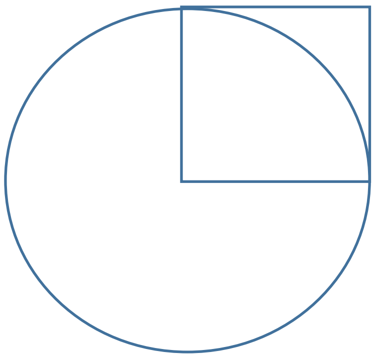
CS 315 - Day 03, Monte Carlo Methods and Optimization

Square on board

Put ***N*** marks in square

Put in quarter circle



$$\frac{Area\ of\ quarter\ circle}{Area\ of\ square} \approx \frac{Number\ of\ points\ in\ quarter\ circle}{N}$$

$$\frac{\frac{1}{4}\pi r^2}{r^2} \approx \frac{Number\ of\ points\ in\ quarter\ circle}{N}$$

Wait, this leads to an approximation for π

$$\pi \approx 4 * \frac{Number\ of\ points\ in\ quarter\ circle}{N}$$

This is called the Monte Carlo method of simulation, https://www.youtube.com/watch?v=7TqhmX92P6U

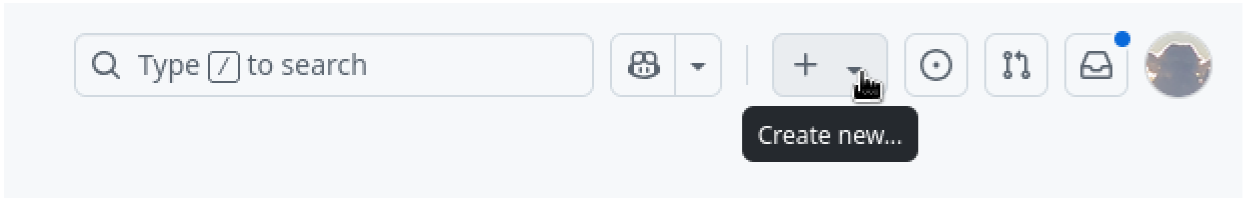
We will run a simulation of this using a Jupyter Notebook running on Google Colab. In order to save our work, we first need a GitHub account.

If you are new to GitHub, you will need to set up a GitHub account at https://github.com/ .

As you are setting up your account, you'll want to consider what username to use. ***Choose carefully***, because your web files will be in a repository that starts with your username. As a previous student stated, GitHub is your resume! Also, just choose ***lowercase***.

Note that the following is based on https://swcarpentry.github.io/git-novice/07-github.html .

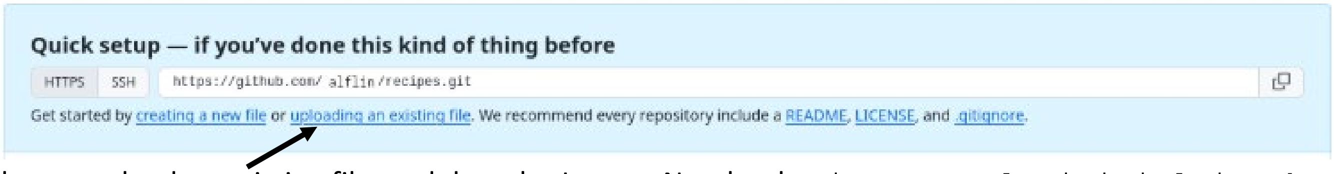
Once you have setup a GitHub account, and you're logged in to GitHub, then click on the icon in the top right corner to create a new repository called cs315:



Name your repository "cs315" and then click "Create Repository".

As soon as the repository is created, GitHub displays a page with a URL and some information on how to configure your local repository.

You will want to have a file in the repository, so under the "Quick Setup"



Quick setup — if you've done this kind of thing before

HTTPS SSH https://github.com/ alflin /recipes.git

Get started by creating a new file or uploading an existing file. We recommend every repository include a README, LICENSE, and .gitignore.

choose upload an existing file, and drag the Jupyter Notebook, `piMonteCarlo-initial.ipynb`, which you can find in the Learning Management System (LMS) for this course.

Also, take note of the URL for your GitHub repository.

Next, go to Google Colab, https://colab.research.google.com/.

In the "Open Notebook" dialog box, click on the GitHub tab, find your repository, and click on `piMonteCarlo-initial.ipynb`. Run (***play*** button or hit `<shift><enter>`) the first execution group, and then fill in the code in the second execution group for a "naïve" implementation of the Monte Carlo simulation to estimate pi.

By "naïve," we mean a straightforward approach, choosing an N, and then looping N times, creating a random point (both x and y values), then determining whether that point is inside or outside the quarter circle (think distance and Pythagorean Theorem).

Run your "naïve" code for a variety of N values. What do you notice about the estimate for π as N gets larger?

Solution for the “naïve” code:

```
#Naive approach
n = 10000000
inCircle = 0
start_time = datetime.datetime.now()

for i in range(n):
  x = random.random()
  y = random.random()
  if x**2+y**2<=1.0:
    inCircle +=1

end_time = datetime.datetime.now()
elapsed_time = (end_time - start_time).total_seconds()
pi_est_python = 4.0*inCircle/n
print(pi_est_python,elapsed_time)
```

After you have the “naïve” simulation working, run the next two execution groups, comparing the compute time for the same values of N in each execution group.

