute the squared distance between $(0, 0)$ and (x, y). Then, it checks if this distance is less than or equal to 1.

However, there's a problem. If you run this, the estimate of π is very different from the correct value of π:

```
$ cc -o darts darts.c
$ ./darts
pi: 3.697600
```

Let's investigate why. Run the following command:

```
cc -o darts.pre darts.c -E
```

This tells the compiler to run the pre-processor, and output the results to `darts.pre`. Then, open up `darts.pre` with a text editor or view it with `less`. If you go to the bottom, you'll see:

```
int main(int argc, char* argv[]){
  srand(10);
  unsigned int j = 0;
  for(unsigned int i = 0; i < 1000; i++){
    if((((1.0*(rand() % 2147483647)/2147483647)*(1.0*(rand() % 2147483647)/2147483647)
    + (1.0*(rand() % 2147483647)/2147483647)*(1.0*(rand() % 2147483647)/2147483647)) <= 1)){
      j++;
    }
  }
  double pi = 4.0*j/1000;
  printf("pi: %f\n", pi);
}
```

You'll notice that `rand` gets called four times. But we're only generating two values for each sample, so that doesn't make sense. Here's the problem: **RANDPOINT** is substituted for `x` in `x*x` (and `y*y`), and then `(1.0*(rand() % 2147483647)/2147483647)` is substituted for **RANDPOINT**, yielding `(1.0*(rand() % 2147483647)/2147483647)*(1.0*(rand() % 2147483647)/2147483647)` for each **RANDPOINT**.

The best way to fix this is to move the **DISTSQUARED** and **SAMPLE** into functions.

Don't forget to submit to Mimir!