The Power of Randomness.

Randomness is a funny thing. The topics of todays lecture is algorithms which might be really hard to solve without some creativity.

Driving

First algorithm, remembering how to drive. There I was going 17 mph on a 40 mph road in the suburbs when I forgot which foot goes on which pedal. I was approaching a car stopped at a light, and had to think fast. I had the choice between the following two algorithms:

algorithm 1:
look in the glovebox for the manual and find the page on which pedal does what

algorithm 2:
just guess.

Alg 1 requires what, $\log n$? n steps? In a time constrained scenario I can't afford that. Choosing algorithm 1 may seem like the only idea but in the time constraint. There is failure (and death) with probability 1. The second algorithm seems ridiculous, but it only takes constant time! It has a failure rate of only 1/2. I just guessed with pedal was gas and which was brake, and stopped just in time. The power of randomness!

Primality

Lets talk about a more realistic problem, primality testing. On input n, determine if n is prime or not. There is an ~~exponential time~~ algorithm test all factors up to $\sqrt{n}$. Can we do better? Turns out with the power of randomness we can!

Before seeing the next algorithm, try to come up with your own deterministic primality testing algorithm.

Recall Fermats Little theorem: prime $p \Rightarrow a^{p-1} \equiv 1 \pmod{p}$.
we can use this to test primality of a general n.

```
def f(n)
    a = math.random(2, ... n-2)
    if a^(n-1) ≡ 1 (mod n)
        return true.
    else
        return false.
```

the arithmetic can be done faster using repeated squaring but this atleast takes polytime. If n is prime it always returns true, but if n is not prime, it doesn't always return false. There do exist numbers which may pass this test and not be prime. (carmichael numbers. They are rare but they do exist. Only 7 less than 10,000. Odds you hit one are negligible, but over zero.

Okay, randomness did enable this faster primality testing algorithm. was there not a fast deterministic primality testing algorithm? It took humanity until 2004!! to find one. [AKS04] uses derandomization techniques to ~~guess~~ give a deterministic polytime primality testing algorithm we have known about primality for millenia, and took us this long to find this algorithm. Its the case that AKS04 is too advanced to present today, but just know it exists and was an achievement.

What about algorithms for computing pi? $\pi = 3.14\ldots$ This is actually surprisingly hard. Everyone just uses math.pi or something, and assume its stored to sufficient precision. But if you had to compute it, how would you?

Heres a half decent try. Recall $\tan(\pi/4) = 1$, so compute

$$\pi = 4\tan^{-1}(1) = 4(1 - 1/3 + 1/5 - 1/7 + 1/9 - \ldots)$$

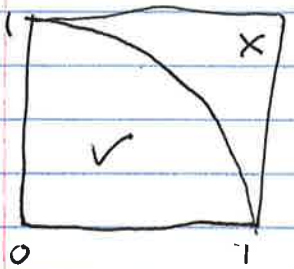
by rules of the Taylor series, we can compute $4\tan^{-1}(1)$. The problem with this is each term takes approximately the same or more operations to compute, but the amount we get to π gets smaller and smaller. For 6 digits of precision, we need like a billion operations :C

Heres a better way. Just guess! Throw a fistfull of darts at the board. Whats the probability that it hits? Its proportional to the area of the circle. The probability a dart hits is the area!

$$P[\text{dart hits circle}] = \frac{\text{dart hits}}{\text{total darts}} \approx \frac{\pi r^2}{4r^2} \approx \frac{\pi}{4}$$

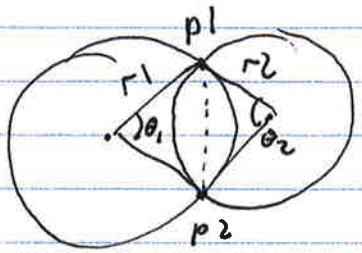
so we can guess π by throwing darts! Lets optimize a bit

$$P[\text{darts hit greater}] = \frac{\text{dart hits}}{\text{total darts}} = \frac{\pi r^2/4}{r^2} = \frac{\pi}{4}$$

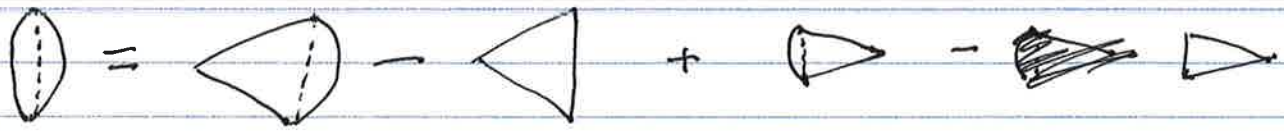


a constant speed up to compute the same value. How do you determine if a dart hits the circle though? If $d = (x, y)$, then $x^2 + y^2 = r^2$ is the boundary. For the unit circle and sphere, if $x^2 + y^2 \leq 1$, then its a hit, $x^2 + y^2 > 1$ is a miss. as we throw more and more darts, the error rate of our algorithm from approximating π will converge. It will do so slowly, but faster than computing a taylor series thats for sure.

Okay I admit, its not faster than computing the Taylor series, and its less accurate, but it generalizes in ways that the Taylor series solution doesn't. See newer programming problem, fortnite.



Heres a classical geometry way. let circles 1, 2 intersect at points p1, p2. Then we have the triangles of lengths r1, r1, x & r2, r2, x where x is the distance between p1 and p2. Then we also circle arcs as pie pieces. This gives us a way to compute the slither, the almond.



I have no idea how to program this but the steps are:

1 find p1, p2 from the radii and centers.
2 find distance between p1, p2, called x
3 let t1 = area(r1, r1, x) } heron formula
4 let t2 = area(r2, r2, x)
5 use c_1, r_1 ~~to find~~ $c_2 r_2$ to find θ_1
6 " find θ_2
7 sector 1 = $\theta_1/2\ r_1^2$
8 sector 2 = $\theta_2/2\ r_2^2$
9 return s1 - t1 + s2 - t2

Or, using the method of Monte-Carlo. Just guess the area! a little math must be done to determine the surrounding bounding box, but that's not too bad, and an overestimate won't hurt. This method works for this problem really since the question only asks for two digits of precision.

One final algorithm. Given a graph G, 2-color the graph so atleast half the edges have endpoints with different colors.

Guess each node to be either red or blue. Then each edge has a $1/2$ chance to be of different or same endpoints. If $> E/2$ edges have different endpoints, halt. else, recolor. Because the endpoints of each are chosen randomly, and independently, the chance our first coloring is valid is very high. The probability we have successive colorings without the property should converge exponentially.