In the second execution group, Jupyter is running in parallel. This simulation is exceedingly parallelizable, one random point does not depend on any of the others.

The third execution group doesn’t have any loops coded! It seems very ***inefficient***. Why are we saving all the generated points in a vector? We only need them once.

But this is the fastest.

And this is a main theme in the course. When we design an algorithm, we consider ***algorithmic efficiency***. But we also have ***hardware optimization***.

We will focus primarily on, ***algorithmic efficiency***, but as we encounter implementations we will take note of seemingly inefficient code which uses ***hardware optimization***.

So, let’s consider algorithmic efficiency from a mathematical standpoint.

You may remember some algorithms from CS 240 – Computer Science 2.

Let's consider three of them:

- Binary Search
    - Do "guess an integer."
    - What is the maximum number of questions getting a "high/low" response for a given n ?
    - $log_2(n) = \lg(n)$
    - Note that in this class we will follow the convention that the textbook uses, log_2 is written as $\lg(n)$.
- Selection Sort
    - Do you remember the implementation depended on nested loops?
    - Which implied $O(n^2)$ efficiency.
- Mergesort
    - A recursive algorithm, the problem is sub-divided by splitting in half, then merging sorted sub-lists.
    - Which implied $O(n \cdot \lg(n))$ efficiency.

Note that we will take a much deeper dive on each of the above sorting algorithms.

<u>In-class w/***assigned groups of four (4)***, write 0. and 1. on board as ground rule(s):</u>

0. No devices.
1. Intro: Name, POE, Icebreaker.
2. Pick someone on the group who will report out.
3. Discuss the following.

This will introduce us to some of the mathematics we will use in class this semester.

For each function $f(n)$ and time t in the following table, determine the largest size n of a problem that can be solved in time t, assuming that the algorithm to solve the problem takes $f(n)$ ***micro***seconds.

Note: 1 second = 10^6 micro-seconds, and although n might not be a whole number, you would round up to get a whole number, since in practice you typically only have inputs of integer size.

For example,

$$lg(n) = 10^6$$

$$2^{lg(n)} = 2^{10^6} = 2^{1000000}$$

We leave that solution as is, since $2^{1000000}$ is too huge to calculate.

If you can't calculate $log_2(n) = \lg(n)$ directly, you can use the identity $log_2(n) = \frac{log_{10}(n)}{log_{10}(2)}$

	t = 1 second = 10^6 micro-seconds	***t*** = 1 minute = $60*10^6$ micro-seconds	***t*** = 1 hour = $60*60*10^6$ micro-seconds	***t*** = 1 day = $24*60*60*10^6$ micro-seconds
$lg(n)$	$lg(n) = 10^6$ $2^{lg(n)}=2^{10\wedge 6}$ $n=2^{10\wedge 6}=\boldsymbol{2^{1000000}}$ this is huge…			
n				
$n\,lg(n)$	***Guess-and-check is fine, or some type of solver.***	***Guess-and-check is fine, or some type of solver.***	***Guess-and-check is fine, or some type of solver.***	***Guess-and-check is fine, or some type of solver.***
n^2				

	t = 1 second = 10^6 micro-seconds	***t*** = 1 minute = $60*10^6$ micro-seconds	***t*** = 1 hour = $60*60*10^6$ micro-seconds	***t*** = 1 day = $24*60*60*10^6$ micro-seconds
$lg(n)$	$lg(n) = 10^6$ $2^{lg(n)}=2^{10\text{^}6}$ $n=2^{10\text{^}6}=\boldsymbol{2^{1000000}}$ this is huge…	$lg(n) = 60*10^6$ $2^{lg(n)}=\boldsymbol{2^{60*10\text{^}6}}$ Even bigger…	$lg(n) = 60*60*10^6$ $2^{lg(n)}=\boldsymbol{2^{60*60*10\text{^}6}}$ Even bigger…	$lg(n) = 24*60*60*10^6$ $2^{lg(n)}=\boldsymbol{2^{24*60*60*10\text{^}6}}$ Wow…

n	n= 10^6
	n= ***1,000,000***
		n= $60*10^6$
		n= ***6*10⁷***
			n= $60*60*10^6$
			n= 3***.6*10⁹***
				n= $24*60*60*10^6$
				n=***8.64*10¹⁰***

$n\,lg(n)$	Sage/CoCalc:
	`eqn=10^6==`
	`x*log(x,2)`
	`eqn.find_root`
	`(30000,90000)`
	n= ***6.3*10⁴***
	n= ***62746.1***
		Sage/CoCalc:
		`eqn=60*10^6`
		`==x*log(x,2)`
		`eqn.find_root`
		`(1e6,3e6)`
		n= ***2.8*10⁶***
		n= ***2,801,417.9***
			Sage/CoCalc:
			`eqn=60*60*10^6`
			`==x*log(x,2)`
			`eqn.find_root`
			`(1e8,3e8)`
			n= ***1.3*10⁸***
			n= ***133,378,058.9***
				Sage/CoCalc:
				`eqn=24*60*60*10^6`
				`==x*log(x,2)`
				`eqn.find_root`
				`(2e9,3e9)`
				n= ***2.756*10⁹***
				n= ***2,755,147,513.2***

n^2	$n^2 = 10^6$ n= ***1,000***	$n^2 = 60*10^6$ n= ***7,746***	$n^2 = 60*60*10^6$ n= ***6*10⁴=60,000***	$n^2 = 24*60*60*10^6$ n= ***2.94*10⁵***